

Clarion™ Databases & SQL

**A Practical Handbook of
Database Design, Flat File &
SQL Data Management,
and Clarion data
handling tricks**



Edited by David Harms

Clarion Databases & SQL

Edited by David Harms

Clarion Databases & SQL

David Harms, Editor

Copyright © 1999-2004 by CoveComm Inc.

Published by:

CoveComm Inc.
1036 McMillan Ave
Winnipeg, MB R3M 0V8

All rights reserved. No part of this book may be reproduced or transmitted in any form by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. For information on getting permission for reprints or excerpts, contact books@clarionmag.com.

See <http://www.clarionmag.com/books> for source downloads and errata

Library and Archives Canada Cataloguing in Publication

Clarion databases & SQL : a practical handbook of database design, flat file and SQL data management, and Clarion data handling tricks / edited by David Harms.

Includes index.

ISBN 0-9689553-3-9

1. Clarion (Computer program language) 2. SQL (Computer program language) 3. Database design. I. Harms, David (David Gerhard), 1959-. II. Title. III. Title: Clarion databases and SQL.

QA76.73.C22C527 2004 005.75'85 C2004-903849-4

The information in this book, and any source code and/or information in downloads referenced in this book, is distributed on as "As Is" basis, without warranty. While every precaution has been taken in the preparation of this book, neither the publisher nor the editor assumes responsibility for errors or omissions in the book, or in the instructions and/or source code in downloads referenced in this book.

Clarion™ is a trademark of SoftVelocity, Inc. This and other trademarks which may appear in the book are used in an editorial fashion only and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Printing History

July 2004 First Edition

ISBN 0-9689553-3-9

Printed and bound in the United States of America

Editor: David Harms

David Harms is a long time Clarion developer, and the editor and publisher of *Clarion Magazine* (www.clarionmag.com), which he founded in 1999. He is also co-author with Ross Santos of *Developing Clarion for Windows Applications*, published by SAMS (1995), and has written or co-written several books on Java.

Terms Of Use

All rights reserved. No part of this book may be reproduced or transmitted in any form by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher.

As a licensed user of this e-book you may keep multiple copies for your personal use only.

Copyright © 1999-2007 by CoveComm Inc.

Published by:

CoveComm Inc.
1036 McMillan Ave
Winnipeg, MB R3M 0V8

www.clarionmag.com

Clarion Databases & SQL

Table of Contents

Introduction to Databases

Designing Databases.....	1
Handling Many-To-Many Relationships	9
Managing Complexity, Rule 1: Eliminate Repeating Fields	21
Managing Complexity, Rule 2: Eliminate Redundant Data	29
Managing Complexity, Rule 3: Eliminate Columns That Don't Belong	39
Managing Complexity, Rule 4: Isolate Independent Multiple Relationships.....	45
Managing Complexity, Rule 5: Isolate Semantically Related Multiple Relationships	49
Displaying Normalized Data	55
Displaying Many-To-Many Relationships	67
True Confessions: A Tale of Two Users	85

Using Topspeed Files

Using Dynamic Indexes With TPS Files	91
Using The TPS ODBC Driver	97
Reading Tables With ADO	103
Accessing TPS Files Via ASP	109
Using Example Files With TPSFix.....	117
Topspeed Driver Error Codes	119
Troubleshooting TPS File Corruption	125
Resolving Network And Other File Problems.....	129

General SQL

An Introduction To SQL.....	141
Getting Into SQL On The Cheap	167
How To Convert Your Database To SQL	175
Converting TPS To MS-SQL	183

Avoid My SQL Mistakes!.....	197
SQL Data Types Comparison.....	201
The SQL Answer Cowboy.....	209

Open Source SQL

Using Clarion With MySQL.....	221
MySQL/MyODBC Notes	241
MySQL: InnoDB Tables And Transactions	247
Large Table Performance in MySQL	265
Getting Started With PostgreSQL.....	273

Oracle

Referential Integrity In Oracle.....	295
AutoNumbering In Oracle	299
Transactions In Oracle	305

MS SQL

Migrating The Inventory Application To SQL Server	317
Using SQL Server's Data Transformation Services.....	341
Converting Data With Linked Servers.....	357
Converting The Inventory Example - Calling Stored Procedures	371
SQL Identity: Another Approach	399
Creating Utilities For MS SQL 2000.....	405
Generating MS SQL Server Side Triggers	415
Date Filtering with MS SQL.....	429

Book Reviews

Book Review: PostgreSQL Developer's Handbook.....	435
Book Review: SQL Tuning	437
Book Review: SQL In A Nutshell	441
Book Review: Managing & Using MySQL	443

ABC Database Class Design Notes

Inside ABC: FieldPairsClass and BufferedPairsClass	447
Inside ABC: The FileManager.....	459
Inside ABC: The RelationManager	483
Inside ABC: The ViewManager	495

Database Tips & Techniques

Clarion File Access Basics	507
Managing Table Opens In ABC	517
Creating ODBC Data Sources At Runtime.....	527
Securing Remote Database Connections With SSH Tunneling	535
Using Client-Side Triggers In Clarion 6.....	543
Working With Control Files	547
Changing Dictionaries	563
Alias - Who Was That Masked File?.....	567
Propitious Memory Corruption.....	577
Detecting Duplicate Records	585
NAME() Comes Of Age	591

Appendices

Appendix A: Getting Support.....	601
Appendix B: Related Articles	603
Author Index	605
Index	609

Introduction to Databases

DESIGNING DATABASES

by David Harms

Software development is a bit like building a house. Many traditional programming tools correspond to hammers, nailers, saws, drills, and other power tools. In this analogy the tools help you shape the raw material into a structure. Newer software development tools, including Clarion, do the software equivalent of prefabricating large sections of a building. And very sophisticated tools like the Wizards let you create the equivalent of an entire building with a certain look and feel.

This sort of technology is a great boon to many developers. Still, even the Wizards can't (yet) read your mind and know what kind of an application you'll want. They work on the basis of the application style you choose (or create) and the data dictionary you create (or use, if one already exists).

From the Wizard's perspective, application creation seems quite simple. All the Wizard needs to know is the expected appearance and behavior of the application and the data structures. Simple, right? All the developer has to do is choose how the application should look, how it should work, and what data it should store, and writing software becomes (in theory) a license to print money.

This is, of course, the reason the world still needs software developers as much as it needs software development tools. And as tempting as Wizzy technology is, there's still a lot of

work for the developer to do before any given application development process hits its automation phase.

Begin At The Beginning

If you've accepted the basic assumptions of the Clarion way of building applications (MDI frame and windows, the browse/form paradigm), then you're probably not going to quibble with the fundamentals of your application's appearance. You are going to be concerned about the data dictionary (which in Clarion will determine a lot of the application's behavior). Therefore much depends on the quality of the database design.

But how do you arrive at your database design? You begin with some sort of requirement, whether that's a formal document or just some half-baked ideas rattling around in your head. From this requirement you can begin to formulate some possible approaches. The possibilities you come up with will depend a lot on your past experience with application development, your familiarity with existing, similar applications, and your knowledge of what options Clarion makes available.

Application design, which ultimately includes database design, is itself a rather large subject, and there are any number of techniques you can use to try to flesh out what the application will need to do. And you can be sure that whatever the initial requirement describes, the final product will be something more complex.

A Student Tracking Application

Somewhat irresponsibly, I'm going to leave aside the principles and techniques of overall application design and focus on the data dictionary. My assumption is that you're comfortable with the kinds of applications Clarion creates by default (MDI, browse/form), and that the user's requirements very conveniently do not deviate in any significant way from the functionality provided by Clarion with the standard ABC templates.

Imagine an application which will be used to track university students and the courses for which they register. The application should be able to generate reports showing students in a given course and the courses taken by a given student, and should provide standard student information such as address and phone number.

It's vital in any development effort to determine how to store the data in the most efficient, reliable, and useful way. As of this writing, for Clarion developers, this means using some sort of relational database. (If you're thinking about object-oriented databases, or object-relational databases, that's still in the future for us and for a lot of other developers too.)

Relational Databases

Clarion is built around the concept of the relational database. As you might guess from the name, a relational database deals with groups of information which are related in specific ways. These groups are typically called tables if you're dealing with an SQL database, or files (at least in the Clarion world) if you're using a non-SQL database. For the rest of this chapter I'll use the term *file* or *data file* to describe such a group, since I'll be creating a Topspeed database.

Each data file can contain zero to many records (rows in SQL parlance) of information. For instance, a file of employee information would contain one record for each employee, and within that record there will be individual fields, such as one for name, another for telephone number, and so on.

One key rule of relational databases is that each record in a table must have a field which is a unique identifier. In Clarion this value is usually defined as a LONG integer, and its value is typically created automatically (by the ABC class library or by code generated by the templates). This field should also not normally be available to the user to change.

Another of the rules which describe good relational database design says that data duplication must be kept to an absolute minimum. This rule is closely related to the first rule, as you will see.

Armed with these two rules you can avoid a lot of problems in your database design.

In the student tracking application you might begin with a table to hold student information. Fields could include FirstName, MiddleName, LastName, as well as fields for the address and telephone numbers.

Before your rush to the dictionary editor and begin creating the Student file (or any other file) ask yourself if you really know the extent of the data. Will the fields you propose be sufficient? Will each student, for instance, have only one address? University students are notoriously mobile. My personal best was 13 addresses in three years (and no, I wasn't running from the law). Often students will have a permanent or home address and a residence address, and perhaps a third address when away on a work term. If you don't have the familiarity with the data to know these sorts of things, then you must rely heavily on those who do. This can be trickier than it sounds, because oftentimes people who are familiar with the data will omit details that to them are second nature, but non-obvious to anyone else.

Avoiding Data Duplication

Clearly, you'll need to store more than one address per student. If you know the exact number of addresses, you can simply add extra fields to the file. But if some students will have one address, and then others two, then there will be a lot of wasted space in the database. And if some students have more than two addresses you won't be able to accommodate the extra information.

Some developers use arrayed fields in these kinds of situations. Arrays are a convenient solution from a coding perspective because you can use a consistent set of labels when processing the data. All you need to do is change the subscript. But arrays don't solve the problem of wasted space, and they tend to introduce a lot of other problems when it comes to sorting data or using the fields in list boxes. Arrays do have a place, but they should be used *only* when they are the best solution.

In the case of student addresses, a better solution is to have a separate address file, and link it to the Student record by using a unique identifier. The record structures I've designed look like this:

```
Student          FILE, DRIVER ('TOPSPEED'), PRE (STU)
                 ,CREATE, BINDABLE, THREAD
StudentIDKey    KEY (STU:StudentID), NOCASE, OPT, PRIMARY
Record          RECORD, PRE ()
FirstName       STRING (30)
MiddleName      STRING (30)
LastName        STRING (30)
StudentID       LONG
                 END
END

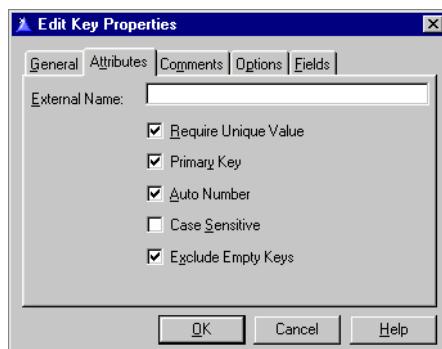
Address          FILE, DRIVER ('TOPSPEED'), PRE (ADD)
                 ,CREATE, BINDABLE, THREAD
AddressIDKey    KEY (ADD:AddressID), NOCASE, OPT, PRIMARY
StudentIDKey    KEY (ADD:StudentID), DUP, NOCASE, OPT
Record          RECORD, PRE ()
AddressID       LONG
StudentID       LONG
Address1         STRING (30)
Address2         STRING (30)
City             STRING (30)
StateProv        STRING (2)
                 END
END
```

You can see that both files have a field called `StudentID` which is a `LONG`. This field serves two purposes. It uniquely identifies each student record, and it also provides a link between the student record and one or more address records.

For fields that you'll be using in more than one file, create a file in the data dictionary of type **Pool** and enter the base field types there. When you need a field of a type you've defined, click on the button to the right of the **Derived From** field in the **New Field Properties** window and choose from the Pool file. Do this *before* you enter a value in the **Field Name** field or you'll also have to reenter the **Prompt Text** and **Column Heading** values.

Simply having a linking field isn't sufficient. You also need to declare a key on the linking field in each file so that the linked record can be quickly retrieved. (This is a Clarion dictionary requirement, and also a requirement for non-SQL database drivers. If you're using an SQL database, you don't actually need to create the key – or index, in SQL parlance – in the database itself, but performance will suffer without it if you have large data sets. See "An Introduction To SQL," p. 141 for more information.) In the Student file the **StudentIDKey** is a unique key (it lacks the DUP attribute). You set this option on the **Key Properties** dialog, on the **Attributes** tab, as shown in Figure 1.

Figure 1: The key properties Attribute tab for **StudentIDKey** in the **Student** file



StudentIDKey in the **Student** file has several other attributes. It's a *primary key*, which means that the key is unique and contains a value for every record in the database. You cannot have a null value in a field that is in a primary key. The primary key ensures that there is always a way to retrieve a specific record in a file. This is essential in an SQL environment where there is normally no equivalent to a record number as there is in a flat file database.

This key also has the **Auto Number** box checked. Checking **Auto Number** won't make any difference to the way the file is declared, but it will cause code to be generated that creates a unique sequential number for the key value for every record added to this table.

Note: QuickLoad now has an **Auto-Increment Key** option that will create an appropriate autonumbering key.

Introduction to Databases

`StudentIDKey` in the `Address` file is slightly different. Here the `StudentID` field will contain a value obtained from the `Student` file, so rather than a primary key this is a *foreign key*. The `Address` file does have its own primary key in `AddressIDKey`. In an SQL database every file (table) should have a primary key.

Just as creating fields doesn't automatically give you a key, creating keys doesn't automatically give you a link between files. This has to be defined in the data dictionary so that the appropriate code can be generated to manage the relationship. In the case of an SQL database, you have the option of also defining the relationships in the database itself, in which case the database server can enforce any constraints. In either case, you should have the relationship defined in the dictionary.

This particular link on `StudentID` is a one-to-many relationship between the student and address records. Any one student can have one (or I suppose zero) or more addresses.

To define this relationship, highlight the `Student` file in the dictionary main window and click on the **Add Relation** button. The **Relationship** properties window appears, and it has one field filled in, **Relationship to Student**, which defaults to **1:Many**. (The only other option is **Many:1** which you could choose if you wished to create or edit the relationship from the perspective of the `Addresses` file.)

Choose `StudentIDKey` as the primary key, `Address` as the related file, and `StudentIDKey` as the foreign key.

You'll see the key fields listed under **Field Mapping**. You can choose to automatically map the fields by name or order (either will work in this case), or you can double-click on the individual fields to specify which field is the linking field. This is more of an issue when creating relationships with multi-component keys.

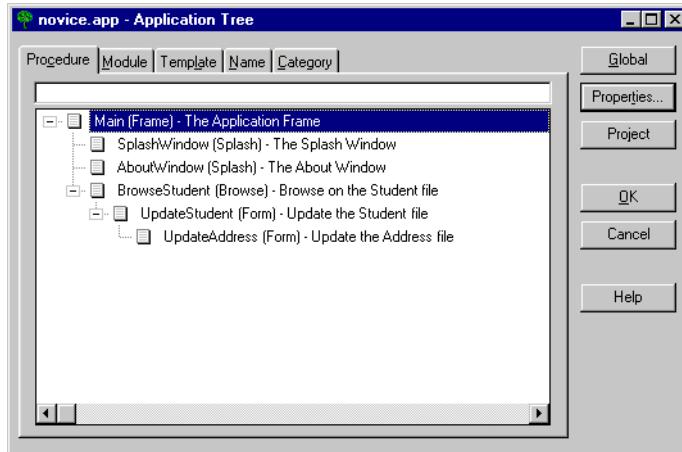
All that's left is to choose update and delete constraints, which determine whether certain actions involving related files will be allowed. The options are **No Action**, **Restrict**, **Cascade**, and **Clear**. The last three options are repeated with the annotation (**Server**) meaning that the restrictions exist but will be managed by the SQL database server rather than by Clarion code.

In a **Restrict** constraint you may not change or delete a primary key value if a related record exists. In a **Cascade** constraint any change to the primary key value will result in the change being made in any related records, and on a delete the related records will also be deleted. A **Clear** constraint clears any foreign key values on a change or delete of the primary.

In general you don't want to give your users access to linking fields anyway, so **Update** constraints shouldn't be a major issue. **Delete** constraints require more thought, and usually the choice is between **Restrict** and **Cascade**. If someone deletes a `Student` record you're probably going to want to delete all the `Address` records as well.

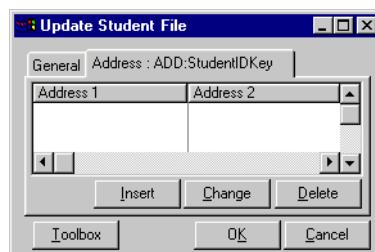
Now that you've determined the relationship you can go ahead and generate an application using the Application Wizard. Choose *only* the Student file when asked which files you want to be able to browse. Figure 2 shows the resulting application tree.

Figure 2: An application which browses the Student file.



Note that although you only specified that you wanted to browse the Student file Clarion has created a an UpdateAddresses form, called from the UpdateStudent form. On the UpdateAddresses form, on the second tab, you'll see a browse for the Addresses file, as shown in Figure 3.

Figure 3: The Addresses tab on the UpdateStudent form.



The Addresses browse exists only because of the relationship defined in the data dictionary between Student and Addresses. The Wizard simply populated a browse control template on the tab and set it to show only those address records which are linked by StudentID to the current student record.

Although Clarion by default puts the address browse on the student form, you can also place an address form on the student browse window if you prefer. You'll have to do this manually as the templates don't have this option.

A Course, Of Course

This application now has a minimal capability of handling student information. The other main requirement is to track which courses students take.

It's easy enough to define a record to represent a course. You might want fields such as `CourseName`, `StartDate`, `Instructor`, `Location`, and so forth. But think about the actual data. You have many students and many courses. In other words this is a many-to-many relationship, not a one-to-many relationship. How can you represent this information in the most compact way possible? And does the data dictionary support many-to-many relationships? Read on.

Source code

See "Appendix A: Getting Support," p. 601, for information on how to get the source accompanying this book.

- `v1n4novice3.zip`

HANDLING MANY-TO-MANY RELATIONSHIPS

by David Harms

In the previous chapter I began developing a data dictionary and application to track information about students attending a university or college. I began by defining a Student file as well as an Address file, on the premise that students may have one or more addresses. This dictionary design reflects a one-to-many relationship between students and addresses, or, if you look at it from the other side, a many-to-one relationship between addresses and students.

This example relationship is obvious and easy to understand. Real data, however, is often a bit more complex. Sometimes any number of records from one file can be related to any number of records from another file. These many-to-many relationships are quite common in databases, and most likely you'll have to know how to handle them. In this chapter I'll examine such a relationship and outline how it can be defined in the data dictionary.

Adding Courses

It's now time to expand the demonstration application to handle not just students and addresses, but the courses for which students can register. You can probably guess at some of the fields required to describe courses:

- ID (keeping in mind the previous chapter's discussion about unique IDs)
- course title
- start date
- end date
- number of registrants
- instructor (suggests a many-to-one link to an instructor file)

As you create the data file, keep in mind any fields which are defined in pool data, and consider whether any other fields should be added to the pool. In the case of the Course file the ID can come from the pool data (where it has the **Do Not Populate** flag set). There are two date fields in the file and you will want them to have a standard date format, and you may wish to make them spin boxes as well. In either case, a pool Date field is a good idea. Figure 1 shows the Course fields in the data dictionary, and Figure 2 shows the Course file's keys.

Figure 1: Fields for the Course file.

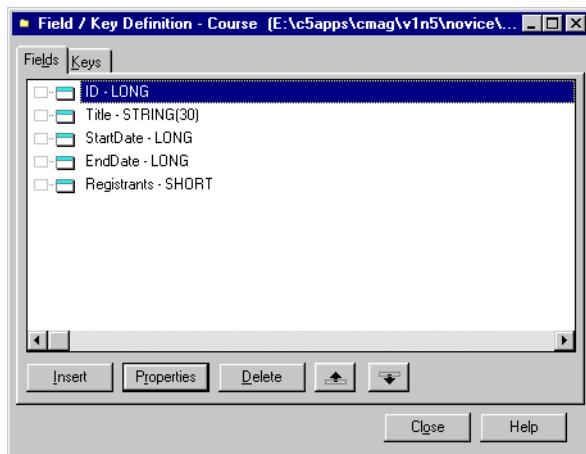
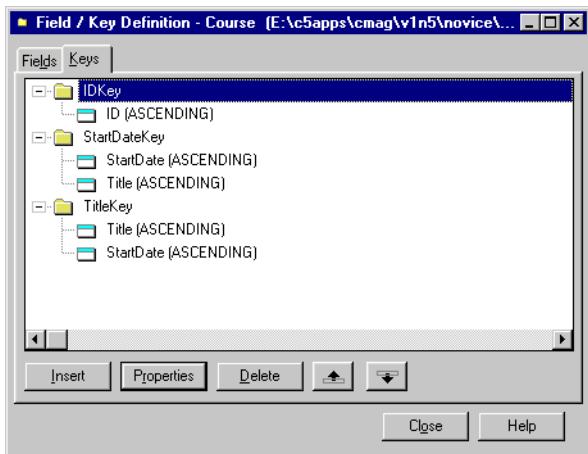


Figure 2: Keys for the Course file.



Linking Courses To Students

Now you have a way to store courses and a way to store students. How do you link students to courses? The problem is that one student can take a number of courses, and any course can be taken by a number of students. The relationship is many-to-many, diagrammed in Figure 3. The two triangles indicate the “many” aspect of the relationship.

Figure 3: A many-to-many relationship.



If you examine the data dictionary’s relationship editor you won’t find any options for creating a many-to-many relationship, because it’s almost impossible to create using just two files.

The problem is that from a student perspective you need to store an unknown number of course IDs, and from the course perspective you need to store an unknown number of student IDs. Some developers approach this the same way they do many-to-one relationships: they use arrays.

As I indicated in the previous chapter, arrays are generally a bad idea for linking files, and they're even worse in a many-to-many situation. Arrays are, by definition, limited in size (at least in Clarion), which means you have to make the array size as large as the highest possible value, thereby wasting a lot of space. Furthermore you cannot use arrayed fields in keys. That's not a problem in a many-to-one where only one side needs to be keyed, but in a many-to-many both sides need to be keyed. You need to see which courses a given student takes, and which students are in a given course.

The answer is to use a file as an intermediary between Student and Course, as shown in Figure 4.

Figure 4: An intermediary for managing the many-to-many relationship.



In Figure 4 the single triangles indicate the “one” side of the relationship and the double triangles indicate the “many” side. As this diagram shows, the many-to-many has been broken down into two many-to-one relationships. This is the standard approach to handling many-to-many situations.

The linking Registration file is simplicity itself – it needs to contain only three fields: a unique autonumbered registration ID, a student ID and a course ID. (You might want to add several additional fields, however, including the date the registration was taken.)

Note: In a TPS or other flat-file (i.e. non-SQL) database you can get away without the unique autoincrement key for this record, but it's a good idea to have it anyway as you may wish to link other records to the registration record.

Handling Many-To-Many Relationships

Figure 5 shows the fields used in the linking file, and Figure 6 shows the keys.

Figure 5: The Registration file fields.

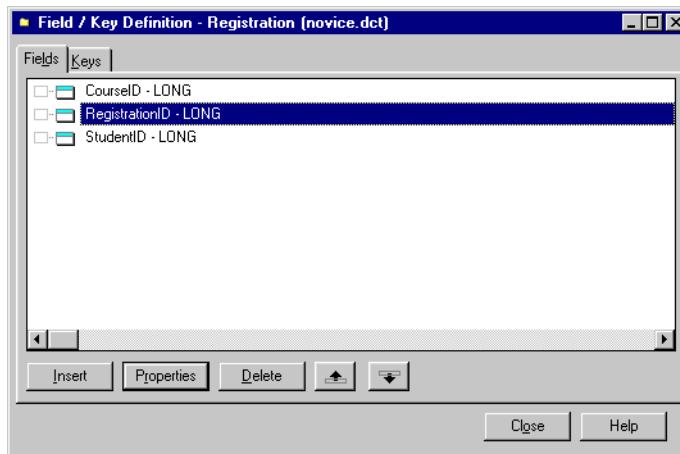
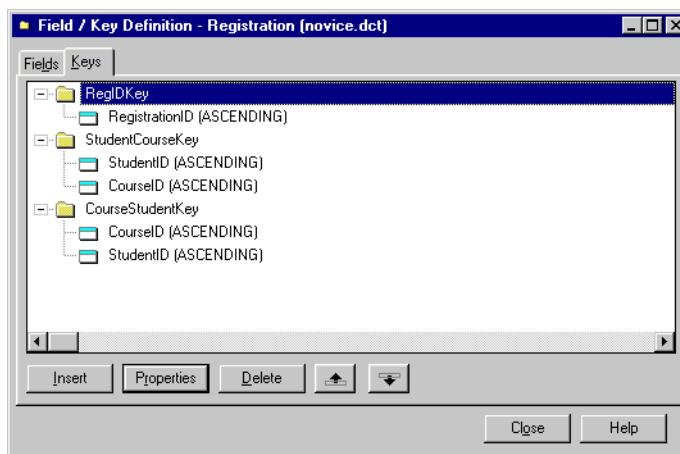


Figure 6: The Registration file keys.



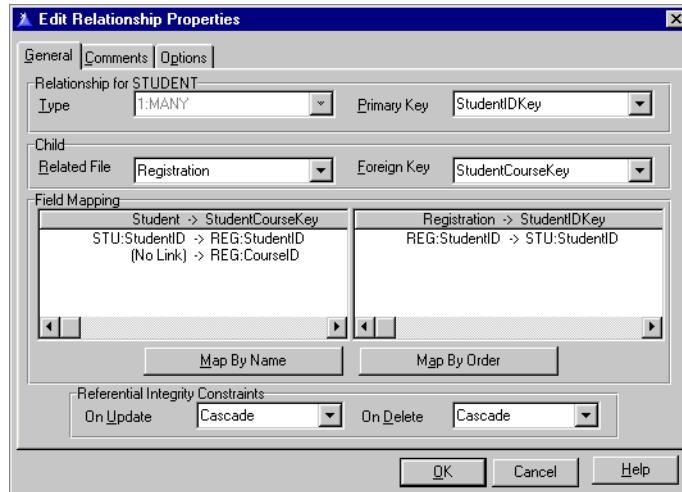
StudentCourseKey and CourseStudentKey both have the course and student IDs as elements, and if you look at the data dictionary you'll see that they have the UNIQUE attribute as well. This prevents a student from being registered for the same course twice.

Now define the relationships in the dictionary editor. Although both the keys in Registration have two elements, you only want to make the links on the first elements

Introduction to Databases

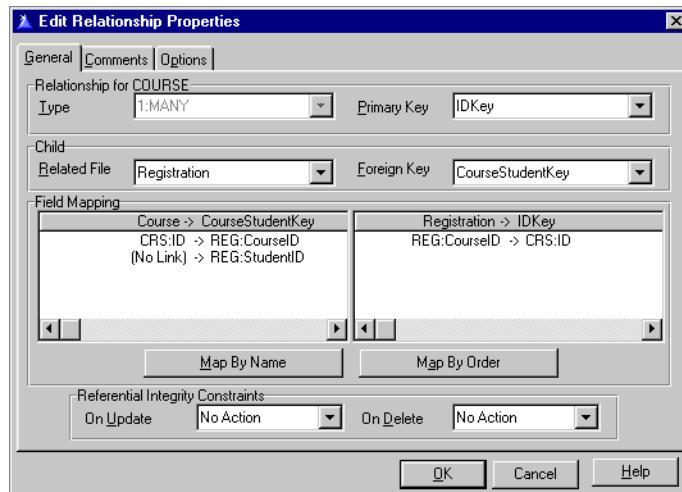
in the key in each case. Figure 7 shows the relationship between Student and Registration.

Figure 7: The relationship between Student and Registration.



A similar relationship exists between Course and Registration, as shown in Figure 8.

Figure 8: The relationship between Course and Registration

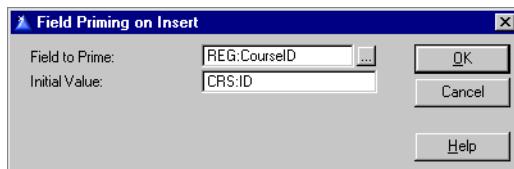


If at this point you use the browse wizard to create a browse and update form for Course, you'll find a browse of Registration records on the Course update form. That may not be the final place you'd like to have it, but it will show you how the link works. (Another option is to populate a browse of Registration records below the Course browse, and restrict its records based on the relationship with Course.)

You'll want to display the student name on the Registration browse rather than just the ID. Highlight Registration in the file schematic and click **Insert**. Select the Student file from the related files list. This will ensure that the related Student record is retrieved for each Registration record (see the example application).

Since the only fields in the Registration file are two LONG IDs, you'll need to set these values when you add a registration. Whether you're updating Registration records from a browse on the Course update, or from a child browse you've placed on the Course browse, you know that the current Course record is in memory. Use **Field Priming on Insert** on the Registration update form to preload REG:CourseID with CRS:ID. Figure 9 shows the appropriate **Field Priming on Insert** setting.

Figure 9: Priming REG:CourseID on insert.

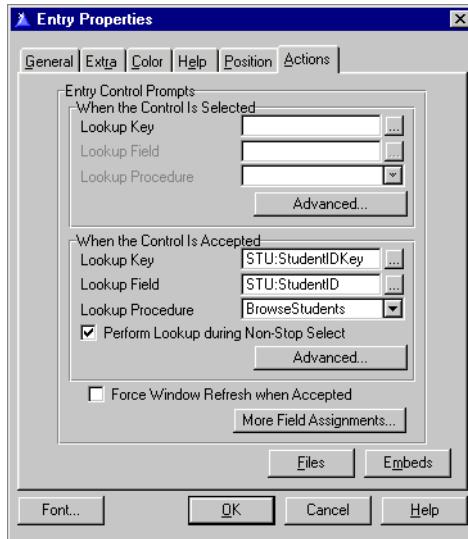


Since you already have REG:CourseID there's no need to display it on the window; delete the entry field. Now all the user has to do is choose a student. You probably don't want the user entering in the student's ID directly, so you should provide a lookup on a list of students.

Introduction to Databases

One way to do this is to set up the REG:StudentID field's actions to do a lookup, as shown in Figure 10.

Figure 10: Lookup settings for REG:StudentID.



After entering the settings in Figure 10 and saving the changes, populate the FieldLookupButton template on the window, and on its **Actions** tab select ?REG:StudentID. Now you have a button you can use to force a lookup on the Student browse.

Next bring up the property window for REG:StudentID. Check the **Hide** box so the ID won't be visible. In the form's file schematic, click on the Registration file and choose Student from the **Related Files** tab. Populate two string fields on the form and on the string properties window, Use field, enter REG:FirstName for one field and REG:LastName for the other (or use the field lookup button to the right of the Use prompt). The relation manager will take care of looking up the student record from the ID and will display the name of the student associated with the registration.

Note: This is just one approach to selecting records. Another would be to use the FileDrop template (though this code is due for major revision and may not be a safe choice). You may also want to consider a third party solution like ProDomus' highly-regarded PDLookup Toolkit (available at www.prodomus.com) which adds Quicken-style incremental lookups. Also read Tom Ruby's "Displaying Many-To-Many Relationships," p. 67.

Bending The Rules

If you look at the data dictionary for the example app you'll see that the Registration file has a Registrants field which is defined as a SHORT variable. Assuming that a successful link is created between courses and students it should be possible to calculate this value, so it isn't strictly necessary to keep it on the course record.

In fact, good relational design suggests that this is unnecessarily duplicated data and should not be stored. In reality there are often trade-offs involved in designing a database. Your aim should be to avoid situations where conflicting data can exist, but sometimes performance requirements must also be addressed. If one of the requirements is to display the number of registrants on a browse of courses, and you don't keep a total on the Course record, then you'll have to loop through the registrant information for each record.

One easy way to do this is to use a hidden browse with the child records and a total (count) field. If you take this approach make sure that you set the child browse's ActiveInvisible property to True as the default behavior for browse objects is to become idle when hidden.

If there are a large number of child records or you frequently run reports or other processes where the number of registrants is required then it may be to your advantage to count the registrations whenever one is added or deleted and update a Registrants field with this value.

You may think at first glance that a BYTE value would suffice for Registrants, but for a particularly large lecture course there might be more than 255 students. Should that happen with a BYTE variable the numbers would wrap around and the displayed count would be incorrect. If there's *any* possibility that you'll overrun the variable's capacity then use the next larger variable. A SHORT has a maximum value of 32767, and while course overcrowding is often a problem, its not likely to get *that* bad.

Many-To-Many Redux

In my initial design of the course file I made the assumption that there would be only one instructor per course. This isn't necessarily the case. There may be multiple instructors, or an instructor and teaching/lab assistants who should also be associated with the course. A simple many-to-one relationship would be too restrictive. To handle this I'll need to create a linking file just as I did for the course registrations. As with the Registration file, this linking, at a minimum, would need only two IDs.

The only difference between this many-to-many and the Registration many-to-many is that this one is a bit more difficult to recognize. In the software requirements (if a formal document existed) you would probably come across the term “Course registration,” which might have led you to think about storing this information in its own file.

What if a suitable term hasn’t been defined for the link between instructors and courses? You can discover the need for such links by examining each file relationship carefully and asking yourself if the relationship will satisfy all the needs of the application.

By now you should be starting to see some other possible links between data. If you’re storing information about instructors, then perhaps there’s some possible duplication of the kind of information you keep about students. Perhaps a generalized contacts file would be better, with a `Type` field to differentiate between students and instructors (and perhaps staff as well). This way you can use the same approach for instructor addresses as for student addresses.

Telephone numbers (and email addresses and the like) are another potential area for generalization. Rather than having two or three telephone number fields in the names file you might want to go with a `Phones` or other kind of contacts file. Remember that you can assign field pictures at runtime, so you could even use a simple string field and use various pictures to format the string for phone numbers, email addresses, and so forth. You might want to keep this kind of configuration information in an INI file (not a great idea) or a single-record control file in your database. A control file is a better option as it allows for encryption, and it’s much less likely someone will inadvertently edit a database file as compared to an INI file. See “Working With Control Files,” p. 547 for more information.

Paranoid Anticipation

When I analyze or create a database design I’m constantly asking myself how the data will be used, given the current requirements and what I anticipate the future requirements will become. Few of us ever actually finish a software project. We may leave it or hand it off to someone else, but there’s always work to be done.

I find that the better I am at anticipating what the user will want to do with the program, the more robust my database design is likely to be.

Knowing your data is one important piece of the puzzle; knowing how to arrange the data is the other. I’ve described a few basic principles, but there’s a lot more to this subject. In the following chapters, Tom Ruby provides a Clarion perspective on the rules of database normalization.

Source code

See “Appendix A: Getting Support,” p. 601, for information on how to get the source accompanying this book.

- vln5novice4.zip

MANAGING COMPLEXITY, RULE 1: ELIMINATE REPEATING FIELDS

by Thomas Ruby

Programs are very complex things because users demand that they solve complex problems. You know to be wary when the customer says, “All it needs to do...” because there is always more to it than that. Your task is to analyze that complexity and produce a program that deals with it, leaving the user to think, “All I need to do is...”

This is a pretty tall order, and the last thing we need to do is add more complexity to the problem ourselves. Within our programs, dictionaries, and embeds, we want our work to be as simple and straight forward as we can make it so we can concentrate on the complexity of the problem, not on the complexity of our solution.

So first, here is a guideline. It’s not a hard and fast rule you must follow, but an example of a principle which can save you some hair.

Guideline 1

Don't take shortcuts. They might save you a few minutes now, but they'll cost you days later.

Guideline 1 suggests a strategy, or overall guide, which might be, "Make up a set of rules and stick to them." Consider a rule I follow when making dictionaries, "Make field names descriptive and don't abbreviate them." It is always a temptation to name a field something like Date. Hmmmm. There already is a field in this table called Date, and it is the date the record was entered. So I call the new field DateDue. But now I have an extra piece of information to keep in my head, and this information is about the solution to the problem, not about the problem, so it adds to the complexity of the task needlessly. This bit of information is, "The date entered is Date, while the due date is DateDue."

Keeping Guideline 1 in mind, I make the other date field more descriptive and change it to EntDate. Oops. This is an abbreviation. Why is that bad? Because it adds another piece of information to keep track of. I have to remember that I abbreviated Date Entered as EntDate. So I fix it by making it EnteredDate.

Now I feel much better, but before long, I have an annoying compile error because I typed DueDate instead of DateDue. Okay, so it isn't a big mistake, but how many times throughout the project do you want to get a compile error because of this mistake? So let's make an arbitrary rule, "The type of the field comes before the use of the field." It could easily be the other way around and it would work fine. In fact, my own convention for naming fields is the opposite. It doesn't matter - the purpose of the convention is just to remove unnecessary complexity.

This convention also makes it clear that the tax amount field is AmountTax and the tax percentage field is PercentTax. Already this convention has saved four scraps of unneeded information about the solution, and will probably save more later on.

Yes, I know Clarion has that handy field box, plus Data Modeller and the View Dictionary feature. I love them all and use them heavily, but which is better, looking up a field name or just knowing it instinctively? There's a lot these tools can help you with, but you don't need to be using them to solve problems you could have designed out at the start.

"Naw, let's not change it 'cause I'll have to re-edit all the embeds in that procedure." Again, how many times throughout the project do you want to make the mistake and get an annoying compile error? Change it now while the fix is easy to make. This brings up Guideline 2, which is more of an observation than a rule:

Guideline 2

It is a lot less painful to fix a painful mistake now than it will be later on.

As I explained to my son Ethan a few days ago, “It hurts to get a nasty splinter out, but it will hurt more to get it out next week. Do you really want to put up with it till then?” He opted to let me take the splinter out. So I’ll bite the bullet and edit the embeds in the procedure. My, the Embeditor is handy for things like this.

Let’s talk about abbreviations a bit. Would you make a field name `AmountFederalInsuranceContributionsAct`, or `AmountFICA`? FICA is indeed an abbreviation for Federal Insurance Contributions Act, but FICA is also the name of a fairly complex, not to mention wordy, concept. I’d consider FICA a name and call the field `AmountFICA`, but I wouldn’t abbreviate it as `FICAAmt`.

I could go on with examples all day, but I won’t. Take it from twentymumble years of sometimes frustrating experience: Stick to your rules!

To recap:

Guideline 1: Don’t take shortcuts. They might save you a few minutes now, but they’ll cost you days later.

Guideline 2: It is a lot less painful to fix a painful mistake now than it will be later on.

And now, without further ado, here is Rule Number 1:

Rule 1

Eliminate repeating fields.

People often paraphrase this rule by saying, “Arrays are bad,” but this tells only part of the story. I have actually seen people eliminate the bad array by defining their data like this:

```
EmployeeRate1  
EmployeeRate2  
EmployeeRate3  
EmpRate4      ! must have got tired of typing  
EpmRate5      ! didn't proof read  
EmpRt6
```

and so on.

And this changes fairly logical and easy to maintain code like this:

Introduction to Databases

```
LOOP I = 1 TO 6 BY 1
  YadaYada
  EmployeeRate[I] Yada
  Yada
  Yada (EmployeeRate[I])
  Yada
END
```

Into a mess like:

```
Yada Yada
EmployeeRate1 Yada Yada
Yada Yada Yada Yada
Yada EmployeeRate1 Yada Yada Yada
Yada Yada
EmployeeRate2 Yada Yada Yada
Yada Yada
Yada EmployeeRate2 Yada Yada Yada
Yada Yada Yada
(lots more nonsense)
Yada
Yada EmpRate4 Yada
Yada Yada Yada
```

So what's so bad about this? You can easily build it using cut and paste or an editor macro, right? Well, it turns out that the second Yada should be

```
Gobbledegeook EmpPlanA
Gobbledegeood EmployeePlaB
```

and so on for all umpteen bunches of code! Somebody has a long day of tedious editing to do. Do you think they're not going to goof somewhere? What if you code EmployeePlanC where you should have put EmployeePlaB? Just try figuring that bug out while the client is looking at screwed up calculations! Then you have to read through that 24 page embed, line by line, looking at these field names which all look alike but are subtly different. Are you sure you're not going to flub up while fixing them?

So getting rid of the array by replacing it with bunches of fields is not a good idea, but what's wrong with the array concept itself? The array causes a big limitation in the functionality of the program, and it adds a bit of complexity you can do without.

When you put an array in your data, you place a limitation on your program's usefulness. I call this "The Extension Cord Effect." Ever notice how often an extension cord is just a little too short? You could just always buy nine foot extension cords instead of six foot cords, but you know, the longer cords somehow just need to be a couple inches longer. Of course, you could always buy 100 ft. outdoor cords, but then you'd have all this orange cord laying around, tripping up the kids and tangling the vacuum cleaner.

The same thing happens to an array in your data design: You've allowed for eight employee deductions, but doggone it, you get a phone call interrupting your dinner because one of the customers needs 24, so you have to dig out the source code, make a change and update their

Managing Complexity, Rule 1: Eliminate Repeating Fields

database replacing the eight element array with a larger one. Then *everybody* gets space for 24, while most use only three, and it works fine until somebody needs 25. Why in the world would they need 25? I don't know, but if the user thinks they need 25, the program shouldn't tell them they can't have 25. And you did remember to make *all* occurrences of LOOP I = 1 TO 24 BY 1 into LOOP I = 1 TO 25 BY 1, didn't you? Everywhere? Are you sure? How sure are you?

A repeating field, be it an array or multiple fields, adds another extra bit of complexity to the project, and you certainly don't want to be making it more complex than it needs to be. You need a way to tell if the array element is used and skip over those that aren't. You need to put the code to do this *everywhere* you use the data out of the repeating fields, and you have to remember, yet again, how you coded the unused fields. You probably also need to move them around when the user deletes one because users always like the blanks at the bottom.

What to do? Put the repeating fields in a separate table, one field, or set of fields, per record, along with the employee's identifier so you know which records belong to which employee. For this example, the EmployeePlan table would contain three fields: the employee identifier, the rate field and the plan field. Now the code looks like this:

```
PLA:Employee = employee identifier
SET (EmployeePlanKey,EmployeePlanKey)
LOOP
    IF Access:EmployeePlan.Next() <> LEVEL:Benign OR |
        PLA:Employee <> employeeidentifier THEN BREAK END
        Yada Yada Yada Yada Yada
        PLA:EmployeeRate Yada Yada
    END
END
```

Oops, remember to change the second Yada to

```
Gobbledegook EMP:Plan
```

You don't have to worry about skipping unused records, because there aren't any unused records and the simple loop works for as many plans as are or aren't used. You also don't have to worry about how many records there are. The loop will just process all of them whether you have 200,000 entries, one entry or zero entries.

"But my users will hate another browse and form," you say. I don't blame them. There are better ways. The most obvious solution is Clarion's Edit-In-Place feature. You put a browse for the plan records on the employee form and range limit it to the employee identifier field. Then you add the update buttons and check **Use Edit-In-Place**. In the example application, I changed the class of the second column and added a single line of embed code to make it a drop list.

If your users, or your clients, are really stuck on the idea that there are a certain number of these repeating fields, you can fake them out on the form while still getting rid of the

repeating fields. Remember, user interface design is in large part psychology. The trick is to make a set of fields on the form and write these to the child table. In the `FRMFakeOut` procedure, I used an intermediate queue to allow the user to “roll” the list through the edit fields. It acts like 10 fields and the users don’t realize that you made your life easier, besides getting their program done faster and with less grief. Is the client going to freak out over the scroll up and down buttons? Hide them, but leave some hidden configuration parameter to unhide them with `PROP:Hide`, just in case.

Problems With Reports

Making the number of fields potentially unlimited can present reporting challenges. “But only four columns fit across the paper.” Now that’s a problem. Paper is just 8.5 inches wide in the US. The rest of you have it worse with 210 millimeter wide paper. With any report, you have to ask, “What does the user want to know by looking at this report,” and secondly, “What is the user going to do with this piece of paper after looking at it.”

When the computer was the great machine hidden away in a special room where only the white coated priests could enter, reports were of extreme importance. That was the only way information could get from the computer to the users. These days, we put computers everywhere. The high school kid at Wal*Mart who makes minimum wage works in front of a computer. We don’t mess with those thick books of 14 inch greenbar paper anymore because it is much more convenient to put the information on a screen than to look it up in that huge printout. Ask your users what they are going to do with that 248 page “Master Inventory Report.” I once had a client demand this very important report, but it was never used. You see, the Inventory browse was much more useful to his employees.

That aside, what do the users want the report for? I’m guessing the problem is a report that shows a line for each employee and a column for each something else. It is more important to find out what the user wants to know from the report than how they want the report formatted. Perhaps only the four latest records for each employee are significant, or four different totals of all the records for each employee would be more interesting. Perhaps the only reason they want it is because they always had it. Are they looking for something specific in the grid, like a number that is much greater than or less than the others? Or are they looking for an unusually high column total?

I failed to ask the “purpose” question once and ran two mainframe jobs, each producing a four inch thick stack of 14 inch greenbar paper, to satisfy a coworker’s request. Turns out all she wanted to know was how many records there were. I could have told her that immediately when she first asked for the report rather than four days and eight inches of paper later!

Say that after the analysis you're still faced with a report with only four columns for each employee. Do you need to restrict the data design for this report, or does the report itself exhibit the "extension cord problem?" That is, what happens when, and not if, somebody needs the seventh column? Rather than restricting your data design based on this report, you need to design the report to deal with the inevitable.

What Good Are Arrays In Tables?

"How can it be so bad if Clarion allows it?" Clarion allows you to put arrays in data tables because it has to. There are three reasons Clarion has to allow it.

The main reason is if they didn't, some numskull will complain loudly in a public place that Clarion is totally unusable because it lacks this important feature.

The most important reason is that we always have to deal with data from other programs. That's the whole reason we have drivers like dBase III. Is there an advantage in storing some table in a dBaseIII file rather than a TPS, Btrieve or SQL table? No. The reason we use the dBase driver is that some other program requires its input or output to be in dBaseIII files. That's why Clarion allows arrays in the data files, because the nit who wrote the other program you have to interface with didn't know any better.

I said there are three reasons, didn't I? The third reason is they have to support people who are too busy with problematic projects to read a chapter like this about how to design the problems out of a project.

Sometimes you have what looks like a repeating field but isn't. Consider a table which represents 1099 forms. There are a lot of numbers on the 1099, and if you're looking at an old program somebody else wrote, they might have stored these in an array. To me, fields named DollarsRents, DollarsRoyalties, DollarsOther and so on are more meaningful than AMT [1], AMT [2] and suchlike. These 11 fields are not repeating fields, so don't feel bad about putting them in the 1099 table.

To recap, here are the two Guidelines and Rule Number 1:

Guideline 1 Don't take shortcuts. They might save you a few minutes now, but they'll cost you days later.

Guideline 2 It is a lot less painful to fix a painful mistake now than it will be later on.

Rule 1 Eliminate repeating fields.

In the next chapter I'll introduce Rule Number 2 and some more guidelines.

Source code

See “Appendix A: Getting Support,” p. 601, for information on how to get the source accompanying this book.

- v2n8complexity1.zip

MANAGING COMPLEXITY, RULE 2: ELIMINATE REDUNDANT DATA

by Thomas Ruby

When you are designing the database, or the data dictionary, you are actually constructing your program's model of the universe. Since the entire universe is awfully big, you concentrate just on the tiny part of the universe that your program deals with. The more realistic your model of the universe is, the more able your program will be to do its job and deal with things you didn't think about at the start.

In the previous chapter, I introduced the idea of sticking to rules in order to reduce the complexity of your application and to speed up development. I presented Rule Number 1 and two guidelines. To summarize:

Guideline 1: Don't take shortcuts. They might save you a few minutes now, but they'll cost you days later.

Guideline 2: It is a lot less painful to fix a painful mistake now than it will be later on.

Rule 1: Eliminate repeating fields.

In Parts 2 through 5, I'll continue to introduce rules (five in all besides the guidelines), show you why they are valuable, and explain how to apply them in your Clarion projects.

You can make up rules for all sorts of things. I've seen rules about how big buttons should be and where they should be located. I've seen rules about how lists should be displayed and about what fonts to use. Many of these serve to enforce a standard look and feel to the application and to save the developer time in deciding how to arrange a window. One rule I like to use is:

Guideline 3

A list is resizable, a form is not.

Now, this isn't a hard and fast rule that says "never make a form procedure resizable," but it is a principle to guide me in deciding how or if to make a window resizable. You see, when a user stretches a list, they obviously want to see more entries. But if they stretch a form what do they want to see? Do they want to see more blank space? If the form is mostly a list, then it's obvious they want to see more entries in the list. The edit-in-place example from the previous chapter contains a form that might be resized.

Anyway, enough procrastinating. Here's Rule Number 2:

Rule 2

Eliminate redundant data

To explore this rule, I'll concoct a ridiculous example. Herb is writing a program for a garage. It is a really big garage with dozens of mechanics, and he would like to keep track of who is skilled at what sort of job. Herb might think of a table like this:

```
Mechanic Table
NameMechanic
DateBirth
SocialSecurity
RateHourly
Skill
```

But, of course, a mechanic probably has more than one skill. A common solution is to just list the mechanic multiple times in the table. But now Sue Pipebender is getting married. She has 241 entries in this table, and somebody has to go through all those entries and change her name. Oh well, what are secretaries for anyhow?

Herb might make his table like this:

Managing Complexity, Rule 2: Eliminate Redundant Data

```
Mechanic Table
NameMechanic
DateBirth
SocialSecurity
RateHourly
Skill1
Level1
Skill2
Level2
Skill3
Level3
Skill4
Level4
```

But Rule Number 1 tells us to avoid this sort of thing, so Herb builds two tables:

MechanicTable	SkillTable
NameMechanic	NameMechanic
DateBirth	Skill
SocialSecurity	Level
RateHourly	

So now each mechanic has a list of skills. This looks pretty good, except he wants to be able to search by skill, and somebody will likely enter, “Wheel Alignment” for Crusty McGinnes, “Alignment” for Joe Schmoe and “Align” for Guido. They could write the “approved” skill names on a piece of paper and tape it to the monitor but hey, this is the 21st century. Use a Post-It instead!

Herb, you can do better than that. Make a table of skills so you can put a drop list to pick the skill from. So now the tables look like this:

MechanicTable	MechanicSkillTable	SkillTable
NameMechanic	NameMechanic	Skill
SocialSecurity	Skill	
DateBirth	Level	
RateHourly		

There's nothing really wrong with this setup except when Herb changes a mechanic's name, he has to change the names of all the related records in the middle table. Using referential integrity, Clarion will take care of this for you, except...

Clarion always edits a record in memory and writes the record to the table when the user completes the form. If you put the list of skills on the mechanic form, when you go to change Sue Pipebender's name to Sue Arcwelder, her skills disappear! That's because there aren't any records that say Sue Arcwelder in the MechanicSkill table. They come back when you save the record and the record is actually changed causing the RI code to change

all the Sue Pipebender entries in the middle table to Sue Arcwelder. The users won't like this at all. What is Herb to do?

Look closely at Herb's `MechanicSkillTable`. It lists the mechanic's name and the name of the skill, but it doesn't really have to. The middle table's job is just to say "this mechanic has that skill," and it doesn't really care what the mechanic's name is. So here is where Rule Number 2 comes in. The purpose of Rule Number 2 is not really to reduce the storage required, but to make updates happen efficiently without surprises. I'll rearrange Herb's tables just a little.

MechanicTable	MechanicSkillTable	SkillTable
SysIDMechanic	SysIDMechanic	SysIDSkill
NameMechanic	SysIDSkill	NameSkill
SocialSecurity	Level	
DateBirth		
RateHourly		

I added a hidden field to the mechanic table and to the skill table which is used solely to link (or relate) them to the `MechanicSkillTable`. Notice I said it's a hidden field. I never display this field! Why? The user doesn't care what the `SysIDMechanic` is, so why show it to her? Also, these hidden fields don't mean anything so they never have to be changed. It might be tempting to use the mechanic's social security number for this linking field, but social security numbers sometimes change, particularly when they were mistyped in the first place. The Social Security Number might be an important key field for looking up a mechanic's record, but don't use it as your primary key.

Sometimes, people suggest making the Social Security Number unique will fix the problem, but what if you're entering a new employee record and a previously entered record has an incorrectly entered number and it's causing the new record to be rejected? Hey, I saw it happen, but it still didn't convince the DP department to use a hidden key field.

Sometimes this hidden field is called a "surrogate key."

A field which is meaningless to the user never has to change. I'll express this in Guideline 4:

Guideline 4

Link your tables by a hidden field that is completely meaningless outside the system.

So what does this gain? For one thing, updating the mechanic's name is now instantaneous. No other records have to be updated. Sure, Herb could buy a power server to hold these tables and run a fast data engine on it, but can even a power server update a bunch of records faster than it can update no records?

I went ahead and used a hidden linking field for the skill. Why? For the same reason. Supposing somebody mistyped "Water Pump." To correct it, they just have to look up "Water Pmup" in the skill browse and fix it. Immediately, everyone who was skilled in "Water Pmup" becomes skilled in "Water Pump." By the way, a form on this skill browse listing all the mechanics who have a skill would be a really convenient place for the users to ask, "Who can balance wheels?" When the garage wants to go to skill codes, all they have to do is edit the skill names, and instantly, all the mechanics that know how to "Flush Cooling System," have skill code 2432. I don't know why the garage would want to do this. Perhaps a skill code sounds more computerish.

"But I want to show the name of the skill on the browse, not some stupid SysID." Indeed! I don't want to show the SysID on the browse either, in fact, I never want to show the SysID. When you build your tables in the dictionary editor or whatever your favorite tool is, specify a key for each of your linking fields. Then specify a relation between the tables.

Always put your relation in the dictionary. "But there's business reasons not to relate these two tables." Nonsense. The tables and relationships are describing the world to the program. These business reasons cited are actually business rules that limit how you do things and they are reflected in the program code, or in the procedures and triggers of your SQL database.

Here is the guideline I use to make keys:

Guideline 5

Use keys to help the application identify records it is interested in.

There are two places where Guideline 5 applies. First, you make your linking fields, which I usually call SysIDs, into keys. This lets the application quickly find all the MechanicSkill records that apply to a mechanic, and the Skill record that applies to a MechanicSkill record. Second, you make a key (or an index if you're dealing with SQL) to help a browse identify "the next eight records to show."

Now that you have a relationship between the tables, it is trivially easy to make the browse show the skill name instead of the skill SysID. You just add the Skill table under the MechanicSkill table and pick the field that contains the skill name. The relationship tells the templates how to connect the Skill with the MechanicSkill. The same thing applies to reports. If you want to make a report showing all the skills each mechanic has,

Introduction to Databases

just put the three tables in the file schematic. Somehow, looping through the child records on a report just became pretty easy.

Sometimes, data is not redundant but looks like it is. Think about a sales and inventory system. There might be three tables that look like this:

Customer	Sold	Inventory
SysIDCustomer	SysIDCustomer	SysIDInventory
NameCustomer	SysIDIInventory	Description
	DateSold	Price
	Price	

Obviously, the `Sold` table tracks who bought what and when, but what is that price field doing there? Wouldn't it be better to skip the price field in the `Sold` table and look instead at the price field in the `Inventory` table? No, actually they are two different price fields sloppily named the same thing. The tables should look like this:

Customer	Sold	Inventory
SysIDCustomer	SysIDCustomer	SysIDInventory
NameCustomer	SysIDIInventory	Description
	DateSold	Price
	PriceSold	

You see, the `Inventory` record shows what the store is asking for the item now, but the `Sold` record shows what the store was asking for the item when it was sold. Supposing the storekeeper wanted to keep more information about the price history? You might build him four tables like this:

Customer	Sold	ItemPrice	Inventory
SysIDCustomer	SysIDCustomer	SysIDIItemPrice	SysIDInventory
NameCustomer	SysIDIItemPrice	SysIDIInventory	Description
	DateSold	DateStarting	
	Quantity	DateEnding	
		Price	

Now `ItemPrice` contains a history of the price of each item, with starting and ending date. I didn't put the Price Sold in the `Sold` table because that can be found in the `ItemPrice` record. This would work unless the store negotiates prices with the customer, in which case `PriceSold` would have to be added back to the `Sold` record.

Managing Complexity, Rule 2: Eliminate Redundant Data

To further illustrate, think about a company that ships things to its customers. Each customer has an address, so you might think you could skip putting the address in the shipment record like this:

Customer	Shipment	ShipmentDetail
SysIDCustomer	SysIDShipment	SysIDShipment
NameCustomer	SysIDCustomer	SysIDInventory
CompanyCustomer	DateShipped	Quantity
Address		
City		
State		
Zip		

Inventory
SysIDInventory
Description

Be careful. This can only record shipments shipped to the customer's address. Perhaps the customer wants the shipment shipped to his client or his mother. Maybe the client moved. You probably want the data to look more like this:

Customer	Shipment	ShipmentDetail
SysIDCustomer	SysIDShipment	SysIDShipment
NameCustomer	SysIDCustomer	SysIDInventory
CompanyCustomer	DateShipped	Quantity
Address	AddressShipped	
City	CityShipped	
State	StateShipped	
Zip	ZipShipped	

Inventory
SysIDInventory
Description

Usually, the AddressShipped field is just copied from the Address field, but the customer might want it shipped somewhere else. Do you want your users telling their customers they can't do that just because the program can't accommodate it? This might seem like a lot of redundant data, but actually it isn't. Consider a customer that moves: you change the address in the customer record, but as you deal with them, you have a record that the TV they bought last year was shipped to 11535 IL HWY 9, while the monitor they just bought was shipped to 19215 N 100th Rd. This can be valuable information if a question comes up, and it's really cheap to store this extra address. Remember, remove redundant data to make updating it fast and reliable. Keep historic data when necessary.

Introduction to Databases

You'll want to automate the process so the computer is printing the shipping label from the database. If a question comes up as to where the shipment went, you have the address it was shipped to right here.

Just be sure to automatically get all the information from the database to the shipping label! I'll illustrate this with a true story. Most of the names have been omitted to protect... you know the drill.

A client wanted to ship me a piece of equipment to test a new program on, so he went to the manufacturer's online ordering web site and entered the order. The order never came. The manufacturer's customer service department had to go searching all over. My client had entered the order as:

Name: Tom Ruby
Company: His Company Name
Address: 19215 N 100th Rd
City: Industry
State: IL

But the clerk looking at the screen, typed the order into their fulfillment system like this:

Company: His Company Name
Address: 129215 N 100th Rd
City: Industry
State: IL

Now, Industry is a small place, and the UPS driver knows me pretty well. If the equipment had been shipped to Tom Ruby at the incorrectly entered address, the UPS driver would have figured it out. Or, if it had been shipped to His Company Name at the right address, the UPS driver would have figured it out. But with neither my name, nor my address, the equipment wound up at the receiving department of a coal mine for several days while they scratched their heads wondering who would have ordered something from His Company Name.

The shipping department had confirmation that the item had been delivered to 129215 N 100th Rd and thought all was fine. The order entry system had recorded that the item was shipped to 19215 N 100th Rd, and all was fine. Since everything was fine, the manufacturer was at a loss to figure out why I didn't have it. Had the shipping address been automatically copied to the shipping label, there wouldn't have been a problem.

Another example: you would think that the big credit-card issuing banks with their huge DP departments would get their data design right, but alas. Have you ever lost a credit card? They have to close the account and open another one just like it, and you'll get two bills, one showing charges and payments made on the old account, and the other showing charges and payments made on the new account. Do you see what the problem is? (Yes,

Managing Complexity, Rule 2: Eliminate Redundant Data

you in the back.) That's right: the card number is the primary key and all the transaction records use the card number to connect back to the account record. If they used a surrogate key to relate everything back to the account record, all they would have to do is set the card number field of the account record to the new card number. Probably they want to keep the card numbers in a separate table so they can record that this card belonged to that account but was stolen.

So to recap, here are the two new Guidelines and Rule 2:

Guideline 3: A list is resizable, a form is not.

Guideline 4: Link your tables by a hidden field that is completely meaningless outside the system.

Guideline 5: Use keys to help the application identify records it is interested in.

Rule 2: Eliminate Redundant Data

In the next chapter I'll discuss Rule 3.

MANAGING COMPLEXITY, RULE 3: ELIMINATE COLUMNS THAT DON'T BELONG

by Thomas Ruby

In this series, I've introduced the idea of using rules to reduce the complexity of your work. Software complexity is increasing because the users and clients expect your software to solve increasingly complex problems for them, and if you don't do it, your loyal customers will beat a path to your competitor's door. Face it, the days when the users were delighted with a 2,000 line program with two tables, a browse and a report are far behind.

How do you handle this complexity? The first thing to do is avoid making it worse. Clarion is a powerful tool for solving problems, but let's apply its power to solving the user's problems, not to solving problems we add to them.

So I'll recap Rules 1 and 2, along with the guidelines, and get on with Rule 3.

Guideline 1: Don't take shortcuts. They might save you a few minutes now, but they'll cost you days later.

Guideline 2: It is a lot less painful to fix a painful mistake now than it will be later on.

Rule 1: Eliminate repeating fields.

Guideline 3: A list is resizable, a form is not.

Guideline 4: Link your tables by a hidden field that is completely meaningless outside the system.

Guideline 5: Use keys to help the application identify records it is interested in.

Rule 2: Eliminate redundant data

If you stick to Rules 1 and 2 when you lay out your data, you will find a lot of the complicated code you're used to writing disappears. There's something else that will disappear; bugs. I mean the type of bug where the users complain, "I changed this here and something went wrong over there..." You like the idea of less headache, fewer bugs and less tangled code? Then I'll go on with Rule Number 3:

Rule 3

Eliminate columns that don't belong

To explain Rule Number 3, I'll talk a little about keys. In Clarion, we have the idea of a key and an index rather mixed up, because you usually want an index on a key. A key is a column (or field) in a table (or file) which is used to identify rows (or records), while an index is a data structure used to impose an order on the rows without having to actually sort them to make finding records faster. You usually want an index on a key, but you don't have to have one. A key is a logical construct, an index a physical construct.

Now, when I talk about "The Key," or "The Primary Key," what I mean is a column or columns that uniquely identify each row in the table. Even more than that, I mean a column or a value that represents the thing the row of the table records. The other columns of the table describe the thing the KEY represents. Take a look at an example of a Student table:

StudentID	Name	Gender	Teacher	Grade	Section
102	Caleb	M	Wilson	2	1
103	Cameron	M	Wilson	2	1
104	Anthony	M	Wilson	2	1
105	Angel	F	Wilson	2	1
106	Sarah	F	Wilson	2	1
107	Ethan	M	Harn	1	1
108	Mandy	F	Harn	1	1

Managing Complexity, Rule 3: Eliminate Columns That Don't Belong

109	Waylon	M	Wilson	2	1
110	Emily	F	Harn	1	1
111	Seth	M	Harn	1	1
112	Ashley	F	Hard	1	1
112	Drew	M	Wilson	2	1

Obviously, the `StudentID` is the primary key in the `Student` table. Remember, the primary key *represents* the student. The other columns *describe* the student. The table also contains the student's name, gender, teacher, grade, and section. The section must be which class of the grade they go to. But wait! Three of these columns describe the class, not the student, so they don't belong here. They belong in a class table. So I'll make two tables:

StudentTable
StudentID
Name
Gender
ClassID

ClassTable
ClassID
Teacher
Grade
Section

Why is this better? For one thing, moving a student from one class to another is a matter of updating one field, not three. Also, you don't have to worry about things getting out of sync, or the users asking, "If Drew is in second grade, why does the database show his teacher as Mrs. Harn, the first grade teacher? Uh oh, Mrs. Bricker will be teaching second grade. With a single table, some user would have had to find and update seven records, or you would have to write a change teacher process. With a separate class table, only one record needs to be updated.

To clarify this, here's Guideline 5:

Guideline 5

The primary key represents the “thing.” The rest of the columns describe the thing the primary key represents.

You probably have more information you want to store about the teacher. Perhaps the teacher's phone number, address, or state certificate number. As long as one teacher only teaches one class, you could probably put these in the `Class` table, which would change it into a teacher table. If you're talking about a bigger school or older kids where a teacher

Introduction to Databases

might teach several classes, you would want to make separate teacher tables and class tables. These tables might look like this:

StudentTable	ClassTable	TeacherTable
StudentID	ClassID	TeacherID
StudentName	TeacherID	TeacherName
StudentAddress	RoomNumber	CertificateNumber
GradeLevel	Period	TeacherPhoneNumber

So how do you record which class the student is in? It is pretty likely that a student is in more than one class. A class also has more than one student. This is the dreaded Many-To-Many problem. You can't put a list of classes in the Student table because that would violate Rule Number 1, and you can't put a list of students in the class table because that would also violate Rule Number 1. Fortunately, the conundrum is easily solved.

"You don't mean another..." Yes, I mean another table.

AttendsTable
AttendsID
StudentID
ClassID
Grade

Now you can record that Sasha is taking Biology, Algebra 1, Wood Shop, Art, English and Civics. You also have a place to record what grade she got in each of these classes. Notice that I gave the AttendsTable an AttendsID. This isn't actually needed, because the table's primary key could be StudentID and ClassID. I gave it a single field primary key for two reasons. First, force of habit. Second, if you discover later on you need a list of something from AttendsTable, maybe an attendance or assignment history, you have a single linking field to use, which is easier to use in a range limit than something like HST:StudentID = ATT:StudentID AND HST:ClassID = ATT:ClassID. When you're building an application, you never know what is going to happen to the requirements for the application, so while you're at it, you might as well simplify the work you'll have to do later on. This is so important that I'll make it a guideline:

Guideline 6

You never know what will happen to the specification later on, so you might as well simplify your future work while you're at it.

Managing Complexity, Rule 3: Eliminate Columns That Don't Belong

You might wonder how you're going to show a student's schedule, or a teacher's class list, with all the fields scattered all over the place. Whenever you make a report or even a browse, the templates make a view structure for you, and a view serves to temporarily present you with an un-normalized picture of your data, much like the original student table.

So, to recap. Here are the first three rules of data normalization:

Rule 1: Eliminate repeating fields.

Rule 2: Eliminate redundant data

Rule 3: Eliminate Columns that don't belong

And here are the guidelines:

Guideline 1: Don't take shortcuts. They might save you a few minutes now, but they'll cost you days later.

Guideline 2: It is a lot less painful to fix a painful mistake now than it will be later on.

Guideline 3: Link your tables by a hidden field that is completely meaningless outside the system.

Guideline 4: Use keys to help the application identify records it is interested in.

Guideline 5: The primary key represents the "thing." The rest of the record describes the thing the primary key represents.

Guideline 6: You never know what will happen to the specification later on, so you might as well simplify your future work while you're at it.

For most situations, three rules, or "Third Normal Form" is considered "normal enough," but there are big advantages to understanding Rules 4 and 5. Next, Rule 4.

MANAGING COMPLEXITY, RULE 4: ISOLATE INDEPENDENT MULTIPLE RELATIONSHIPS

by Thomas Ruby

"Rules! Rules! Rules! What ever do we need all these rules for? Creativity must not be restricted!"

Bugs! Delays! Rewrites! Upset Clients! What do those do for your creativity?

By knowing your science, you'll be released to pursue your art. I've given you the three rules of normalization plus six guidelines to help you manage the complexity of your application. This way you can keep your mind on the user's problem you're trying to solve rather than keeping busy with problems you've created.

Rule 1: Eliminate repeating fields.

Rule 2: Eliminate redundant data

Rule 3: Eliminate Columns that don't belong

Guideline 1: Don't take shortcuts. They might save you a few minutes now, but they'll cost you days later.

Guideline 2: It is a lot less painful to fix a painful mistake now than it will be later on.

Guideline 3: A list is resizable, a form is not.

Guideline 4: Link your tables by a hidden field that is completely meaningless outside the system.

Guideline 5: Use keys to help the application identify records it is interested in.

Guideline 6: The primary key represents the “thing.” The rest of the record describes the thing the primary key represents.

Guideline 7: You never know what will happen to the specification later on, so you might as well simplify your future work while you’re at it.

At this point, you’re probably thinking either, “Give me more!” or “Oh no, not again.” To tell you the truth, I’m procrastinating myself, and Dave is going to be wondering if I’m ever going to finish the five chapters I promised him. Take charge! Grab the bull by the horns! Carpe Diem! So, without further ado:

Rule 4

Isolate independent multiple relationships.

So what does this gobbledegook mean? It’s not as bad as it sounds; in fact, it’s almost a no-brainer. It means if you have more than one multiple relationship, and they don’t have anything to do with each other, put them in separate tables. The best way to understand this, or perhaps the only way I can figure out to explain it, is with an example. Think about the garage example of part two with these tables:

MechanicTable	MechanicSkillTable	SkillTable
SysIDMechanic	SysIDMechanic	SysIDSkill
NameMechanic	SysIDSkill	NameSkill
SocialSecurity	Level	
DateBirth		
RateHourly		

Managing Complexity, Rule 4: Isolate Independent Multiple Relationships

If Herb wanted to keep track of tools owned by each mechanic, rule four tells him not to do it in the `MechanicSkillTable`, but to make a new `ToolTable`, which might look like:

```
ToolTable
SysIDMechanic
ToolDescription
```

Now, this might seem like a no-brainer, but sometimes you can't convince the client that the two different "things" really have nothing to do with each other and they demand to see them in one list. It is pretty obvious that tools and skills don't look anything alike and so should be in separate tables, but suppose the issue is "Tools" and "Uniforms," both of which the mechanic owns? Perhaps Herb has been lead astray by the garage's vocabulary where when they refer to "Tools and Uniforms," they really mean "Property," including tools and uniforms.

Rule 4, in this case, doesn't tell me how to organize the data like Rules 1 through 3 do. Instead, Rule 4 tells me that Herb didn't quite understand the requirements for the program. Herb knocks himself in the noggin and goes off to make the changes, creating a separate `ToolTable`. And I feel another guideline coming on:

Guideline 8

If it's not making sense and it looks like you really have to break a rule, you might not understand the problem fully.

In case you haven't figured it out yet, the five rules are the Five Normal Forms. The whole idea of "Data Normalization" is to construct a model of the world out of tables. The better you construct this model, the better your program will be able to deal with the world and the better your program will be able to adapt to changes in its environment.

Next time, Rule Number 5 and the dreaded one-to-one relationship.

MANAGING COMPLEXITY, RULE 5: ISOLATE SEMANTICALLY RELATED MULTIPLE RELATIONSHIPS

by Thomas Ruby

In the previous four chapters I explained four of the five rules of Data Normalization. I also gave eight related guidelines, and showed how to apply them in your Clarion Programs.

Let me start this chapter by shocking you with Guideline 9:

Guideline 9

Consider one-to-one relationships harmful.

How does that go again? “Consider one-to-one relationships harmful.” A one-to-one relationship means that every row in one table is paired with exactly one row in another table. If the relationship is truly one-to-one, then why do you have two tables? They should be combined into one.

Having said that and having tried to convince you that there is no place for a one-to-one relationship, I'll tell you why you might want to use a one-to-one relationship. Clarion, like many other data management languages, edits a record in memory and saves it to the table when the user presses the **Ok** button. As a nod toward multi-user activity, Clarion checks to see that nobody else has updated that record before saving it and gives you the message, "**This record was changed by another station. Those changes will now be displayed. Use the Ditto Button or Ctrl+Z**" to recall your changes."

In a lot of multi-user systems this is a quite adequate solution to the "last one who saves, wins" problem because it is unlikely two users will be updating the same record at the same time. But suppose you have an online inventory system with dozens of cash registers ringing up tickets and "taking" things out of inventory, and a receiving department with several employees unloading trucks and putting items in inventory. The `QuantityOnHand` value is a very useful figure for store management, so operators would probably like to see it on the Item form. If you put it in the item record, the chance is pretty good that one of the cash registers or receiving clerks will change the number before the store manager, who is just wanting to correct the description, can hit the **Ok** button. And the chances are again pretty good that the record will change again before the manager can make his change again, even using the history feature, and hit the **Ok** button. It could lead to frustrated users.

A frequent solution is not to update the `QuantityOnHand` during the day and update it during an end-of-day operation. But this is an online system, and the user doesn't want to know how many the store had yesterday, but how many it has now.

Do you throw up your hands and announce that Clarion is totally useless for intensive multi-user applications? Some do. I don't. Remember Guideline 8: "If it's not making sense and it looks like you really have to break a rule, you might not understand the problem fully." With this in mind, I realize that I just haven't understood the problem fully. You see, there is a field, `QuantityOnHand`, which is *not* updated on a form, but by some other activity. The other activities happen to be cashiers selling items and receiving clerks unloading trucks. In a sudden burst of insight, probably accompanied by a burned out light bulb, Guideline 10 pops into mind:

Guideline 10

Separate automatically updated columns from manually updated columns.

Considering the new Guideline 10, there are likely several columns that are "automatically" updated, and I realize why Guideline 9 doesn't say "*never* use one-to-one relationships." So I'll make a separate table and update it with a sequence like:

```
!Ainventory being the automagically updated part
```

Managing Complexity, Rule 5: Isolate Semantically Related Multiple

```
LOGOUT(10,Ainventory)
Access:Ainventory.Fetch()
AINV:QuantityOnHand -= QuantitySold
Access:Ainventory.Update()
COMMIT
```

Or in an SQL environment:

```
Ainventory{PROP:SQL} = 'UPDATE Ainventory WHERE
    ItemSysID = xxx SET QuantityOnHand =
    QuantityOnHand - QuantitySold'
```

I could be extremely clever and put a timer on the form to retrieve the automatically updated figure and redisplay it now and then so the user can “watch” his inventory go up and down. Now the user knows exactly how many packages of “Screaming Yellow Zonkers” she has on hand, except for the one a customer is pushing around in a shopping cart.

The whole point of Guidelines 9 and 10 is to reserve one-To-one relationships to situations where there’s a column that is being updated by something other than the form, and to prevent me from having to update two records on the same form.

Rule Number 5, or Fifth Normal Form, is also related to updating the database. In fact all the rules so far are related to updating the database.

Rule 5

Isolate Semantically Related Multiple Relationships.

It’s not my fault! Somebody at a university made this rule up. To illustrate Rule Number 5, imagine you’re interested in tracking distributors and manufacturers of different parts. Since the manufacturer doesn’t want to deal in the quantities you buy, you have to buy from distributors. Each distributor sells the parts from several manufacturers, and the same part may be available from different manufacturers. In a word, it’s a mess.

You might think of four tables that look somewhat like this:

Distributor	Part
DistributorSysID	PartSysID
DistributorName	PartName
DistributorAddress	PartDescription
And all that rot	

Introduction to Databases

Manufacturer	DistributorPartManufacturer
ManufacturerSysID	DistributorSysID
ManufacturerName	PartSysID
ManufacturerAddress	ManufacturerSysID
Bla bla bla	

The fourth table, `DistributorPartManufacturer` tells you which distributors sell which parts from which manufacturer. Rule Number 5 (or Fifth Normal Form) dictates against this since there are two different relationships, even though they're related. Instead, you should separate this table into two like this:

DistributorManufacturer	PartManufacturer
DistributorSysID	PartSysID
ManufacturerSysID	ManufacturerSysID

So what does this gain you besides two more tables? Like the other rules, simplicity in updating. Suppose Acme Widgits starts making three of those handy bolts with the threads offset from the shafts for when the holes don't line up. Since Acme is handled by four distributors, with the un-normalized table, you need to add 12 records to the big cross reference table to show that these parts now come from the four distributors where you get Acme parts. You also have a fairly complex piece of logic to figure out from the `DistributorPartManufacturer` table which distributors sell Acme so you can add these records to the table. If nobody else makes these parts, you have to add the three new part records.

With the normalized table, you need to add fewer records, and the code to add these records is simpler. You only need to add three records to the `PartManufacturer` table to show that Acme now makes the thread offset bolts. If you have lots of update activity, Rule Number 5 can improve your efficiency quite a bit, not to mention reducing complex logic.

To wrap this up, I'll enumerate the Five Rules and Ten Guidelines:

Rule 1: Eliminate repeating fields.

Rule 2: Eliminate redundant data

Rule 3: Eliminate Columns that don't belong

Rule 4: Isolate independent multiple relationships.

Rule 5: Isolate Semantically Related Multiple Relationships.

Guideline 1: Don't take shortcuts. They might save you a few minutes now, but they'll cost you days later.

Guideline 2: It is a lot less painful to fix a painful mistake now than it will be later on.

Guideline 3: A list is resizable, a form is not.

Guideline 4: Link your tables by a hidden field that is completely meaningless outside the system.

Guideline 5: Use keys to help the application identify records it is interested in.

Guideline 6: The primary key represents the “thing.” The rest of the record describes the thing the primary key represents.

Guideline 7: You never know what will happen to the specification later on, so you might as well simplify your future work while you’re at it.

Guideline 8: If it’s not making sense and it looks like you really have to break a rule, you might not understand the problem fully.

Guideline 9: Consider one-to-one relationships harmful.

Guideline 10: Separate automatically updated columns from manually updated columns.

One more point. You may have heard it said that Third Normal Form is usually considered “normal enough.” Most say it’s because the situations in Fourth and Fifth Normal Forms rarely crop up. Actually, these situations appear all over the place, but usually by the time you’ve thought through the first three normal forms, you’ve already satisfied the Fourth and Fifth forms.

Finally!

Do you have to follow these five rules of data normalization? No, you don’t. I have a hard time understanding why you wouldn’t want to make your work easier. Indeed most software is developed the hard way anyhow so just go along with the crowd and give yourself ulcers.

Okay, I *am* being sarcastic. You don’t *have* to follow the rules of data normalization, but I know that somehow, every time I’ve broken them, I’ve wound up wishing I hadn’t.

DISPLAYING NORMALIZED DATA

by Steven Parker

One of the basic concepts of normalization is that every record in a file, or row in a SQL table, should be uniquely identified. The effort necessary to ensure unique identification has practical and immediate benefits in some cases: browses on SQL tables without row level unique identification can behave bizarrely in Clarion (for example, rows may show up multiple times, or errors can be thrown on an attempted update).

Since I am still developing against flat files, unique identifiers preventing spurious duplicate browse lines doesn't motivate me a whole lot. To me, the real utility of unique identification lies in the facilitation of relations and lookups. This is true for both flat files and SQL.

I want to address two aspects of unique IDs: the generic motivation for using them, and the difficulties they can present when displayed to the end user.

Typically, Clarion developers call this unique identifier `SysID` or something similar, and create it by template-controlled autonumbering.

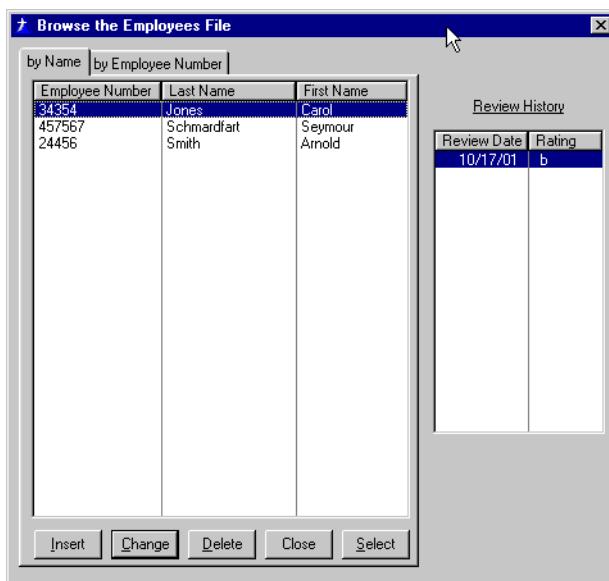
How does autonumbering work?

Imagine a human resources application. One of the files (or tables) will almost certainly contain basic demographics on employees (possibly, this file will be called EMPLOYEES). Another might contain employee reviews (I suppose it *might* be called REVIEWS).

If each employee has a SysID assigned (i.e. a primary key) then, if that number is also contained in REVIEWS records (i.e. a foreign key), it is easy to create code to retrieve all the reviews for a given employee (and only that employee) with a few mouse clicks.

The point is not that the same datum must be in both files/tables in order to link records; that should be fairly obvious. The point is that this manner of linking makes it *easy* to do things in our apps that make them appealing to users. For example, using this kind of link, it is (even) possible to “pair” the parent record with the child records in a single browse, as shown in Figure 1.

Figure 1: Review table “follows” employee table



In the dictionary for the demo application, you will notice that I have both a SysID and an EmployeeNumber field. This is because an employee number is not sufficiently reliable as a linking datum. An employee number can change. Of course, *I* would never allow that without adequate referential integrity constraints; I would configure things so that changes would always cascade. *Always*.

But, suppose *you* write an application against these data files

Now suppose a DBA changes a table layout. Suppose there is a data corruption. Suppose a data entry clerk, in another application that does not check for duplicates, accidentally reuses an existing employee number.

So much data, so many sources of mashed links.

What I need is a datum that uniquely identifies a master (parent) record, is included in each related record (RI constraints), and *is not changeable* by a user or, even a DBA (i.e., is under no one's control but mine).

An autonumbered key fits this bill to a "T." Indeed, many developers use such a field and never display it to the end user at all.

And that is why there is a `SysID` and an `EmployeeNumber`.

Okay, unless you're completely new to data-driven applications, this is not news and it is not rocket science. Neither is it news that if data is used in many places, it should physically exist in only one place. From the one place where the data physically exists, it is looked up everywhere else it is needed or is useful. And, in all such cases, the data is identified by a `SysID`-type field. For example, an employee's name, address and phone number are in `EMPLOYEES` but anywhere else I need to see or print that data, I look it up from the central location using `EMP:SysID`.

Now, consider the simple Employee-Review structure described above. But, consider it from the point of view of the `REVIEWS` file (entry of performance reviews will not always be through the employee master file, so access directly to `REVIEWS` is necessary).

A "Wizard-ed" browse would first appear empty, expecting an employee to be selected, only then completing (as in Figure 2).

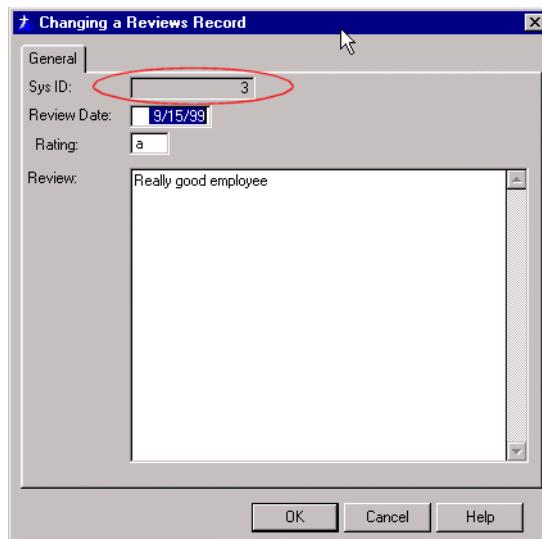
Figure 2: "Wizarded" child browse after parent selection



Introduction to Databases

Not very user friendly. The typical update form (Figure 3) is no better:

Figure 3: Child update form



It is no better because, while I may or may not know Seymour, the employee whose review is shown in Figure 3, I have no idea what a “3” is (the data files are included in the demo app so you can see that “3” is indeed Seymour Schmardfart¹).

It is easy enough to create a browse of all employee reviews (i.e., without using the range limits that the wizard defaults to) and, using the relationship between EMPLOYEES and REVIEWS, display the employee name. Because related data is projected into the browse’s view, the related file’s fields can be used just as if they were in the primary file, including concatenating first and last name fields into a single name display. This makes a much friendlier browse (Figure 4):

1. Seymour Schmardfart, according to Jerome Singer, is the brunt of every weird example in Social Psychology. He has served me faithfully through the years in a role expanding beyond the narrow confines of mere inter-personal behavior studies

Displaying Normalized Data

Figure 4: Child browse with parent data

REV:RevSysID		
Name	Review Date	Rating
Jones, Carol	10/17/01	b
Schmardfart, Seymour	9/15/99	a
Schmardfart, Seymour	9/15/00	aa
Schmardfart, Seymour	9/15/01	aaa

I can even use this in the update form to display the employee name (Figure 5).

Figure 5: Name display added to SysID

The screenshot shows a Windows application window titled "Changing a Reviews Record". The window has a tabbed interface with the "General" tab selected. Inside the "General" tab, there are four input fields: "Sys ID:" with value "3", "Review Date:" with value "9/15/01", "Rating:" with value "aaa", and a large text area labeled "Review:" containing the text "REALLY ok.". At the bottom of the window are three buttons: "OK", "Cancel", and "Help".

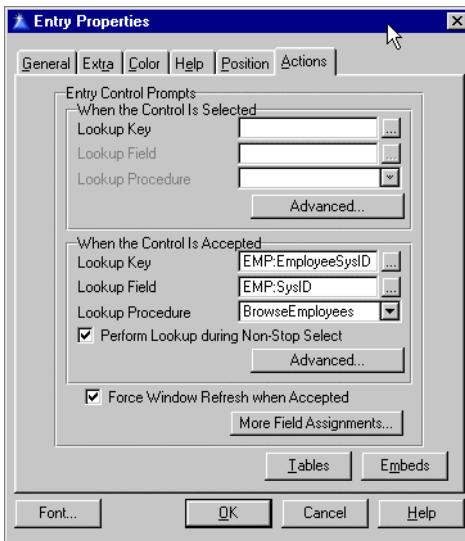
A digression

By the way, I've discovered a marvelous little trick for creating and displaying the name from the lookup into the EMPLOYEES file.

Previously, I would press the **Field Priming on Insert** button and prime FullName to '' which forces it to be blank when inserting a new record.

After looking up the employee (in **SysID, Accepted** - see Figure 6), I let the templates handle retrieval of the employee's name for me. If I want a full name display, I concatenate the data after the lookup.

Figure 6: Standard validation



To display the name when editing an existing record, I check whether the form is being called to edit with `ThisWindow.Request = ChangeRecord` and, if so, I manually do the lookup in `INIT`, after opening the files.

However, I have since found that the following code, after the files are open eliminates the need to prime-on-insert to ensure a blank FullName:

```
EMP:SysID = REV:SysID
Access:Employees.Fetch(EMP:EmployeeSysIDKey)
FullName = Clip(EMP:FirstName) & ' ' & EMP:LastName
```

Because the `FETCH` method clears the buffer when it cannot find the record, which will happen on an Insert because there is no value in `REV:SysID`, the two name fields are

blank and, therefore, the FullName will also be blank. But, if called to edit an existing record, the `FETCH` will succeed.

Back to my story

The fact of the matter is that the update form, shown in Figure 5, is pretty clunky looking. It is even clunkier to use.

In fact, it is unusable.

On its face, it looks like I am expecting the user to enter a `SysID`, not an employee name, not an employee number - either of which I might reasonably expect an end user to know or to have access to - but this utterly mysterious `SysID` thingee.

Furthermore, in the demo app, I've made the `SysID` field skip and read-only. How could any one ever enter the required datum? What if I'd followed common practice of not displaying it at all?

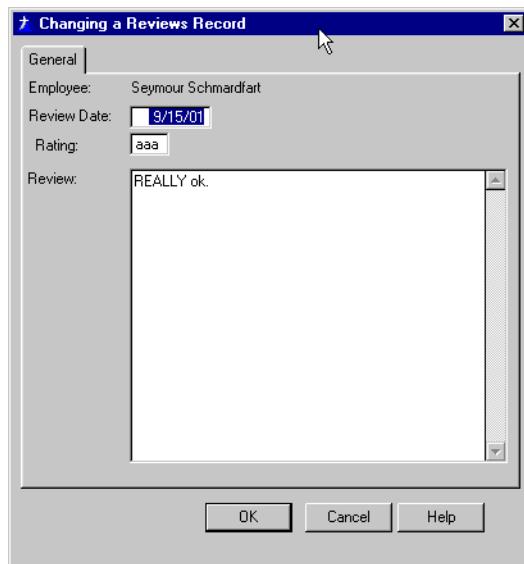
Even if I made it enterable, its virtue to me, as a designer, is that it is meaningless and this very virtue now plays against me when the user needs to complete it. So, I have to ask the user to enter a field that isn't there, or a field that can't be written to or a field that makes no sense to the user.

Right.

Introduction to Databases

What I want would look more like Figure 7:

Figure 7: Form without “SysID”



This is easily achieved by changing the prompt, hiding the `SysID` field and moving the name field to the left. But I still can't enter a new record.

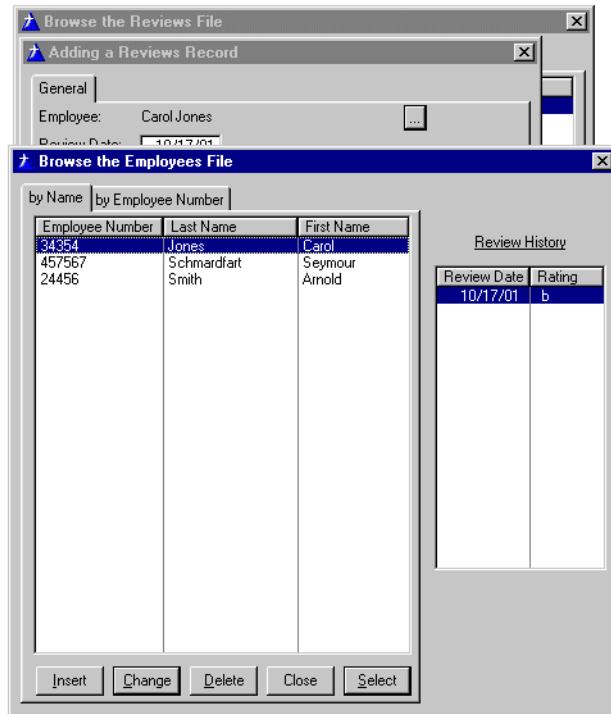
The conundrum is that I do not want the users touching (sometimes not even seeing) a datum that I *require* them to enter. And the field that the user can see and make sense of doesn't really exist anywhere. Do I sense a consistency problem here?

The `FieldLookupButton` control template, of course, resolves all the problems (and even makes the consistency issue moot). This template requires the label of a control with a lookup (as in Figure 6, above). It does not require that the field be visible. So, if I have a lookup on `REV:SysID`, even though I've hidden that control, the lookup button will still

Displaying Normalized Data

work. Figure 8 shows what happens when pressing the lookup button when the SysID field is hidden.

Figure 8: Pressing “...” with SysID field hidden

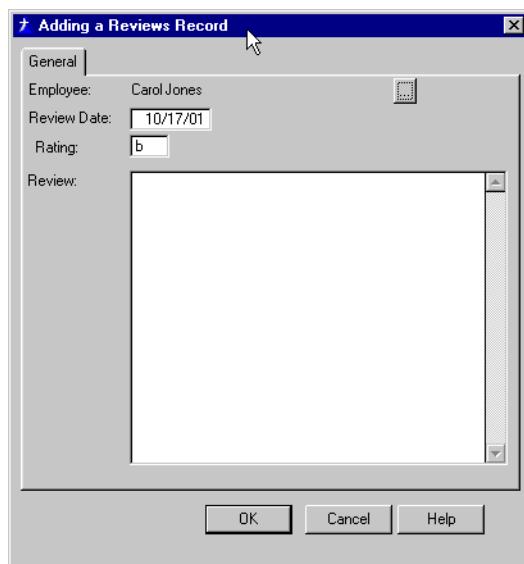


Voila! I only need to embed this code:

```
FullName = Clip(EMP:FirstName) & ' ' & EMP:LastName
```

in the button's **Accepted** embed to ensure that the form displays as I wish:

Figure 9: A more user-friendly form



It is also possible to trick the users by having them look up the last name (there is a procedure in the demo app constructed this way). In this case, the lookup button would refer to the `EMP:LastName` field and the "Additional Assignments" button would assign:

```
REV:SysID = EMP:SysID
```

Either method will work. It's a matter of taste.

Summary

Normalization is a two edged sword. It makes relations, relational views and lookups much easier. It allows me to provide a lot of eye candy with a few mouse clicks. It allows me to give a hierarchical file structure a flat (or flatter) appearance to the user (by simply adding update buttons to a child browse).

On the other hand, normalization makes entering and displaying records in child files much more of a challenge for the typical user. With a judicious use of the tools at your disposal, however, you can lead the user down the primrose path to do what the file structure requires without creating any great intellectual challenge... at least, not more than is absolutely necessary.

Source code

See “Appendix A: Getting Support,” p. 601, for information on how to get the source accompanying this book.

- v3n9normal.zip

DISPLAYING MANY-TO-MANY RELATIONSHIPS

by Thomas Ruby

If your users don't understand your program, or don't like how you've presented their data, they're not going to be as productive as they could be (and they may not be your users much longer!) The more complex the data, the more careful you have to be about how you present it to the user.

One area that causes developers problems is many-to-many relationships. The concept of many-to-many has been previously covered in this book ("Handling Many-To-Many Relationships," p. 9); in this chapter I'll cover three of the ways you can present this kind of data to your users. I call these ways "Form and Lookup," "Check List," and "Selection Pool."

Many-To-Many Basics

A many-to-many relationship describes a link between two entities, one on the left, the other on the right, where each item on the right may be matched to zero or more items on

Introduction to Databases

the left, while each item on the left may be associated with zero or more items on the right. Many-to-many relationships occur all the time. Some examples include:

- Which students are registered for which classes
- Which employees have which benefits options
- Which contractors have which skills
- Which facilities are reserved for which meeting

These are very easy to deal with using an intermediate *cross reference* table containing one record for each link with the record containing the primary keys values of both the other tables. This is easily demonstrated with a hokey example, and a very hokey example follows.

Penelope Puppylove has a dog act in the circus (didn't I warn you?). As part of her Dog Act Management System and Electronic Listing (DAMSEL), she wants to keep track of which puppies she has trained to do which tricks. Any puppy might be able to do several tricks, and she naturally trains more than one puppy to do each trick just in case one puppy is out of sorts for a show.

Immediately you can think of two tables, or files (I'll use the term table in this discussion, but for many flat-file database systems table and file mean the same thing). One table lists puppies, and one lists tricks. To deal with the many to many relationship between puppies and tricks you introduce a third table, often called a cross reference. I'll call it `DoesTrick` because it indicates which puppies do which tricks. Here are the three tables as I've set them up. You can look at them in the DAMSEL dictionary.

Puppy	DoesTrick	Trick
PuppySysID LONG	DoesTrickSysID LONG	TrickSysID LONG
PuppyName STRING(20)	TrickSysID LONG	TrickName STRING(20)
	PuppySysID LONG	

Notice that I've given each table a `SysID` field. This extra field is sometimes called a surrogate key, and its main purpose is to keep from having to put the `PuppyName` and `TrickName` in the `DoesTrick` table. Most experienced developers structure the data this way and make the `SysID` independent of any outside characteristic. If this was a database of people in the USA, you might be tempted to use the Social Security number as the primary key, but that isn't a good idea since social security numbers sometimes have to change and may be duplicated, often due to data entry errors.

I'll use the `SysID` as a primary key, and I'll set it as an autoincrementing key as well. The `PRIMARY` attribute is important for SQL databases, and the autoincrement setting causes Clarion code to be generated to do the autoincrementing.

The DoesTrick table could get away without a SysID since the two fields, PuppySysID and TrickSysID could make up a primary key. However I always put a separate field for each table's primary key because it is less trouble to have one you don't need, than to need one you don't have. SQL systems depend very heavily on the primary key to recognize which record is being updated.

Each table gets a primary key on its own SysID, and the DoesTrick table gets a key on the PuppySysID and TrickSysID fields. I also put a key on PuppyName and TrickName for convenience. The DoesTrick table gets two more keys, one on PuppySysID and TrickSysID, and the other on just TrickSysId. I marked the key on PuppySysID and TrickSysID to require a unique value so a trick can only be listed once for a puppy and vice versa. In some applications you might want to allow duplicates.

Two relations connect each Puppy record to many DoesTrick records by the PuppySysID keys, and each Trick to many DoesTrick records by the TrickSysID keys. The Clarion dictionary editor lets you define referential integrity to tell what you want done to the child records (DoesTrick) when the parent records (Puppy and Trick) are deleted or changed. Since I've used surrogate keys, there isn't any reason to change a SysID, so I have **On Update** set to **No Action**. I set **On Delete** to **Cascade**, so the DoesTrick records will disappear if the Trick or Puppy record is deleted. You could set the relationship between DoesTrick and Trick to **Restrict on delete** if you wanted to be sure a trick is never deleted if there is a puppy connected to it, which would be appropriate in many applications.

Notice that there are two ways of looking at the data. Penelope might want to look at all the tricks a puppy can do, or she might want to look at all the puppies that can do a trick.

The User Interface

Now that the data for this hokey application is defined, it's time to look at ways of presenting the data relationship to the user. The three options I use are "Form and Lookup," "Check List," and "Selection Pool."

Form And Lookup

In most Clarion programs the user edits a many-to-many relationship on the form where one of the tables is edited. A form and lookup interface is useful if there is extra information that needs to be stored in the cross reference record like how well the puppy does the trick.

Note: The example application contains completed procedures. If you want to follow along with the example application, you can either create new

Introduction to Databases

procedures with different names, or create a new example application and import demo app procedures as required.

If you create the DAMSEL data dictionary as I've described it so far, and let the wizards go at it, they will build you a Form and Lookup type interface. To show how such an interface works, I built it using the application generator. I started with an MDI frame as the main procedure with a menu option to call a browse of puppies and another to call a browse of tricks, look closely at my DAMSEL.App. Each of these browses calls a form to update the puppy or the trick. In the sample DAMSEL application, I named these:

```
LookupPuppyBrowse  
  PuppyLookupForm  
LookupTrickBrowse  
  TrickLookupForm
```

These probably aren't names you would use in a real application, but I used them here to mean "The example that uses a lookup to edit a puppy," and "The example that uses a lookup to edit a trick."

The main feature of PuppyLookupForm, in addition to the puppy name field, is a browse listing tricks the puppy can do. To build this browse, populate a browse box template onto the form. When the file schematic pops up, select the DoesTrick table, press the **Edit** button, and select the PuppySysIDKey. Since you want to show the puppy name and not anything out of the senseless DoesTrick table, hit Insert again, and select the Trick table from the list. Now, select the TrickName field from the Trick table. Now our browse box will show the names of the tricks rather than the SysIDs which Penelope doesn't know or care anything about.

Now go to the **Actions** tab for the browse. Select the PuppySysID for the range limit field, set the **Range Limit Type** to **File Relationship**, and select Puppy for the related table. Just for grins, fill in TRI:TrickName under **Additional Sort Fields** so the tricks will be listed alphabetically. See Figures 1 and 2.

Displaying Many-To-Many Relationships

Figure 1: Actions tab for the trick browse on the puppy form.

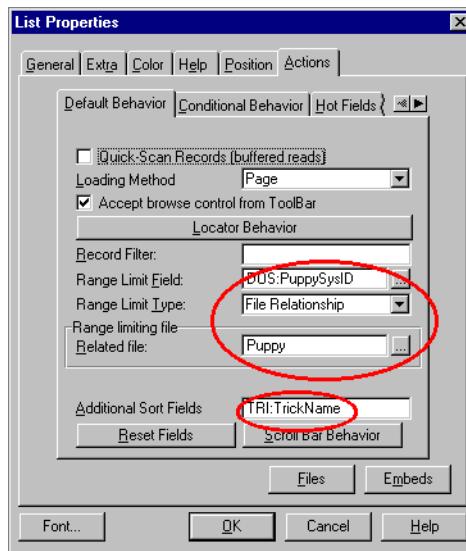
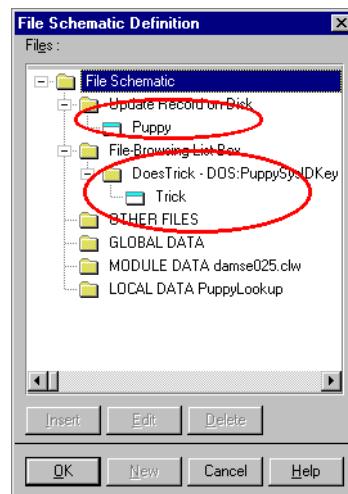


Figure 2: File schematic for the puppy lookup form.



Introduction to Databases

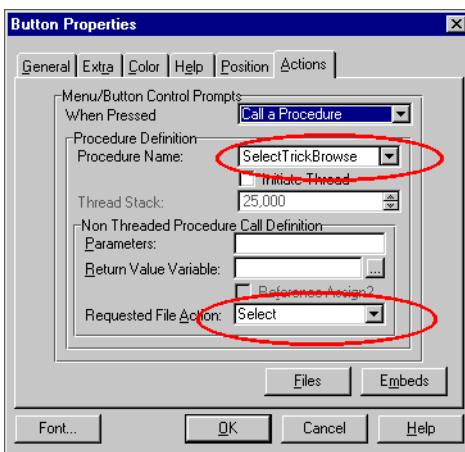
Now the form will show the list of tricks the puppy can do. More precisely, it shows the trick names from trick table that match the DoesTrick records which contain the puppy's SysID. All that's missing is a way to edit the list. Populate a set of browse update buttons, and set the update procedure to SelectTrickLookup (Select Trick for the **Lookup Example**). The working parts of this form are finished, so beautify it however you like and back yourself out to the procedure tree. I usually press the **Save** button any time I get back to the procedure tree, just for good luck.

There is now a SelectTrickLookup procedure labeled **ToDo**. Select this (or delete the existing procedure first to get a **ToDo**), and make it a form. This will be a very strange form since there won't be any entry fields. Instead, put a button, and label it "**Look up Trick**". You'll have this button call up a list of tricks to select from, so on its **Actions** tab set **When Pressed** to **Call a procedure**. Fill in SelectTrickBrowse, for the procedure, and set the **Requested File Action** to **Select**. (See Figure 3) The SelectTrickBrowse will be a browse to select a trick. Now you need an embed to set the DOS:TrickSysID when the user has selected a record, so press the **Embed** button to get the tree. Open **Control Events**, **Button3**, and **Accepted**. Select **Generated Code** and press **Insert** to put the embed after the generated code, that is, after the call to the browse procedure. Fill in the embed to look like this:

```
DOS:TrickSysID = TRI:TrickSysID
```

You're done inside this form, so back out to the procedure tree, and make the new SelectTrickBrowse procedure a browse on tricks. Use the TrickName key so the tricks will be listed alphabetically. If you didn't use the browse procedure template, add the select button template.

Figure 3: Actions tab for the Look Up Trip button.



Displaying Many-To-Many Relationships

So how does Penelope use this thing? If she's looking at the sample DAMSEL application, she'll select **Examples|Form & Lookup|Lookup Puppy**. This gives her a list of her dogs with the usual **Insert**, **Change** and **Delete** buttons. Add a dog called, say, Phyadoux. Just hit the Insert button. The form pops up, and she puts the dog's name in. There are no tricks listed, so hit **Insert** to add one. The form with the **Look Up Trick** button pops up, and pressing the button shows an empty list of tricks. Only the **Insert**, **Change** and **Delete** buttons don't work. There's no trick editing form specified here and the program is assuming the tricks are entered somewhere else.

To make the program more useful, go back to `SelectTrickBrowse` and look at the **Actions** tab for the update buttons. Check **Edit In Place**. This will let you add a new trick to the select list.

Now this application works, but to misquote Crocodile Dundee, "You can use it, but it works like..." Let's try making the Trick browse, form and lookup a little more graceful.

Select `LookupTrick` out of the procedure tree, and make it a browse procedure. In the file schematic, select `Trick` for the file browsing list box, and use the **Edit** button to select the `TrickName` key. Then go to the **Actions** tab of the update buttons and set the update procedure to `TrickLookup`.

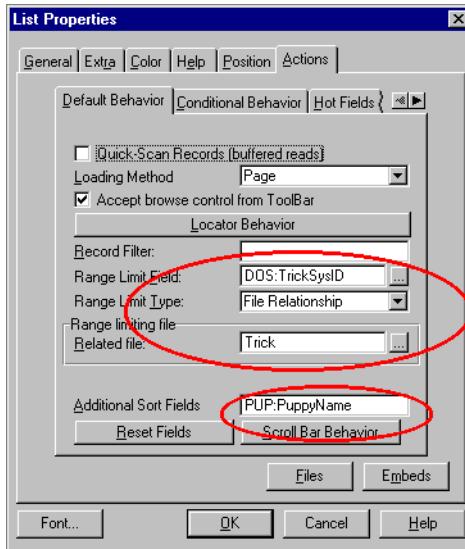
Make `TrickLookup` a form, and select `Trick` in the file schematic under Update Record on Disk. Populate the trick name control. Then put a browse template to show the puppies that do this trick. In the file schematic for the **File Browsing List Box**, select `DoesTrick`, and use the **Edit** button to set the key to the `TrickSysIDKey`. Hit **Insert** and add the puppy file under `DoesTrick`. In the list box formatter, put the puppy name.

On the **Actions** tab of the browse box, select the `TrickSysID` as the **Range Limit Field**, and set the **Range Limit Type** to **File Relationship**, and set the related file to `Trick`. (See

Introduction to Databases

Figure 4) In the **Actions** tab of the browse update buttons, put SelectPuppyLookup for the update procedure.

Figure 4: Actions tab for the puppy browse on TrickLookup.



When you make the `SelectPuppyLookup` form, use a file loaded drop box instead of a lookup button and browse procedure. Okay, it stinks from a consistency aspect, but the idea here is to demonstrate different ways of dealing with these many-to-many relations. Make the `SelectPuppyLookup` a form, and select `DoesTrick` as the table being updated. Put a **File Loaded Drop Box** template on the form. The **File Schematic** will pop up, but select **Local Data** instead of selecting a file. Press the **New** button on the right side, and add a `LocalPuppyName` variable that looks just like the `PuppyName` out of the `Puppy` table. I made it `STRING(20)` just for simplicity's sake. Select this new variable.

The file dialog will pop up again. Now it wants to know what to show in the list, so select the `Puppy` table and use the **Edit** button to set the key to `PuppyName`. Then the list box formatter will appear. Choose `PuppyName` as display field and close the formatter.

The clever part of a file loaded drop box template is all on the **Actions** tab. You will find two important blanks on the **Actions** tab, **Field to fill from**, and **Target field**. Set the **Field to fill from** to `PUP: PuppySysID`, and the **Target field** to `DOS: PuppySysID`. When the user selects an entry from the drop list, the program will put the `PuppySysID` from the selected puppy into the `PuppySysID` of the `DoesTricks` record. When it displays, it will find the correct puppy by matching the `PuppySysID` from the `DoesTrick` record and show the name. It actually does this lookup from the list box queue.

When Penelope adds a puppy to a trick, this little form with a file drop box will show up, and she can select a puppy from the drop list. If I had used a drop combo instead of a drop list, using the same technique, I could have checked the **Allow Updates** box, and Penelope could insert a new puppy right there.

Keep three things in mind when implementing many-to-many relationships using the Form and Lookup technique:

- Browse the cross reference table, and show fields from the other related table.
- Range limit the browse to show only cross reference records matching the record being edited.
- Edit the cross reference table, selecting a record from the other related table.

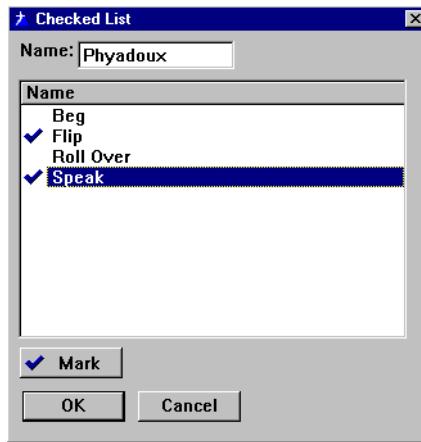
Now I'll show how to do this as a "Check List" and as a "Selection Pool."

Note: The example application contains completed procedures. If you want to follow along with the example application, you can either create new procedures with different names, or create a new example application and import demo app procedures as required.

Check List

A check box list presents a list of all the options available and indicates which are selected (See figure 5). I use it when the user thinks of tagging which options apply or don't apply. This is useful when the list of options is fairly restricted, but would get unmanageable if the list contained hundreds or thousands of items. To try this out, make a new menu option on your frame to call a browse of puppies, and make a new form procedure for the browse update buttons. On the new form, populate the puppy name field and a browse box.

Figure 5: A check box list.



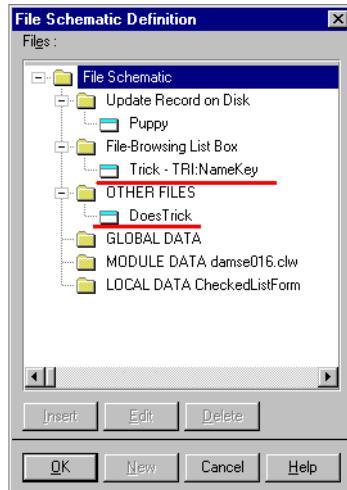
In the file schematic for the browse box, select the `Trick` table and use the **Edit** button to select the `TrickName` key so the tricks are listed alphabetically. In the list box formatter, add the Trick Name field, and select **Normal** under **icons** on the **appearance** tab. You won't need a range limit here since you want to show all the tricks.

The browse box will need to pick which icon to show in its `SetQueueRecord` method, and this information is in the `DoesTrick` table, so go to the procedure's file schematic and add the `DoesTrick` table under **Other Files** (See figure 6). You can have the `SetQueueRecord` method look in the table to see if there is a `DoesTrick` record for each trick, but in many database environments this would be inefficient. Granted, it

Displaying Many-To-Many Relationships

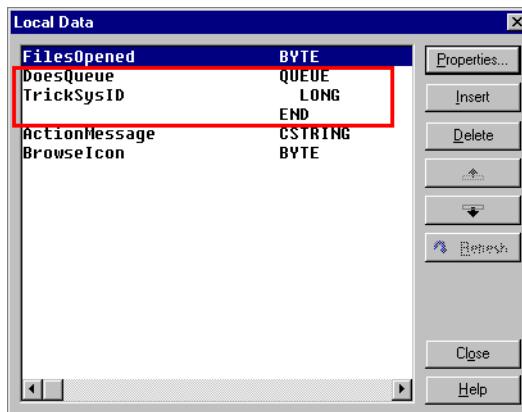
probably wouldn't matter in this application, but go ahead and do it the hard way just for fun.

Figure 6: File schematic for a check box list.



Next make a queue to hold the **TrickSysID** of all the tricks this puppy knows how to do, and refresh the queue when the browse is reset. I built the queue in the data area for the procedure using the **Data** button and gave it a prefix of **DQ**. (See figure 7)

Figure 7: A queue in the procedure's data.



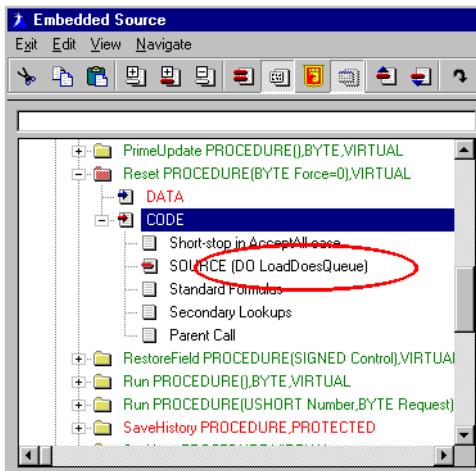
You will need some code to fill this queue. I put it in a routine:

Introduction to Databases

```
LoadDoesQueue ROUTINE
    FREE( DoesQueue )
    Access:DoesTrick.ClearKey(DOS:PuppySysIDKey)
    DOS:PuppySysID = PUP:PuppySysID
    SET(DOS:PuppySysIDKey,DOS:PuppySysIDKey)
    LOOP
        IF Access:DoesTrick.Next() <> LEVEL:Benign OR !
        DOS:PuppySysID <> PUP:PuppySysID THEN BREAK .
        DQ:TrickSysID = DOS:TrickSysID
        ADD( DoesQueue )
    END
    SORT( DoesQueue, +DQ:TrickSysID )
```

Put a call to this routine in the `Reset` method of the window (See figure 8), and in the `ResetFromAsk` method of the browse.

Figure 8: The ThisWindow.Reset embed point.



You will need to add a variable to the procedure. I called it `BrowseIcon` and made it a byte. Go to the browse **Actions** tab, and slide the tabs along till you find the **Icons** tab. Look at the properties for the `TRI:TrickName` icon. Press **Insert** to add a condition. The condition will be `BrowseIcon = 1`, and the icon will be whatever icon you want displayed for selected tricks. If you want to show an icon for unselected tricks, put that icon in for **Default icon**. Back out to the window formatter.

Next, put some code in the `SetQueueRecord` embed. Double click the browse box in the window formatter and find the **Local Objects|BRW5|SetQueueRecord** code embed, and press **Insert**. Then select **Source**. The embed code will look like this:

```
DQ:TrickSysID = TRI:TrickSysID
GET( DoesQueue, +DQ:TrickSysID )
IF ERRORCODE() OR DQ:TrickSysID <> TRI:TrickSysID
```

Displaying Many-To-Many Relationships

```
    BrowseIcon = 0
ELSE
    BrowseIcon = 1
END
```

You want this code *before* the generated code so the `BrowseIcon` variable will be set when the generated code looks at it to pick the icon. Otherwise, the icon will indicate whether the trick above it is selected.

Penelope (remember Penelope?) will want to be able to turn the icons on and off, so put a set of browse update buttons on the form, and fill in a new procedure name. I called it `CheckPuppyTrick`. You don't want to insert or delete tricks here, just insert or delete `DoesTrick` records, so hide the insert and delete buttons. You will need to tell the browse template that there aren't any insert and delete buttons, so go to the embed editor and find where the `BRW5.InsertControl` variable is set (about priority 8505). In the next available embed, put:

```
BRW5.InsertControl = 0
BRW5.DeleteControl = 0
```

Now, make the form procedure. I used the form template and selected `Trick` as the file being updated. List `Puppy` and `DoesTrick` under **Other Files**. You'll need to press the **Window** button to make a window so the template will be happy, but don't worry about what it looks like because the user will never see it.

Go to the embed tree for this "form;" look at the **Local Objects|ThisWindow|Init** embed and find where the files are opened. Put an embed after the files are opened which looks like this:

```
IF SELF.Request <> ChangeRecord
    SELF.Response = RequestCancelled
    RETURN LEVEL:Fatal
END
DOS:PuppySysID = PUP:PuppySysID
DOS:TrickSysID = TRI:TrickSysID
IF Access:DoesTrick.Fetch(DOS:PuppySysIDKey) |
    = LEVEL:Benign
    Relate:DoesTrick.Delete(0)
ELSE
    DOS:PuppySysID = PUP:PuppySysID
    DOS:TrickSysID = TRI:TrickSysID
    Access:DoesTrick.Insert()
END
SELF.Response = RequestCompleted
RETURN LEVEL:Fatal
```

This embed code does four things. First, it checks that it isn't being asked to insert or delete a record and sets `SELF.Response` to `RequestCancelled` so any calling browse will think the user opted to cancel. Second, it checks to see if there is a `DoesTrick` record for the `Puppy` and `Trick`. Third, if there is a `DoesTrick` record it deletes it, and if not, it

adds one. Fourth, it sets SELF.Response to RequestCompleted and returns LEVEL:Fatal.

The RETURN Level:Fatal statement will cause the form to close without ever displaying the window, and since SELF.Response has been set to RequestCompleted, the browse box which called it will think the user changed the record and do its ResetFromAsk method to refresh the display.

Notice that this form procedure doesn't care if it is called from a puppy form or from a trick form, so you can use the same procedure from a trick form.

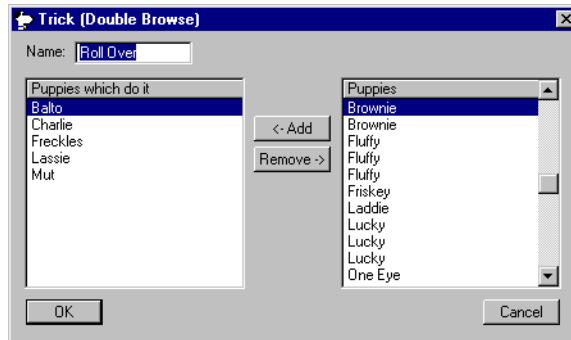
To make check lists, remember to:

- Browse the other table and put an icon.
- Make the icon conditional on some local variable.
- Use the SetQueueRecord method to look in the cross reference table and set the local variable.
- Make a form to add or delete the cross reference records but don't let it open its window.

Selection Pool

A selection pool is useful where a user thinks of picking an option to add. It shows two browses. One browse shows the selected options and the other shows the available options. This is useful when the user thinks about the relationship as "adding" one thing to the other.

Figure 9: A selection pool browse.



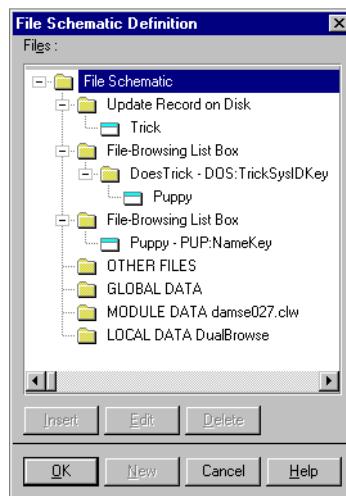
Displaying Many-To-Many Relationships

Make another option in your app frame menu to call up yet another browse, and make the browse show tricks. Make yet another trick update form, and place a control to edit the trick name. Make the form wide enough for two browses side by side with some space between them for buttons.

On the left side of the form, put a browse to show the DoesTrick table using the TrickSysIDKey key. Have it show the PuppyName out of the Puppy table. Range limit the browse on TrickSysID file related to the Trick table. Set an additional sort field to the puppy name so the puppies will be listed alphabetically.

Put another browse on the right side of the display to browse the puppy table. Yes, I know this table is used in the other browse, but don't worry, it will work fine (See Figure 10). Have it use the name key so the puppies will be listed alphabetically. You'll want to know the class names for these two browses, so look them up on their actions tabs and write them down or make them something meaningful. Mine are BRW5 and BRW6.

Figure 10: File schematic for a double browse form.



Now, look at the procedure's embed tree and go to **Procedure Routines**. Build yourself two routines. I usually put them in two different embeds so they both show in the embed tree. They should look like this:

```
AddPuppy ROUTINE
  IF CHOICE(BRW6.ListControl)
    BRW6.UpdateBuffer()
    DOS:PuppySysID = PUP:PuppySysID
    DOS:TrickSysID = TRI:TrickSysID
    Access:DoesTrick.TryInsert()
    ThisWindow.Reset(1)
  END
```

Introduction to Databases

```
RemovePuppy ROUTINE
    BRW5.UpdateBuffer()
    IF Access:DoesTrick.Fetch(DOS:DoesSysIDKey) |
        = LEVEL:Benign
        Relate:DoesTrick.Delete(0)
        ThisWindow.Reset(1)
    END
```

Take a close look at these two routines because they do the work.

AddPuppy first checks to see if a puppy is selected in BRW6. Second, it calls BRW6.UpdateBuffer so the puppy record will contain the selected puppy. Third, it builds a DoesTrick record and inserts it. The routine uses TryInsert instead of Insert so if the user mistakenly adds a puppy twice, the program will appear to do nothing rather than showing a strange looking error message. Fourth, it resets the window.

RemovePuppy does an UpdateBuffer on BRW5 so it knows which puppy it is removing. The browse is listing DoesTrick records, so BRW5.UpdateBuffer() will get the DoesTrickSysID into the DoesTrick record. It then fetches the DoesTrick record by the SysIDKey. Then it deletes the record and resets the window.

Now that you have these two routines, put DO AddPuppy in the embeds for the **Add** button (**Control Events|?Button3|Accepted**). Put DO RemovePuppy in the embeds for the **Remove** button.

Penelope can now add and remove puppies to her heart's content using these buttons. Since she's "moving" records from one list to another, this would be a natural place to use Drag and Drop, so go ahead and add drag and drop IDs to the list boxes. On the left side, the drag ID will be RemovePuppy and the drop ID will be AddPuppy. On the right side, do just the opposite and make the drag ID AddPuppy and the drop ID RemovePuppy. Double Click the browse on the left to get the embed tree and go to **Control Events|?List|Drop**, and add DO AddPuppy there. Double click the browse on the right and go to **Control Events|?List2|Drop**, and add DO RemovePuppy there.

You may want to restrict the right side browse to show only the puppies that are absent from the left side browse. I haven't figured out how to filter a view based on records that aren't there, so I built a queue in the procedure data to store the SysID's of puppies that do the trick and used the browse's verify record embed to check it. Performance shouldn't be too terrible unless there are 200,000 puppies and only two are included in the list. I gave the queue a prefix of PQ. An embed in ThisWindow.Reset() method fills this queue:

```
FREE (PuppyQueue)
Access:DoesTrick.ClearKey(DOS:TrickSysIDKey)
DOS:TrickSysID = TRI:TrickSysID
SET(DOS:TrickSysIDKey,DOS:TrickSysIDKey)
LOOP
    IF Access:DoesTrick.Next() <> LEVEL:Benign THEN BREAK .
```

```
IF DOS:TrickSysID <> TRI:TrickSysID THEN BREAK .
PQ:SysID = DOS:PuppySysID
ADD (PuppyQueue)
END
SORT (PuppyQueue,+PQ:SysID)
```

The embed to filter the browse looks like this:

```
PQ:SysID = PUP:PuppySysID
GET(PuppyQueue,+PQ:SysID)
IF ERRORCODE() OR PQ:SysID <> PUP:PuppySysID |
    THEN RETURN 0 .
RETURN 2
```

The code returns 0 if the puppy SysID is not in the queue, and a 2 if it is. Be careful! If you return a 1, the browse class will decide this record is out of range and will not look any farther.

To show a selection pool, remember to:

- Browse the cross reference table displaying identifying fields from the other table.
- Browse the other table.
- Make Add and Remove routines to build and remove the cross reference records. These should reset the window when they're done.
- Make controls and/or drag & drop to call the Add and Remove routines.

Summary

There are many ways to present many-to-many relationships to the user, and if you're clever you can hide the cross reference from the user. The trick is to figure out how the *user* envisions the task, and make an interface accordingly. I hope you find these three approaches useful.

Source code

See “Appendix A: Getting Support,” p. 601, for information on how to get the source accompanying this book.

- v1n10damsel.zip

TRUE CONFESSIONS: A TALE OF TWO USERS

by Thomas Ruby

The cost of knowing what you are doing is having learned the hard way. Somehow, it doesn't matter how many books, manuals, articles or true confessions you read, you never really learn until you've made the painful mistake yourself. I confess here in the hope that you'll recognize the mistake when you make it and know the cure. In the spirit of anonymity and protecting the innocent, I will tell the stories of "User 1" and "User B."

The Tale Of User 1

I helped build a large system with many tables and display screens. Many of these tables had a field to store the initials of the employee who had created or updated the record, sent a letter, called the client, or performed some other activity. Many tables had several of these fields. These were all three character strings. Later, I was in charge of making the second version (actually, the third), and we had a user table and surrogate keys, which we called Sequence Keys. These keys linked the users to their records by means of an arbitrary ID rather than the employee initials. I was, however, convinced by a co-worker to store the initials of the user everywhere for convenience of reporting. The reasoning went something

like this: “You don’t always want to be looking up the initials in the user table; it’s a lot of extra work.”

This system was in place, and working quite well. As time went on and the system evolved, we had lots of information linked together by these three-letter initial fields.

Then, it happened. User 1 got married. We all went to the wedding, and she was a beautiful bride. I don’t know where they went on their honeymoon because I didn’t want to seem nosy, but she was back at work about a week later. And her initials now were different.

To change her initials in the database we would have had to find the hundreds of places the three-letter field appeared in dozens and dozens of tables. If we just changed her initials in the user table, we would have lost the connection between her and all her work. She had two options: live with her old initials, or suffer a split personality. She decided keeping her old name was better than dealing with multiple personalities. If we had stored the Sequence Keys instead of the user’s initials, we could have just updated her record in the user table and her initials would have magically updated themselves everywhere they appeared.

- Painful Lesson #1: *Always* link your tables together by the surrogate keys.
- Corollary to Painful Lesson #1: The purpose of normalization is to make sure *updates* happen properly and with reasonable IO demands, not to reduce storage space.

Now there are certain situations where you might “over normalize” your data, and you need to be careful of this. Consider a sales tracking system. You probably have an item table of some sort with a price field. You might also have a sales record table, and you might be tempted to leave the price out of this sales record because you already have the item price in the item table. *Don’t do it!* These are actually two different price fields, the normal price for the item, and the price you sold the item for. If you change the price of an item from \$2.98 to \$2.35, you probably don’t mean to forget you sold it at \$2.98 to everybody you sold it to before the change.

The Saga Of User B

As the system evolved over time, I kept the painful lesson of User 1 in mind, and new development religiously followed the rule of Painful Lesson #1. Guess what? It wasn’t a lot of extra work or a nuisance, always having to look up the initials in the user table. Along the line, we implemented a “file cabinet” feature in which the users could store all sorts of information related to the customers they dealt with. There was an elaborate security system whereby users could grant access to certain records in the file cabinet to other users.

True Confessions: A Tale of Two Users

Now User B was the kind of user you always like to have. He understood the system, made full use of the features, enjoyed discovering ways the computer could make him more efficient, and abused his trusted position to personal gain at the expense of the company. Okay, so he wasn't *quite* the kind of user you like to have. His activities were discovered and his employment at the company was abruptly terminated. The owner of the company called up User B's record on the user browse and pressed Delete.

Now, I can't blame the owner of the company for wanting to eliminate all memory of this villainous fellow, but what he *really* wanted to do was call up the user record and set the status to Terminated.

It was a few days before the magnitude of the owner's mistake was fully understood. Remember, User B had made effective use of the system, including the "file cabinet" feature which he used to record miscellaneous information and make it available to his co-workers. The system had very happily deleted all the file cabinet contents related to his user record.

- Painful Lesson #2: You usually *don't* want users deleting data from your database.
- Painful Lesson #2b: The owner of the company *is* a user.
- Painful Lesson #2c: The user-friendly "Are you sure you want to delete User B" message wasn't any help. Of course, the owner was sure he wanted to delete User B. The trouble was, he didn't know why he did not want to delete User B.
- Painful Lesson #2d: A comprehensive security system cannot keep users from misusing information you have entrusted them with. If you told the computer the person is trustworthy it has no choice but to believe you.

So here you have some lessons learned at my expense. Experience is a wonderful thing since it lets you recognize a mistake when you make it again.

Using Topspeed Files

USING DYNAMIC INDEXES WITH TPS FILES

by Bill Florek

Dynamic indexes are often overlooked as a way to efficiently access data from TopSpeed (TPS) files, especially if you are dealing with files that hold large numbers of records and a custom sort order and filtered subset is required. By using a dynamic index, you can eliminate the need to create additional file keys.

You can retrieve data from a TopSpeed file by either sequential or random access. However, to access records in some specified sort order, you are limited in the options that are available to you. You can use keys, views, or dynamic indexes.

Keys may be used to read data in a predefined sequence. This methodology is very fast, regardless of the number of records in the file. However, if you need to filter the data on fields other than the key elements, you must read all records from the file to determine which records do not belong to the filtered data subset.

Views may be used to create a user-defined sort sequence. Also, if a filter is applied to a view, only the records that match the filter criteria are returned. The problem with this methodology is speed. If a data file contains a few hundred to a few thousand records, this is a viable option. However, when dealing with tens or hundreds of thousands of records, views give the illusion that the application is “locked-up”.

It should be noted that *generated* browse, report and process procedures use views to access data. Views, when coupled with a key from the primary file, produce acceptable results, as long as record filtering on fields other than the key elements is not used.

Keys and views are not the only options available to you. Dynamic Indexes, or “static keys”, incorporate the features of keys and views into a single structure.

Dynamic index basics

Before you can use a dynamic index with a TPS file, you must declare the index as part of the file structure. This is accomplished by creating a file key and selecting Runtime Index as its type.

Before using a dynamic index, you need to *build* the index. . The following defines the Clarion language `BUILD` statement as it pertains to creating a dynamic index:

`BUILD (index , components , [filter])` where:

index is the label of the dynamic index.

components is a comma-delimited list of fields to sort on

filter is an optional expression to filter the records

See the Clarion language reference for a complete description of the `BUILD` statement.

Dynamic indexes create a temporary file that is exclusive to the user who built it (when the file is closed, the temporary file is deleted). This allows multiple users to create indexes specific to their needs without affecting anyone else. However, because an index is a *static* structure, updates to the file are not reflected in the index after it is built.

After the index is built, you may use it to access records in a sequential or random access manner. The `RECORDS ()` function will return the actual number of records in the index, which is very useful when creating a process procedure that uses a progress bar.

Note: When using Clarion versions after 5501, the `RECORDS ()` function returns the total number of records in the index *plus* the records in the file.

If an application is using the legacy templates, dynamic indexes may be used as the key on generated procedures such as browses and reports. However, if an index is used on a browse, the locator must be set to NONE. If a locator is required, you must handle it manually (that is, hand-code it).

In the ABC templates, generated procedures will not use a dynamic index. If you specify an index as the *key*, no sort order will be used. The reason ABC template procedures do not

directly support runtime indexes lies in the `FileManager` class. When a file is “registered” with the `FileManager`, the file’s keys and associated key fields are saved using the `FileManager.AddKey` method. This method retrieves a key’s component fields using the key property `PROP:Components`. Since a dynamic index has no fields defined until the index is built, the `FileManager` has no components to register. The key definition stored by the `FileManager` is used when setting sort orders, range limiting files or processing locators. Therefore, since no component fields are initially defined for a dynamic index, the `FileManager` does not know what indexed fields it is dealing with. This would be similar to creating a generated browse procedure where the primary file for the browse does not use a key but has “additional sort fields” defined. The view engine has to handle the record sorting internally.

Why use a dynamic index?

If the generated procedures using the ABC templates do not allow the use of developer-defined runtime indexes, what possible use could there be for them?

The volume of data stored in today’s business applications continues to grow, and files with hundreds of thousands to millions of records are becoming very common. When an application is first designed, it is almost impossible, and definitely impractical, to incorporate every conceivable sort order that may ever be required by the application into a file definition. However, by adding a dynamic index to the file definition, you essentially eliminate this problem. Remember that this discussion on dynamic indexes applies to TPS files and not SQL databases, although indexes are vital there as well.

As I stated earlier, views suffer from a speed problem in situations that require record filtering. The following example illustrates this fact:

Test file	two fields defined as string(10) with 262,000 records and no keys
Test Criteria	sort on field1 and filter on field2 by using <code>SUB(field2,1,1) = ‘M’</code> , resulting in 468 selected records
Results: Dynamic Index	1.08 seconds
Results: File using Sequential Access	4:06 minutes
Results: View using Order and Filter	9.09 seconds

This simple test shows that a dynamic index is much faster than a view, and processing a file sequentially should not even be considered unless the order is unimportant and few records will be filtered out. Although this is a very simple test, the same type of result holds true when very complex file structures are used.

Therefore, if you are presented with a situation that requires sorting and filtering a file's records so that they can be processed in some manner, and the file has a large number of records, a dynamic index may very well be the perfect solution.

Typical use

Dynamic indexes can be substituted for keys or views in almost any situation. One of the deciding factors in whether or not an index should be used is the number of records in the file, although performance will generally be better using a runtime index. As I mentioned earlier, the ABC template generated procedures do not directly support dynamic indexes, so the developer (i.e. the programmer) will need to do something that is becoming more foreign every day: *write code*.

To illustrate a simple, yet powerful use of runtime indexes, look at the following pseudo-code, which uses a BUILD statement to determine exactly which records a report will process, and in what order:

```
Access:file.open
Access:file.usefile
Open(ProgressWindow)
Display
Open(Report)
Build(DynNdx,sortorder,filter)
ProgressBar{PROP:rangehigh} = |
    records(DynNdx) - records(File)
Set(DynNdx)
Loop
    If Access:file.next() then break.
    ProgressBar{PROP:progress} = |
        ProgressBar{PROP:progress} + 1
    Print(ReportDetailBand)
End
Close(Report)
Access:file.close
```

In this example, a report may be printed in *any* sort order and filtered on *any* fields. Simply set the sortorder and filter parameters of the BUILD statement to whatever is required to generate the report. Also, as a side benefit, the progress bar is truly accurate. (Note: when calculating the records contained in the dynamic index, remember to subtract the file record count when using Clarion versions after 5501, as shown in the listing) Although you could use a view with order and filter properties, the report would take much

Using Dynamic Indexes With TPS Files

longer to generate and the end-user would not be informed as to the *true* progress of the report.

By simply replacing the “report specific” code (such as the print statement) with some other type of processing code, you can accomplish any record-specific task.

As is evident in the example, no range or filter checking exists in the main processing loop. Since all filtering, which is synonymous with range checking, is done in the BUILD statement, none of this code needs to be written. On this premise, multi-file filtering becomes a simple task with very little additional coding required. For example, the following code will process file1 in some key order and only include records on the report if a related record exists in the filtered subset of records in file2:

```
Access:file1.open
Access:file1.usefile
Access:file2.open
Access:file2.usefile
Open(ProgressWindow)
Display
Open(Report)
Build(File2DynNdx,sortorder,filter)
ProgressBar{PROP:rangehigh} |
= records(DynNdx) - records(File)
Set(File1Key)
Loop
  If Access:file1.next() then break.
  ProgressBar{PROP:progress} =
    ProgressBar{PROP:progress} + 1
  File2.DynNdxSortField = File1.RelatedField
  Set(File2DynNdx, File2DynNdx)
  If ~Access:file2.next() AND |
    File2.DynNdxSortField = File1.RelatedField
    Print(ReportDetailBand)
  End
End
Close(Report)
Access:file1.close
Access:file2.close
```

The statement

```
If ~Access:file2.next() AND |
  File2.DynNdxSortField = File1.RelatedField
```

takes into account that there may be multiple file2 records that match the file1 related field. The purpose of this type of coding technique is not to process (or in this case, print) file2 records, but to include file1 records in the result set if any related record exists in the filtered subset of file2.

By replacing this statement with

```
If ~Access:file2.fetch(File2DynNdx)
```

Using Topspeed Files

and removing the `SET(File2DynNdx, File2DynNdx)` statement, a unique relationship between `file1` and `file2` is accomplished. The following example illustrates this technique:

- `file1` is an invoice header file that contains a customer code that relates to a customer file
- `file2` is a customer file that contains various customer information
- a dynamic index is built on the customer file in customer code order and only includes records that have a specific zip code
- When processing through `file1`, records (invoices) may be included or omitted from processing based on whether or not the `file1.fetch(DynNdx)` is successful.

The methodology presented in this example may be easily adapted to ABC generated procedures. To do this, place the file with the dynamic index into the procedure's file schematic under **Other Files**. Do not place it under the primary file as a related file (it would become part of the generated view and the purpose of the dynamic index would be defeated). In the embeds for the procedure, place the build index statement(s) after the files have been opened in `INIT`. The record checking code that uses the index could be placed into a variety of places, such as the `ValidateRecord` or `TakeRecord` embeds.

Summary

Dynamic indexes, also known as runtime indexes, can be a useful tool when dealing with TPS files that hold a large number of records. When you need a custom sort order and/or filtered subset of records for a processing task, a dynamic index will generally produce much faster results than a view. However, to be able to realize these benefits, you must first overcome the fear of hand coding a procedure.

USING THE TPS ODBC DRIVER

by Vince Du Beau

In this chapter I will explore the possibilities of using the TPS ODBC driver with other applications. You might be asking yourself, “Why do I need this?” Imagine the following scenario:

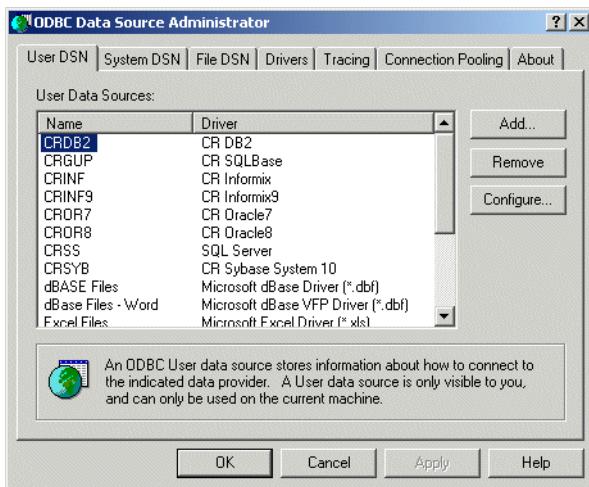
Your client is about to sign an agreement for you to build a killer application. Then you’re asked, “Will I be able to use the data in Excel or Word?” You can of course explain that you need to rewrite the quote to add export capability, or you can point out the TPS ODBC driver, and show the client how to use it (billable time, of course).

The ODBC driver I’m discussing here is the developer version that comes with Clarion. It will display a notice every time you access it. Do *not* distribute this driver to clients. Your clients will need to purchase their own licenses.

Setting up the ODBC driver

In this example I will use the invoice database provided with the Clarion examples. You first need to go into the ODBC Administrator. This is usually found in the either the Control Panel or the Administrative Tools, depending on your version of Windows.

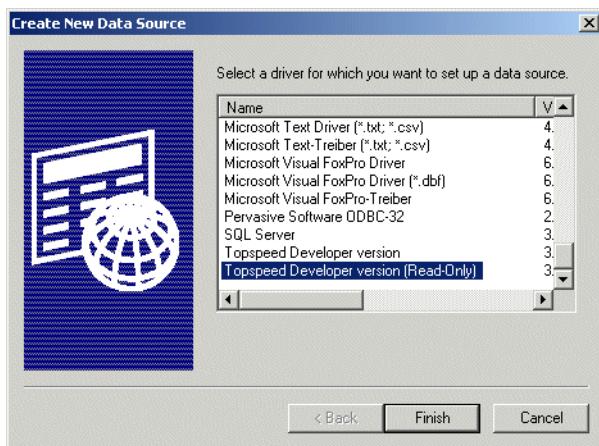
Figure 1: The ODBC Data Source Administrator, User DSN tab



On the **User DSN** tab, click on **Add** and you will be presented with the **Create New Data Source** dialog. Scroll down until you find the Topspeed drivers. You will find

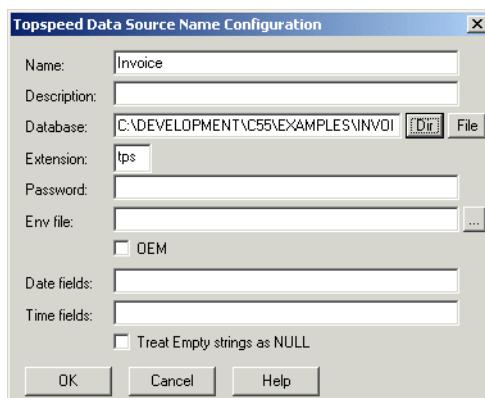
the normal driver and a read-only version. Setting this up for a client, you might want to choose the read-only version, as shown in Figure 2.

Figure 2: Creating a new data source



Click the **Finish** button and you will see **Topspeed Data Source Name Configuration**. Fill the dialog in as in Figure 3, replacing the data directory with the location of your example files.

Figure 3: Configuring the TPS data source



At the bottom of the configuration dialog, you will notice fields for Date and Time. If you specify fields from your table here, the driver will convert them to ODBC compliant dates or times. You can specify single fields, multiple fields or use wildcards for field names. The

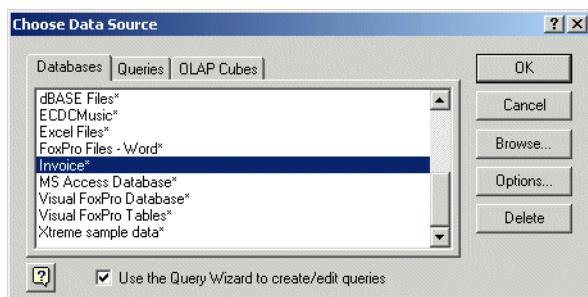
Using Topspeed Files

online help gives more detail and also provides some hints to converting Clarion LONG dates to other applications.

Making the connection

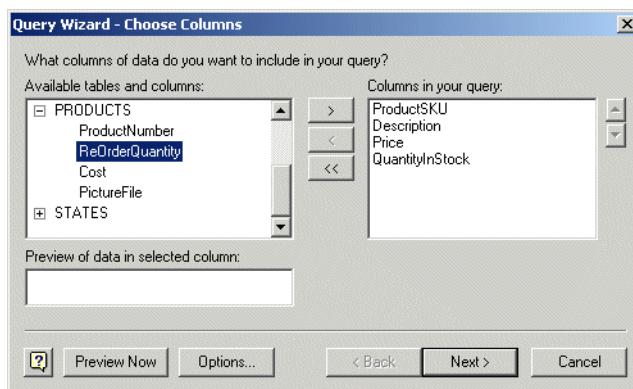
A good way to demo the ODBC capabilities to a client is by using Excel. Here's how to set up a simple spreadsheet using fields from the Products table. After opening Excel, you have to use the **Data|Get|External Data|New Database Query** menu. This will bring up the **Choose Data Source** dialog. Scroll down to the Invoice data source, highlight, and click **OK** as in Figure 4.

Figure 4: Choosing an ODBC data source in Excel



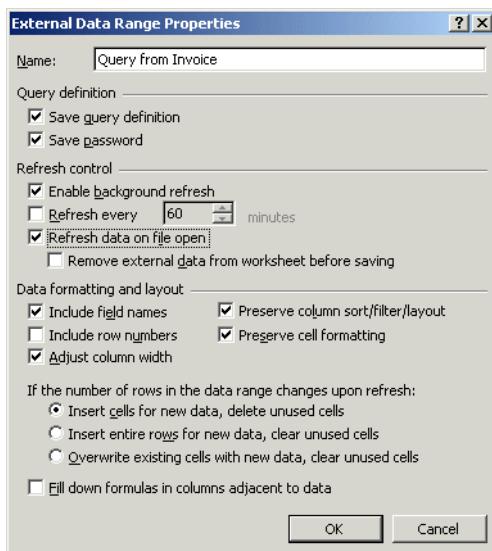
You will then have to choose the table and fields that you want to use for your query. For the demo, I've chosen fields as shown in Figure 5.

Figure 5: Selecting fields to import



The next two screens allow you to set filters and sort order. Click **Next**, **Next**, and **Finish** to skip through these and finish the process. The query will show a **Returning Data to Microsoft Excel** dialog. This dialog will let specify where you want the data returned in the spreadsheet. The properties button is what you will select. This brings up the **External Data Range Properties** dialog. You can play around with the options, but for now I'll set it up as in Figure 6.

Figure 6: External Data Range Properties dialog



Click **OK**, **OK** and you will see the Product data in the spreadsheet. Having checked the "Refresh data on file open" box, the spreadsheet will reflect any changes made to the data when it is open.

Summary

This was a very simple demonstration of using the TPS ODBC driver. I think this driver offers a lot of potential for promoting good will with clients by showing that their data is not isolated because of this strange thing called Clarion.

Using Topspeed Files

READING TABLES WITH ADO

by David Harms and Brian Staff

Have you ever wanted to write a generalized utility to handle a data file which may exist on more than one back end database? Do you need a utility to handle a file when you don't have (or want) a DCT layout? Have you ever wanted to use ADO (ActiveX Data Objects) in Clarion as a standard way of managing your data?

Clarion 6 does come with ADO support, but if you're using Clarion 5.x you can still do ADO by using Jim Kane's COM code, which included in the source zip, and which Jim described in more detail in his *Clarion Magazine* articles (www.clarionmag.com - search for author:Kane).

The RecordSet object

To use ADO, you create a RecordSet object which you use to retrieve data from one or more data files. A RecordSet object contains all the selected data as well as information about the data (metadata), such as field names, data types, and so on. RecordSets are a bit like ViewManager objects, except that you have to tell a ViewManager explicitly which fields you're working with, whereas you can tell a RecordSet to get, say, all the

fields in a file, and then you can ask the RecordSet what those fields are before you attempt to retrieve the data.

The RecordSet's Open method takes five parameters, all of which are optional:

`Source` - a variant data type which can be one of many things: a SQL statement, a table name, a stored procedure, a URL, an ADO Command object, or a file or Stream object containing a persisted record set.

`ActiveConnection` - a variant data type holding a Connection object, or a connection string identifying the data source

`CursorType` - a cursor type enum (see below)

`LockType` - a lock type enum (see below)

`Options` - information on how the source should be handled (see below)

The sample application

The sample code (160 or so lines) we describe in this chapter will allow you to use an ADO RecordSet object to read a TopSpeed file - although this can be any data file if you have an ODBC driver for it - and place the data in a standard Clarion LIST control. You can download the code at the end of this chapter.

Here's how the application works. To begin with, you need to know some standard ADO equates.

The cursor location equates refer to the cursor, or marker, used to keep track of your current position in the record set. Client cursors seem to work much better than server cursors, especially in a multi-user environment.

```
!---- CursorLocationEnum Values ----
adUseServer      EQUATE (2)
adUseClient      EQUATE (3)
```

Besides cursor location, you also need to choose a cursor type. Cursors not only manage your current position, they may also need to manage changes to the database (deletes, inserts) and possibly communicate information about those changes to other users. You pass the cursor type to the RecordSet object's Open method.

The `adOpenDynamic` equate indicates a dynamic cursor. With this cursor you can navigate backwards and forwards through a RecordSet, and you can see any changes made by other users.

The `adOpenKeyset` equate indicates a cursor like a dynamic cursor, with the following differences: you can't access records others delete; you can't see records others add; you can see data others change.

The `adOpenStatic` equate indicates a static cursor. This cursor keeps its own copy of the records you manipulate, so you can't see any changes others make.

The default cursor type is `adOpenForwardOnly`, which is the same as a static cursor but only lets you scroll forward through the data set. Generally speaking, `adOpenForwardOnly` will result in faster operation than `adOpenStatic`, though at the obvious expense of functionality.

```
!---- CursorTypeEnum Values ----
adOpenForwardOnly      EQUATE (0)
adOpenKeyset            EQUATE (1)
adOpenDynamic           EQUATE (2)
adOpenStatic             EQUATE (3)
```

The `LockTypeEnum` values indicate how ADO handles concurrency, and you pass this value to the `RecordSet`'s `Open` method. If you use `adLockReadOnly`, you can't alter data at all; `adLockPessimistic` indicates that the data source will lock records as you retrieve the data to begin your edit; `adLockOptimistic` indicates that the data source will lock records only when you issue an update; and `adLockBatchOptimistic`, as you might guess, applies optimistic locking to batch updates.

```
!---- LockTypeEnum Values ----
adLockReadOnly          EQUATE (1)
adLockPessimistic        EQUATE (2)
adLockOptimistic         EQUATE (3)
adLockBatchOptimistic   EQUATE (4)
```

The `CommandTypeEnum` values indicate the type of command used to query a database and return data in a `RecordSet` object. Like cursor type and lock type, the command type is passed to the `RecordSet`'s `Open` method.

The `adCmdText` equate indicates that the command passed to `Open` as the `Source` parameter is to be evaluated as a command. For instance, if you wish to execute a `SELECT * FROM MyTable`, you pass this string to `Open` as the `Source`, with an `Options` parameter of `adCmdText`. The `adCmdTable` equate indicates that the `Source` is a table name, and all fields in the table should be returned in the `RecordSet`. The `adCmdStoredProc` equate, as you'd expect, indicates that the `Source` is a stored procedure that should be executed. The default is `adCmdUnknown`.

```
!---- CommandTypeEnum Values ----
adCmdUnknown            EQUATE (0008h)
adCmdText               EQUATE (0001h)
adCmdTable              EQUATE (0002h)
adCmdStoredProc         EQUATE (0004h)
```

Using Topspeed Files

The sample application includes a few more equates and variables, most of which are straightforward. Note that the ADO RecordSet is called `oFileX`, and is declared as an OLE object using Jim Kane's `oleTclType` base class.

```
oFileX &oleTclType
```

The purpose of the sample application is to connect to a data source using ADO and retrieve the records into a queue for display in a window. The first step is to set the name of the database, and the value of the connect string. The sample application uses the developer version of the TPS ODBC driver, but you can change this to any data source you like. Note that although the TPS file is actually called `cust.tps`, it's only necessary to supply the table name, not the physical file name (that is defined in the ODBC data source definition – see “Using The TPS ODBC Driver,” p. 97) :

```
ThisFileName = 'cust'  
strConnect = 'DRIVER={{Topspeed Developer version}};DBQ=C:\data\'
```

Now it's time to create the ADO RecordSet object:

```
oFileX &= NEW oleTclType  
oFileX.init('ADODB.RecordSet',0)
```

Next, create the SQL statement, and pass it to the RecordSet object with the appropriate equates:

```
sql = 'SELECT * FROM ' & ThisFileName  
oFileX.CallMethod('Open("'"&sql&"', "'&strConnect&"',  
"'&adOpenForwardOnly&"', "'&adLockReadOnly&"',  
"'&adCmdText&"'))
```

You don't need to know anything about the data source in advance - the application will examine the resulting data and retrieve the column (field) names and other data. This code returns the number of columns in the table:

```
Cols = oFileX.GetProp('Fields.Count')
```

The sample code loops through the available columns and retrieves the column names (up to the maximum supported by the display queue), using them to format the queue's columns:

```
LOOP j = 1 TO CLIP(Cols)  
s1 = |  
    oFileX.GetProp('Fields(' & j-1 & ').Name')  
?p1{PROP:Text}= CLIP(s1)  
?List{PROPLIST:Format,j} = |  
    (?p1{PROP:Width} + 4) |  
    & 'L(2)M|~Hdr~(2)@s30@'  
?List{PROPLIST:Header,j} = s1  
IF j >= max THEN BREAK.  
END
```

The following loop retrieves the record data from RecordSet and adds it to the display queue:

```
LOOP
    MyEOF = oFileX.GetProp('EOF')
    IF MyEOF <> 0 THEN BREAK.
    LOOP j = 1 TO CLIP(Cols)
        qs.colx[j] = |
            oFileX.GetProp('Fields('& j-1 &').Value')
        IF j >= max THEN BREAK.
    END
    ADD(qs)
    oFileX.CallMethod('MoveNext() ')
END
```

Now just close, kill, and dispose of the RecordSet:

```
oFileX.CallMethod('Close() ')
oFileX.Kill()
DISPOSE(oFileX)
```

That's how easy it is to use ADO with Clarion 5.x. The SoftVelocity ADO templates, in Clarion 6, make it even easier. But if you're not on C6 yet you may want to use some of this code for your own ADO explorations.

Source code

See “Appendix A: Getting Support,” p. 601, for information on how to get the source accompanying this book.

- v3n5ado.zip

Using Topspeed Files

ACCESSING TPS FILES VIA ASP

by Brian Staff

I'm guessing that most of you have spent years developing Clarion applications for your clients, where their data is stored in TopSpeed (TPS) files. I'll also wager that you've been asked more than once to explore the possibility of displaying some of that data on an Internet/intranet web page.

This chapter is designed to introduce you to a new development environment: Active Server Pages, or ASP. With ASP you can use the TopSpeed ODBC driver to display a TPS file (or any ODBC compliant data) in a browser, for your client.

ASP files typically contain VBScript, which is really easy to read if you can read Clarion and HTML. The VBScript I'll describe will use ADO methods/properties and an SQL statement to read the data from the TPS file(s). You could also use JavaScript or PerlScript, but VBScript is easier to learn, although it's not as rich as JavaScript.

Firstly a word about the contents of ASP files. You do not need any special third party software to generate ASP pages - they are pure text. All you need is an editor. I use my C55 editor (I like CTRL-2 and CTRL-Y). These ASP files are not that different in their function from Clarion's own template (TPL/TPW) files. TPL/TPW files have a mix of template language statements and Clarion language statements. In ASP files, VBScript and HTML are intermixed too. In TP? files, you can identify template language by the # character on

Using Topspeed Files

the front of each statement. In ASP files, server side VBScript code (that is, code that executes on the server before the page is sent to the client) is inside a `<% . . . %>` structure. The code outside of those structures is pure HTML tags, and is simply passed through to the client.

To test ASP on your own machine you'll need to install IIS (Internet Information Services) or PWS (Personal Web Service, included with MS FrontPage). You will need to set up a root virtual directory, which by default is `\inetpub\wwwroot`.

Your machine can now be both the client and the server. Launch ASP files in your browser by typing in a URL in this format:

```
localhost/listcust.asp
```

The `localhost` identifier is a special server name identifying your own (local) machine, and it resolves to an IP address of `127.0.0.1`. Do not use a syntax of `c:/yourDirectory/listcust.asp` - that will merely try to copy the contents of the ASP page to the client and it will not invoke the IIS process. You'll get a similar symptom if you double-click an ASP page in Explorer.

When you enter the correct URL, your browser sends that URL from the client to the server using the HTTP protocol. IIS (or PWS) on the server receives that request and loads the requested ASP pages and any `INCLUDED` files (I'll talk about these a little later). Once the file(s) have been loaded on the server, IIS then executes the server-side script to generate an HTML page, which is usually a combination of the HTML in the ASP page and any HTML which is generated by the server-side VBScript code. This page is then sent back to the client to display in the browser. Server-side scripts are never sent to the client, however client side scripts are sent to the client. More on those later too.

It is beyond the scope of this chapter to explain every piece of VBScript, however I do want to identify one line that will allow you to access your TopSpeed data: the connection string assignment. I will not be showing how to use data sources as defined in the ODBC administrator, but rather a direct connection string.

All ODBC drivers have their very own connection parameters. The TopSpeed version in its simplest form is like this:

```
strConnect = "DRIVER={Topspeed ODBC Driver};"  
    & "DBQ=c:\yourDirectory\;PWD=;"
```

I was unable to use the developer version of the TopSpeed ODBC driver in an ASP page. I suspect the cause is the driver splash window. Maybe SoftVelocity could fix that problem for all of the developers wishing to experiment with ASP.

Carl Prothman, a Microsoft Visual Basic MVP has compiled a list of ODBC connection strings (http://www.able-consulting.com/ADO_Conn.htm) for a number of applications and databases, including Access, Excel, MySQL, SQL Server, Sybase, and many more.

Here's an example of real-world TopSpeed ODBC driver connection string:

```
strConnect = "DRIVER={Topspeed ODBC Driver};DBQ=" |
& Server.MapPath("./") & "\;PWD=;"
```

ASP will convert the root virtual directory to a real directory string using `Server.MapPath` and then add a “\” on the end. The first step is to make a connection to the TPS directory

```
Set objConn = Server.CreateObject
("ADODB.Connection") objConn.CursorLocation = adUseClient
objConn.Mode = adModeRead
objConn.Open (strConnect)
```

The connection string using the DBQ value points to the data directory containing your TPS files, and the PWD value identifies a null password. You do not have to point to a specific TPS file with a TPS connection string. You merely point to a directory which contains one or more TPS files. The specific file you wish to read is identified in your SQL statements, i.e.:

```
sql = "SELECT * FROM customer"
```

In the above statement `customer.tps` is the TopSpeed data file. You could also use the following syntax:

```
sql = "SELECT * FROM ""customer.tps"" "
```

Next, use an ADO connection method to get a recordset, which is all the rows and columns of data specified by the SELECT statement:

```
set oCust = objConn.execute(sql)
```

Once you've retrieved the recordset from the TPS file, it's fairly simple to iterate through the individual records using the following syntax:

```
While Not oCust.EOF
...
oCust.MoveNext
Wend
```

Then it's a matter of writing the HTML tags so that the data is formatted appropriately for the web page, which is then sent to the client. Be sure to use the **View|Source** option in IE or Netscape to see exactly what was sent to the client browser.

ASP objects

ASP, which is an environment, not a language, offers just six built-in objects, which the server-side scripts can use:

Object	Description
Request	Gets information from the user (client) that's passed in an HTTP request.
Response	Used to send information back to the client.
Server	Controls server-side activity, like creating component objects.
Session	Stores information about the current user session.
Application	Stores information for the entire lifetime of the application.
ObjectContext	Used to access the Microsoft Transaction Server system.

The most frequently used statement (`response.write`) is used to send text directly to the generated HTML file.

Included in the downloadable source for this chapter is the LISTCUST.ASP file, which you can see in action on the web by submitting the following URL in your browser:

`http://www.scoreboard.to/listcust.asp`

The file contains about 70 lines of code which is fairly easy to read. It uses the samples above to output a simple HTML table from a TopSpeed file.

It is important to note that whenever a VBScript variable is primed with an object, usually using the SET statement as in the following syntax:

```
set oCust = objConn.execute(sql)
```

That variable must be cleared after you have finished with it. This is to avoid memory leaks, and it is done with the following statement:

```
set oCust = nothing
```

LISTCUST.ASP

Here are the basic building blocks for inside the LISTCUST.ASP file. The actual file does contain some more padding, like comments and error checking. This first part describes the language to be used for the server-side scripts, and I will not allow implicitly declared variables.

```
<%@ language="VBScript" %>
<%
    Option Explicit
%>
```

Here is where I include any other files I will be needing, the first one being an equates-type file for ADO.

```
<!-- #INCLUDE FILE="adovbs.inc" //-->
<!-- #INCLUDE FILE="stt.inc" //-->
```

Here I declare the beginning of the HTML file to be sent back to the client.

```
<html>
<head>
</head>
<body style="text-align: center; background-color: silver;">
```

Now, I revert to server-side scripting and declare some VBScript variables, define the connection string and open an ADO connection object:

```
<%a
dim objConn
dim oCust
dim sql
dim strConnect
strConnect = "DRIVER={Topspeed ODBC Driver};DBQ="
    & Server.MapPath("./") & "\;PWD=;"
Set objConn = Server.CreateObject ("ADODB.Connection")
objConn.CursorLocation = adUseClient
objConn.Mode = adModeRead
objConn.Open (strConnect)
```

Here is the SQL statement to be used inside the execute method of the connection object which will return a recordset object.

```
sql =  "SELECT "
    " sysid,"      &_
    " company,"    &_
    " city,"       &_
    " country "    &_
    "FROM "
    " customer "   &_
    "WHERE "
    " sysid = sysid " &_
    "ORDER BY "
    " country, "    &_
```

Using Topspeed Files

```
" city, "      &_
" company "    &_
" "
set oCust = objConn.execute(sql)
```

Next, I define the HTML table and specify the table headers.

```
response.Write "<table border='2' cellspacing='0' cellpadding='3'
width='auto' style='background-color: #eeeeee; '>" &
vbCRLFresponse.Write "<caption>List of Customer Records</caption>" &
vbCRLFresponse.Write "<tr style='background-color: gray; color:
white; '>" & vbCRLFresponse.Write
"<th>Company</th><th>City</th><th>Country</th>" response.Write
"</tr>" & vbCRLF
```

I now iterate through the recordset rows, and create the HTML for each row in the HTML table:

```
While Not oCust.EOF
    response.Write "<tr>" response.Write "<td>" &
        oCust("company").Value & " </td>" response.Write "<td>" &
        oCust("city").Value & " </td>" response.Write "<td>" &
        oCust("country").Value & " </td>" response.Write "</tr>" &
        vbCRLF
    oCust.MoveNext
```

At the end of the recordset, the HTML table must be closed, and the objects cleaned up:

```
response.Write "</table>" & vbCRLF
oCust.Close
objConn.Close
clearDims()
sub clearDims()
    Set oCust = Nothing
    Set objConn = Nothing
end sub
%>
```

Lastly, I must declare the end tags for the HTML file:

```
</body>
</html>
```

That's it! It's really quite simple. The vbCRLF is added to some of the generated code so that the resultant HTML code that gets sent to the client is nicely formatted.

The actual LISTCUST.ASP file contains just enough clues to make anyone familiar with Clarion dangerous on the internet. Be my guest, and modify it for your own purposes. The most common coding mistakes you will likely make are using the wrong quotes character to identify a string, and omitting the word THEN at the end of an IF statement line.

In the downloadable source at the end of this chapter I have included a GLOBAL.ASA file which contains some optional event handlers for the application, and ADOVBS.INC, which is an equates-type file for ADO. Both of these need to be in the virtual root directory. I have also included an empty STT.INC file. Ordinarily, multiple client requests to the same

ASP page at the same time is not a problem, but the TopSpeed driver seems to have a weakness with windows threading. STT.INC will make it possible to throttle the requests to a single user at any one time.

If you happen to get the dreaded ASP 0115 (table not found) error displayed in your browser window, you will have to restart IIS or, if you're using PWS, reboot your machine...sorry about that! It appears to be a TS ODBC driver problem. This is almost certainly caused by the connection string not pointing to the correct directory containing your TPS file.

I encourage you to learn HTML thoroughly. The specifications can be found at <http://www.w3.org/TR/html401> and there is an on-line validator available at <http://validator.w3.org> which will check for well-formed HTML. You can either submit your URL or upload an HTML file that site.

Finally, here are some further resources for ASP:

- <http://www.w3schools.com>
- <http://www.aspfaq.com>
- <http://www.infinitemonkeys.ws/infinitemonkeys>
- <http://www.15seconds.com>

Source code

See "Appendix A: Getting Support," p. 601, for information on how to get the source accompanying this book.

- [v4n05asp.zip](#)

Using Topspeed Files

USING EXAMPLE FILES WITH TPSFix

by John Heck

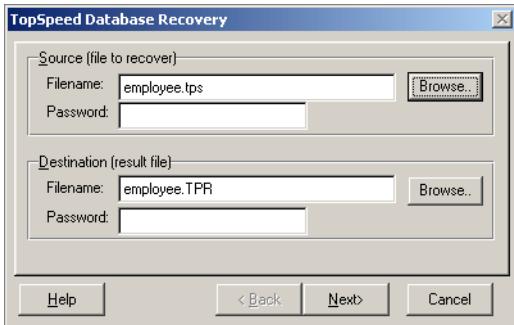
I had been working with an application and the Employee file became corrupt. So I did the normal run of TPSFix and selected the corrupt Employee file to fix. After TPSFix completed I deleted the corrupt file and renamed the Employee.TPR file to Employee.TPS. The file worked fine for a while and then it became corrupted again. After a point in time it became so corrupted that I could not even view it through the TPS scanner.

I found that there is a way to resolve this kind of problem when using TPSFix. The first step is to create an empty TPS file via the application dictionary. So for this example you would create an empty Employee.TPS file. Now rename the file to Employee.TPE; this file will be used on the second screen of the TPSFix process.

Using Topspeed Files

Start TPSFix and complete the first screen with the name and location of the corrupt data file (see Figure 1). Accept the default ‘Destination (result file)’ and press the **Next** button.

Figure 1: Specifying the source and destination files



The second screen allows the user to enter an example file. This file is the empty Employee.TPS file that was renamed to Employee.TPE, as in Figure 2.

Figure 2: Setting the example file



When you use the **Browse** button to select the example file, the file dialog uses the default extension .TPE. This example file is nothing more than a file with a good header definition record. TPSFix then uses this file to rewrite the header and data from the corrupt file, rather than using the header from the corrupt file, which could be causing the problem.

Press the **Start** button to start the fix process. The corrupt file should not be a problem any more after this process is complete.

TOPSPEED DRIVER ERROR CODES

Many Clarion developers use the TPS file format for small-to-medium sized databases, and with good success. Through Clarion release 4A, however, some users reported regular data corruption problems.

With driver fixes in 4B these problems seem to have largely gone away, but some forms of corruption are still outside the control of the driver, and network installations can be a particular problem. Anyone storing TPS files on an NT server with Windows '95 workstations should be sure to install Microsoft's redirector patch. You can download it from Microsoft:

<http://support.microsoft.com/support/kb/articles/q148/3/67.asp>

Thanks to Steve Mull for providing the link.

The following table lists the possible Topspeed error codes (thank you to Nigel Hicks for providing this).

Using Topspeed Files

Error	Description
112	Not a TPS error, this is an OS error meaning out of disk space.
231	Trying to append a record when the Btree is marked read-only
232	Cannot get btree header when trying to append a record
256	Cannot get btree header when trying to get a record;
300	Btree structure corrupt (discovered when deleting a record)
327	Btree structure corrupt (discovered when putting a record)
337	Trying to put a record when the btree is marked read-only
522	Invalid data size found while unpacking record (from disk)
530	Invalid repeat count found while unpacking record (from disk)
706	Btree structure corrupt (discovered when inserting a record)
780	Btree structure corrupt (discovered when removing a record)
824	Btree record size too big (on allocation)
1013	Cannot get btree header (while loading root page)
1043	Cannot get btree header (while packing a record to disk)

Topspeed Driver Error Codes

1163	Trying to create new record ID when the btree is marked read-only
1164	Cannot get btree header (while allocating new record ID)
1173	Maximum record ID reached on allocation (probably indicating file corruption)
1194	Trying to insert record when the btree is marked read-only
1203	Trying to remove record when the btree is marked read-only
1258	Btree structure corrupt (obsolete in C5)
1364	Btree structure corrupt (discovered while splitting a page)
1477	Btree page size (from header) does not match size stored on disk, could result in a truncated file, run tpsfix
1602	Btree unpacked page size (from header) does not match size loaded from disk
1659	Btree page size increased after packing
1678	Btree page size larger than maximum allowed
1735	Btree header corrupt (discovered when trying to calculate disk file size)
1781	Btree structure corrupt (discovered while shifting btree pages up)

Using Topspeed Files

1891	Btree structure corrupt (discovered while moving btree pages)
1894	Btree structure corrupt - invalid page parent level (discovered while moving btree pages)
2172	Reading of btree page from disk failed
2183	Too many files logged out
2272	Encryption block invalid size (discovered when reading page from disk 32-bit)
2277	ReadFile Win32 API function failed (when reading page from disk)
2286	Encryption block invalid size (discovered when writing a page to disk)
2328	Invalid internal btree locking mode (on lock (32 bit))
2352	Invalid internal btree locking mode (on unlock (32 bit))
2341	SetFilePointer Win32 API function failed (when reading page from disk)
2361	UnlockFile Win32 API function failed (obsolete on C5)
2447	16 bit close file failed
2458	16 bit handle duplicate failed (during commit)
2460	16 bit close file failed (during commit)
2476	Invalid internal btree locking mode (on lock (16 bit))
2519	Invalid internal btree locking mode (on unlock (16 bit))

Topspeed Driver Error Codes

2528	Unlock DOS function failed (obsolete on C5)
2572	Encryption block invalid size (discovered when reading page from disk 16-bit)
2582	Encryption block invalid size (discovered when writing a page to disk 16-bit)

Using Topspeed Files

TROUBLESHOOTING TPS FILE CORRUPTION

by Eric Vail

Although TPS files are generally quite reliable, every once in a while someone posts an urgent message in the SoftVelocity newsgroups about TPS file corruption on a network. Eric Vail recently posted the following list of questions and answers, reprinted here with Eric's permission. For additional resources, see Mark Riffey's "Resolving Network And Other File Problems," p. 129.

Questions to ask yourself

- Is your program the only application sharing data over the LAN?
- When did data corruption begin?
- What changed in the LAN at that point?
- Was there a new program loaded or some new hardware introduced?
- Is there enough open space on the server where the data is stored?
- When is the last time the cooling fans were checked on the processors in the server and/or the case for that matter? When you are running a database the

server processors run hotter than usual so if the fans are failing it can cause the server to mess up.

- Does the server ever lock up? What about the workstations? This could be caused by a flaky power supply and/or RAM.
- When was the last time the server or workstation disks were defragged? (NT does *not* defrag on the fly like Novell.) There are some utilities out there, but usually we partition the drives C and D and put the boot and system stuff on C and all programs and data on D. This way when we do maintenance we can just copy away the D data somewhere, then reformat the drive and do a scandisk before copying back the programs and data. On W2k you can defrag the partitions in the computer manager.

Verifying network integrity

If none of the above solves the problem, then I would start looking at network hardware, specifically the network card and cables for the server. Here is an easy way to verify network integrity.

Get a block of data that you can verify size and number of files on. I usually use the I386 directory from an NT CD. It is large and has lots of big and little files in it so you get a good sampling of all data types and sizes.

Now put that directory onto the server somewhere and then copy it from the drive on the server in the following sequence.

- 1) Copy from the server to each station. Time it and see how long it takes, then make sure that *all* the data makes it. Right-click on the folder and choose **Properties**: make sure the number of files matches the server. Also make sure the total size of the files matches - *not* the space the files take up on disk, but the *total file size*, which is the number above the total disk space. That is the true size of the files.
- 2) Next do the same from station to station.

If you find that one station-to-station copy is slower than another, or some files were dropped, move the cable in the hub first and make sure that the hub is okay. I have seen where one port in a hub will go bad and cause all kinds of flaky problems. The problem could also be a loose cable or a bad network card.

If all server-to-workstation and workstation-to-workstation copies are the same then you need to go back to the server and check the following:

Troubleshooting TPS File Corruption

- 1) How much RAM is in the server and what type. The problem may be memory going bad.
- 2) What type of hard drives are in the server and what configuration. If they are mirrored are they healthy? If RAID 5, are they optimal?
- 3) Is the swap file size adequate? It should be twice the size of the physical RAM if you are sharing data files on it. That is not the default setting by the way. So if they have 256 meg of RAM, then it should be 512 meg minimum, with a 768 meg ceiling.

Using Topspeed Files

RESOLVING NETWORK AND OTHER FILE PROBLEMS

by Mark Riffey

Problems with TPS files on a network, such as those with symptoms like error 1477 and 2172, can take many forms. They can be caused by one or more of the following:

- Kinked or damaged cable. Just because it looks okay doesn't mean it is - test it or swap it out for another one you may have.
- Cable running close to a fluorescent light ballast (fixture)
- Loose connector/plug on cable
- Old, worn out cables, particularly coax cables that have been around for years
- Out-of-date drivers
- Bad hub, or a bad port on a hub
- Failed/failing network card

Using Topspeed Files

- Power problems. *Please* protect your systems with a UPS. Yes, a UPS might cost \$79 to \$450 depending on how big a unit you buy, but how much is your computer and a day's worth of business worth? Consider it much less than the time to fix a power-caused mess.
- Network setup and/or configuration problems.
- Inadvertent shutdowns
- Shutting down servers while workstations are still in the program
- Out-of-date network drivers (even those right out of the box are sometimes a problem)
- Improper or less than desirable network bindings/settings

Note: Remember that having backups is a saving grace in the face of file/network problems. Network problems can corrupt your files in a heartbeat. If you have no backups, you are in big trouble. Having backups is a responsibility you must take *very* seriously.

Drivers up to date?

Windows networking is subject to a number of problems, *many* of which can be solved simply by installing updated driver software from the manufacturer or (more often) Microsoft. The link below will go to a web page that describes just one of the problems in Windows peer-to-peer networking, yet there are several other problems referenced at the bottom of that page. In particular, anyone on Windows 95 needs to get their network drivers and requestor updated.

<http://support.microsoft.com/support/kb/articles/q174/3/71.asp>

<http://support.microsoft.com/support/kb/articles/q148/3/67.asp>

Windows NT users

Are you on service pack 6 instead of service pack 6a or another service pack? If so, expect lots of problems. Microsoft has acknowledged that service pack 6 broke a lot of things network-wise. You can get service pack 6a at the MS site, or you can go back to service pack 5, either of which is stable. In addition, do *not* mix service packs on different NT machines on your network. In other words, run all your NT machines on service pack 5 or on service pack 6a, but not a mix of both service packs.

Is your network slow when using a mapped drive letter?

The reason is this: The computer has both TCP/IP and NetBEUI protocols, TCP/IP for the Internet and NetBEUI for the local network. TCP/IP is the default protocol. When connecting to a mapped drive after some idle time, the computer tries to connect first over TCP/IP and times out. Then, and only then, it tries the NetBEUI connection. Go to the Control Panel, and choose **Networks|Bindings**. Make NetBEUI the default protocol.

Is your network slow when using a mapped drive letter? (Part 2)

Is the drive mapped to the main computer's drive, or to a folder? If it is mapped to a folder, you will likely see a decrease in performance, often a quite noticeable decrease. I am not sure why this happens, but mapping directly to the drive has been proven time and time again to be faster.

Windows 98 networking

Here is Microsoft's "best place to start" page for dealing with Windows98 issues, including networking issues:

<http://support.microsoft.com/highlights/w98.asp>

Windows ME (Millennium) networking

Here is Microsoft's "best place to start" page for dealing with Windows ME/Millennium issues, including networking issues:

<http://support.microsoft.com/highlights/winme.asp>

Windows 2000 networking

Here is Microsoft's "best place to start" page for dealing with Windows 2000 issues, including networking issues:

<http://support.microsoft.com/highlights/Win2000.asp>

Windows XP networking

Here is Microsoft's "best place to start" page for dealing with Windows XP issues, including networking issues:

<http://support.microsoft.com/highlights/winxp.asp>

Using Topspeed Files

Need Netbeui on your XP systems and can't find it?

Here's how to install Netbeui:

<http://support.microsoft.com/search/preview.aspx?scid=kb;en-us;Q301041>

Workstation drive letters “getting the red X” (disconnecting from the main computer)

You can disable this by issuing this command from the DOS command line:

```
net config server /autodisconnect:-1
```

Before using this command, read the Microsoft article that discusses autodisconnect:

<http://support.microsoft.com/default.aspx?scid=kb;en-us;138365>.

Windows 2000 or Windows XP mapped drives disconnecting for no apparent reason? (showing the red X over the drive in explorer)

For further information:

<http://support.microsoft.com/default.aspx?scid=kb;en-us;138365>

Novell Netware problems?

The problem could be your Novell Opportunistic Locking setting. Contact your network person for further details. How to turn it off? Go to **Control Panel -> Networks -> Novell Client Properties -> Advanced Settings Tab -> Opportunistic Locking** and make sure this is switched off on all client Machines. Also *make sure True Commit is ON* at each client PC (this should help stop data corruption).

Performance issues are often caused by network protocol “bindings”

Check the following Network protocols basics:

- Make sure that your default network protocol has no bindings to a virtual device (dialup.....).
- If you are using TCP/IP and you have dialup on this workstation, try NetBEUI.

- Try to avoid using IPX and NetBEUI together. IPX gets confused when you have a “chatty” NetBEUI. Removing IPX (if you can) is strongly advised.
- If you need to examine the network further, check out <http://www.sysinternals.com/tdimon.htm> to get a bird's eye view of what's going on.

Does the system work on some machines, but seems to think about it and then do nothing on others?

Sometimes your network times out when loading a large application across a network. Try installing the application locally.

Sometimes your Windows doesn't have enough files set in your config.sys. Try 100 or 125. Sometimes having full-time virus scanning turned on does this. Ask your virus software vendor how to work around this *or* exclude the application from your scanner if you can.

Power management

Do you have Energy Star features on your computers? Probably so. Power management and networking do *not* mix. You can have your power management features turn off and/or dim the monitor, but do *not* have them turn off the hard drive, network cards, etc. This will definitely cause you grief, and grief = lost data

Database corruptions, timeouts and other troubles

Another issue is the various ways that Windows9x and NT try to improve performance, often at the price of stability. Sometimes these things work, other times they cause network timeouts because they force additional file operations behind the scenes, and those file operations time out. One way to improve things is to turn off synchronous buffer commits. To do this, click **Control Panel, System, Performance, File System, Troubleshooting** and check the **Disable synchronous buffer commits** checkbox.

Database corruptions, timeouts and other troubles, Part 2

Windows NT users face issues caused by some performance improvements that NT tries to implement with network applications by “faking” multiple use of files. Unfortunately, some users experience file corruption because of this. Take a look at the following article if you are seeing Access denied errors on network files when the network permissions are set properly:

Using Topspeed Files

<http://support.microsoft.com/support/kb/articles/Q129/2/02.asp>

The subject of this article can also be the cause of database corruption and network timeouts (drive not available messages and the like).

Win9x/Me users - Turn off write caching

You need to disable the write-behind cache. When the program ask to save the data, the data is kept in cache on the local machine until the cache is flushed, instead of being on the server. Click on:

- **START > SETTINGS > CONTROL PANEL**
- **System**
- **Performance tab**
- **Troubleshooting**
- **Performance**
- Disable the write-behind cache
- Restart the computer

Windows 2000 and Windows XP users - Turn off write caching

You need to disable the write-behind cache. When the program ask to save the data, the data is kept in cache on the local machine until the cache is flushed, instead of being on the server.

- Right-click **My Computer > Properties > Hardware > Device Manager**
- Right-click **Disk Drive > Properties**
- Disable: **Write Cache Enabled**
- Restart the computer

Opportunistic locking (oplocks) and performance

This white paper discusses issues related to opportunistic locking, something that can seriously impact performance on ISAM databases:

<http://www.dataaccess.com/whitepapers/opportunlockingreadcaching.html>

Also see the following Microsoft articles:

- **Some Client Applications Fail when writing to Windows NT**
<http://support.microsoft.com/default.aspx?scid=kb;en-us;q124916>
- **PC EXT: Explanation of Opportunistic locking in Windows NT**
<http://support.microsoft.com/default.aspx?scid=kb;en-us;q129202>
- **Event error 2022: Server unable to find a free connection**
<http://support.microsoft.com/default.aspx?scid=kb;en-us;q130922>
- **How the autodisconnect works in Windows NT**
<http://support.microsoft.com/default.aspx?scid=kb;en-us;q138365>
- **Locking error or Computer hangs Accessing network database files**
<http://support.microsoft.com/default.aspx?scid=kb;en-us;q142803>
- **Possible network file damage with redirector caching**
<http://support.microsoft.com/default.aspx?scid=kb;en-us;q148367>
- **Possible network data corruption if locking not used**
<http://support.microsoft.com/default.aspx?scid=kb;en-us;q152186>
- **How to disable network redirector file caching**
<http://support.microsoft.com/default.aspx?scid=kb;en-us;q163401>
- **Possible database file damage when data is appended**
<http://support.microsoft.com/default.aspx?scid=kb;en-us;q174371>
- **Improving performance of MS-DOS database applications**
<http://support.microsoft.com/default.aspx?scid=kb;en-us;q219022>
- **Configuring opportunistic locking in Windows 2000**
<http://support.microsoft.com/default.aspx?scid=kb;en-us;q296264>
- **Write caching settings for hard disk may not persist after you restart your computer**
<http://support.microsoft.com/default.aspx?scid=kb;en-us;q290757>

Tune up your network

Many of the aforementioned settings are automatically checked/corrected via a utility program called Network Tune Up, available at:

<http://www.studiomarketing.com/downloads/networktuneup.exe>

This program is free. Note that it does change internal network settings and requires a reboot afterwards. The settings changed include oplocks (on Windows NT and Windows

Using Topspeed Files

2000) and the Windows9x/WindowsMe buffering settings noted above. It also checks Windows9x/WindowsMe machines to be sure they don't have a buggy version of the Microsoft network driver installed.

Another NT issue

Re slow network performance with Service Pack 4, 5, 6, or 6a (Q249799):

<http://www.microsoft.com/technet/support/kb.asp?ID=249799>

Fix that leaky hose

While it is certainly possible that the problem is with an application, don't automatically assume network errors are a program problem. Do other multi-user applications work okay? Can you save a text file into the application's directory using Windows Notepad? If not, the problem is more than likely with the network setup. Just one little thing related to sharing or permissions can mess things up. Like a leaky hose where you don't see the leaks till lots of water is going through the hose under pressure, a network can exhibit similar behavior and not fail until it is under a heavy load.

Getting a TPSBT 1477 and/or 2172?

The Topspeed driver 1477 and 2172 errors are caused by improperly closed files. Improper closing can be caused by rebooting the server while the workstation is in the program, rebooting a workstation while it is in the program, logging out while you are in the program, having a power outage or even a burp in the power, and so on. The items noted above can help this situation as well.

The following link will download a program from GraniteBear that will detect which version of the Windows network redirector you have:

<http://www.studiomarketing.com/downloads/redirve4.exe>

Just run the application – there is no install. It may or may not point out a problem, and it is of no use if your server isn't Windows NT or Windows 2000. If you don't have the current network client/redirector, you are asking for trouble.

Just one more

Here's one more nice network troubleshooting resource:

http://farreachtech.com/network_troubleshooting.htm

Resolving Network And Other File Problems

Using Topspeed Files

General SQL

AN INTRODUCTION TO SQL

by David Harms

In a recent *Clarion Magazine* poll I asked developers how much of their development effort was directed at SQL databases. Out of 193 responses, 46% did no SQL development, 20% did half or less development for SQL, and 34% did most of their development for SQL.

Interest in SQL is at an all-time high in the Clarion development community, but many are still uncertain about whether to go to SQL, or how to make the switch. In this chapter I'll compare SQL databases to TPS databases, and suggest some reasons and strategies for moving to SQL.

What is SQL?

SQL stands for Structured Query Language, and can be pronounced either “sequel” or “es-queue-ell” depending on which side of the religious war you prefer. SQL was originally developed by IBM, and inspired by IBM researcher E.F. Codd. As the name suggests, SQL is an English-like language that lets you selectively retrieve data from a database. For instance, the statement:

```
SELECT FirstName, LastName FROM Names WHERE Country= 'Canada'
```

will retrieve the first and last names of all the Canadians from a table called `Names`. It could be that your application created that `SELECT` statement, or perhaps you typed it in yourself using a special database client. In the former case, your application will then display the names (perhaps in a browse); in the latter, the database client will display a list of names.

Actually SQL can do a whole lot more than just retrieve data. You can use SQL to create and modify databases, tables, and keys as well as update, insert, and delete data. You can also use SQL to enforce relationships between tables, creating what's called a relational database. (For more on relational databases, see Tom Ruby's series of chapters on database normalization, beginning with "Managing Complexity, Rule 1: Eliminate Repeating Fields," p. 21). A relational database doesn't have to be a SQL database, but in most cases you'll find that relational and SQL go hand in hand. That's because SQL isn't just a standard query language, it's also a standard language for defining tables and their relationships.

SQL is a bit like the standard Clarion language grammar for accessing databases. The Clarion language has `CREATE`, `SET`, `NEXT`, `PREVIOUS`, `ADD`, and `DELETE` statements which you can use on a variety of different file formats; all you need is a file driver appropriate for the data files you're accessing. SQL has `CREATE`, `SELECT`, `INSERT`, `DELETE`, and `UPDATE` statements which will work on any SQL database; all you need is a way of presenting those statements to the database.

If SQL and Clarion take similar approaches to handling data, why would you bother with SQL? Part of the answer lies in the differences between relational and flat file databases.

Relational vs. flat file databases

A lot of people think of Clarion as a relational database development tool. In fact, Clarion is really a database-agnostic fourth generation language, or 4GL. You can use Clarion with flat file and relational databases. And for those of you who think of TPS files as a relational database, nope, that isn't the case. TPS files are flat files.

A flat file database simply contains data files; there is no code involved in processing the data, rather, the application does all the work. If, for instance, you have an order entry application, you'll probably have something like an `Order` header table, which contains one record for each order, and an `OrderDetail` table, which contains one record for each item purchased as part of a given order. If you delete an `Order` record, and there are related `OrderDetail` records, you could leave behind orphaned records. To prevent this you'll want to either delete those related records, or prevent the deletion of the `Order`

record. In a flat file database your application has to manage the relationships between tables; in a relational database, this is the server's job.

Note: The term *database server* has two common meanings: one is the software that manages the database, and the other is the physical computer that holds the database and the database server software. In most cases, database server software is installed on a dedicated machine, so that other processes don't slow database handling.

Clarion is quite good at handling relationships between tables in a flat file database. You can define those relationships in the data dictionary, and by clicking a few options you can tell Clarion to generate appropriate code to cascade or restrict deletes, and so forth. What Clarion can't do, however, is stop any other program from violating the rules you've so carefully defined in the dictionary. Your data is just sitting out there in a no-brain TPS file, waiting to be trashed by any program that can read the file.

Client/server and relational databases

Another term frequently used in database application development is *client/server*. If you're using a flat file (i.e. TPS database) on local computer, you're not doing client/server. If you place that flat file database on a server, so more than one program can work with the data at a time, you're still not doing client/server. In a client/server environment, both the client and the server have some intelligence. Relational SQL databases are a common example of client/server processing.

The database server's intelligence provides a number of key advantages over flat file databases, including speed, data integrity, compatibility with other products, ease of administration, and scalability.

The need for speed

A SQL database can provide dramatic speed improvements over flat file databases in a network environment (from this point on, I'll use the term "SQL database" to mean "relational SQL database," but keep in mind that not all SQL databases are fully relational). If you're using flat files such as TPS files, every time you request a record from a table, all of the fields in that record have to travel across the network. If your table has fifteen fields with a total of 500 bytes per row, and you only want to retrieve one 25 byte field, then you're moving 20 times more data across the network than you need! In most cases you'll want more than just five percent of the row's data, but if you even use only half of the available data, you've doubled your bandwidth requirements by using a flat file.

And since the flat file database contains no intelligence, it doesn't know anything about how tables are related. If your browse makes use of multiple tables, your application will have to retrieve full rows of data from the related tables as well, and match those results with the first table.

If you're using a SQL database, your application sends a `SELECT` statement to the database server, which sends only the requested fields across the network. Your application can also tell the database server to return fields in related tables; it's up to the server to decide how it locates this related data, and it only sends the requested fields back to the application. This also means the client computer doesn't have to expend processor cycles matching the related records.

Note: Reducing the client computer's processing load may or may not speed the application. Much will depend on the speed of the network, and the load on the database server.

Data integrity

I've already mentioned data integrity in the case of orphaned records. Although you can easily create a Clarion application that manages related table data, this puts the burden entirely on the application. In a shared database environment you often have multiple applications working with the same data. Even if you've created all these applications with Clarion, you'll need to ensure that you use the same dictionary, or maintain the same relationship settings across multiple dictionaries. And if someone wants to work with the database using, say, Excel, or a Visual Basic application, all bets are off.

In a heterogeneous environment, you're far better off putting core database integrity rules in the database itself. These rules can embody the sort of basic referential integrity (RI) options you see in the Clarion data dictionary (restrict or cascade changes and deletes), and they can specify default and allowed values for individual fields. In most SQL databases you can also create triggers, which execute SQL code when a certain action (like a delete or insert) happens. Triggers can call stored procedures, which are functions written in SQL and stored on the server.

You can build a SQL database with sufficient RI and other rules so that it's virtually impossible for any application (or any individual executing SQL statements by hand) to corrupt that database.

Compatibility

Although the major database vendors each have their own flavor of SQL, there is enough adherence to the 1992 ANSI SQL standard that you can readily port most applications from one server to the next. And because SQL is a relatively standardized language, there are many tools and utilities available, for everything from database design to syntax checking to reporting. One place to look for SQL tools is the CNet www.download.com site.

Ease of administration

As I indicated earlier, SQL isn't just for querying data. You also use SQL to create and alter databases, tables, and indexes. It's easy to add a field to or remove a field from an existing table, or to change a field's data type or default value. In a TPS file, changing the table definition doesn't change the physical data; you still have to create a conversion program, or use the data dictionary's table conversion feature.

You can also easily do mass updates in SQL by applying UPDATE or DELETE statements to a selected set of records. While mass updates are inherently dangerous (you can easily wipe out an entire table), they're also amazingly useful.

Scalability

Applications are not only a lot bigger than they used to be, but they typically deal with a lot more data. TPS files can store a lot more data than the old Clarion DAT files, but in general flat file databases hit storage limits and performance walls a lot sooner than SQL databases. The history of each kind of database suggests this is likely to happen: Flat file databases derive from small, single user systems, and SQL databases started out on the big iron. You wouldn't want to store a terabyte of data in a TPS file, even if it were possible (it isn't - the maximum size of a TPS file is two gigabytes).

SQL databases are designed to hold massive amounts of data, and to work with that data efficiently. SQL server performance scales with the hardware, while the same isn't generally true of flat file databases. At the same time, SQL servers are invading the personal computer data space, and some of the more popular SQL databases are even available for handheld computers.

Clarion and SQL

Clarion's support for SQL databases has improved considerably in recent years. Prior to ABC, your only real option was the Cowboy Computing Solutions SQL templates (<http://www.csccowboy.com/products.htm>) . ABC provides acceptable SQL features and performance, although for serious work you should still take a look at the CCS SQL templates at www.icetips.com. (Clarion 6.x also has ADO templates although these do not appear to be in wide use among Clarion developers just yet.)

If you already have an SQL database, and you have a suitable SQL or ODBC driver for that database, all you probably need to do is import the tables into a dictionary and you can start creating your application in the usual manner.

If you're porting a TPS database to SQL, I strongly suggest you read "How To Convert Your Database To SQL," p. 175, and "Converting TPS To MS-SQL," p. 183. There are a number of important data type and table design requirements in SQL that don't exist in TPS files, and there's a good chance that you'll need to make at least a few minor changes. If you don't yet have a SQL database to play with, take a look at Tom Hebenstreit's suggestions for getting into SQL on the cheap in "Getting Into SQL On The Cheap," p. 167. One free SQL database increasingly popular with Clarion developers is MySQL (<http://www.mysql.com>), and you can read more about this database in the "Open Source SQL" section of this book, page 219.

Choosing a driver

To use Clarion and SQL, you need two things: a SQL database, and a driver that lets Clarion talk to that database.

Clarion 5.x Professional ships with the ODBC and MS SQL drivers; Clarion 5.x Enterprise adds the SQL Anywhere (a.k.a. Sybase) and Oracle Accelerator drivers. If you only have the Professional version you can still use Oracle, SQL Anywhere, and many other SQL databases using the ODBC driver.

What you won't get with ODBC in Clarion 5.x is the ability to do multi-file imports from the SQL database, and you won't be able to use the Clarion Database Synchronizer to transfer changes from the database to your dictionary and vice versa. For now, I'll assume that you're using the ODBC driver (which is still a fine solution) to talk to your SQL database.

Choosing a SQL database

It's possible that your choice of SQL database has already been made for you. If this is the case, chances are you're looking at one of the commercial SQL servers, such as Microsoft SQL Server, Oracle, or SQL Anywhere (Sybase). There are also an increasing number of good, free, open source servers available, including MySQL, PostgreSQL, and Interbase/Firebird.

Each SQL server has its own strengths and weaknesses. For many developers, commercial database licensing costs are a serious drawback, but commercial products may have features not available elsewhere. And although all SQL servers share a common core of SQL statements, most vendors have enhanced SQL with database-specific commands. If your applications begin to make significant use of server-specific functionality, you may find it difficult or even impossible to port your application to another server.

For example, I use MySQL as the core database behind *Clarion Magazine*, in large part because of MySQL's reputation for speed and reliability. But MySQL's support for transactions is relatively new. If I needed transactions I might think seriously about PostgreSQL (a.k.a. Postgres). Postgres is considered fairly reliable, but doesn't deliver data as fast as MySQL does, even if you disable immediate flushing of data to disk. On the other hand, Postgres has sub-selects (you can use one SELECT statement as part of another SELECT statement) which I'd dearly love to have in MySQL. I can work around the missing sub-select capability with temporary tables, but that's a bit awkward. On the whole, I think MySQL is a better choice for this particular application, although I've read that Postgres may actually scale better...

You get the idea. Throw in a few more key features like triggers, stored procedures, the ability to store programming objects in the database, and you have a whole lot to consider when choosing a SQL server. Fortunately, there's a lot of competition for SQL customers, so you can usually get a low-cost or free trial version of just about any database server you want to take a look at.

Installing the server

Whichever server you decide on, your next job is to install the server software. In a production environment you'll almost always install the database server software on a dedicated computer (also referred to as a database server). But for development work you can often install a personal version of the server on your own computer. Although you won't get a realistic assessment of the performance your application can expect in a networked installation (depending on the computing and network hardware involved, an application may run faster or slower using a personal server installation than using a networked database server), you'll still be able to test all of your application's functionality.

Creating a database

You might think that one of the advantages of running a personal SQL server on your own machine is that you won't mess up somebody else's database. In fact, SQL servers let you create multiple databases for various purposes, so it's easy to isolate your test or development data from other data on the server.

A SQL database is a bit like a directory of TPS files. Just as you can have same-named TPS files in different directories, so you can have same-named tables in different databases. How the SQL server actually stores the databases is entirely dependent on the server implementation, but that's a detail you don't need to concern yourself with. In SQL, you deal with table definitions and data; you let the server worry about where and how it actually stores that data.

A SQL server can contain a large number of databases, and many tables within each database. The SQL server can also actively manage all of this data to ensure efficiency and integrity. Larger organizations, with large databases, typically employ one or more Database Administrators (DBAs) to ensure that nothing bad happens to all that important data. To make the DBA's job a little easier, most SQL servers allow the DBA to set various permissions on different table operations. At the lowest level, a particular user may only be allowed to view a particular column of data in a particular table in a particular database. At the highest level, a user (really now a DBA) can create and delete entire databases.

If you're working in a larger organization, and the departmental SQL server is your only option, the best you can hope for is that the DBA will give you your own database in which you can create and delete tables to your heart's content. If you have your own SQL server, then you can do anything you want, can't you? Ah, the power.

There are two ways to create a database. One is to execute a `CREATE DATABASE` statement. The other is to use a database administrator utility which provides a somewhat friendlier interface, but which will ultimately execute that same `CREATE DATABASE` statement.

Creating a database with SQL

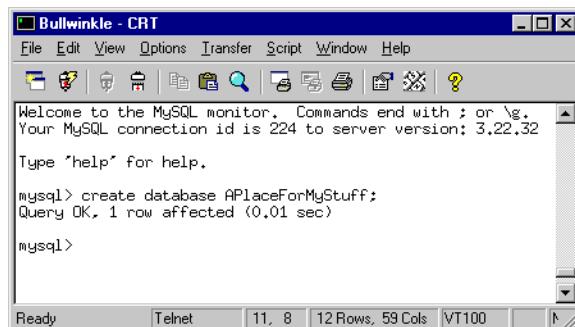
To create a database called Test, you use a SQL statement like the following:

```
CREATE DATABASE Test;
```

Notice that the statement ends with a semicolon, which in SQL is the line terminator. You can spread a SQL statement over multiple lines by pressing the Enter key, which in the case of longer SQL statements often greatly improves readability. The SQL server won't attempt to execute the statement until it encounters the terminating semicolon.

Creating a database is simple, isn't it? But where do you type this command? SQL servers generally come with a number of bundled utilities, one of which will be a SQL interpreter, sometimes called a SQL query tool, or SQL query analyzer. You can use this tool to execute any valid SQL statement (at least one you have permission to execute). For example, I have MySQL running on a Linux box on my office network. To interactively execute SQL statements on that server, I telnet to the Linux box and type mysql at the shell prompt to run the mysql client program. Figure 1 shows how I use that program to create a database called APlaceForMyStuff.

Figure 1: Creating a database with a direct SQL statement

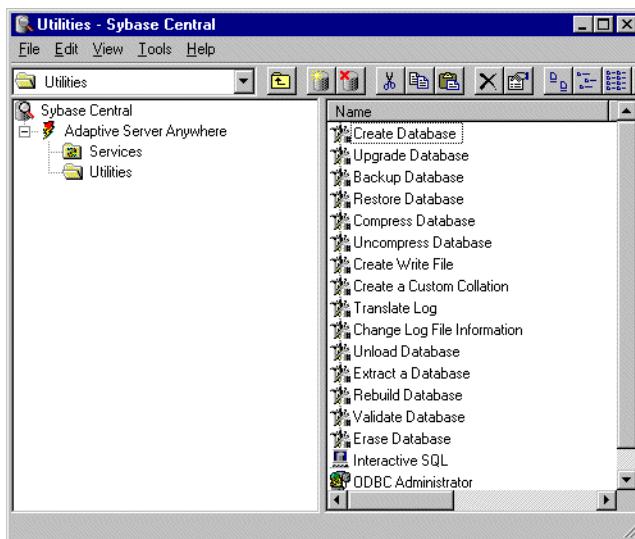


The screenshot shows a window titled "Bullwinkle - CRT". The menu bar includes File, Edit, View, Options, Transfer, Script, Window, and Help. Below the menu is a toolbar with various icons. The main window displays the MySQL monitor. It starts with a welcome message: "Welcome to the MySQL monitor. Commands end with ; or \g. Your MySQL connection id is 224 to server version: 3,22,32". It then shows the command "Type 'help' for help." followed by the execution of "mysql> create database APlaceForMyStuff;". The response is "Query OK, 1 row affected (0.01 sec)". At the bottom of the window, there is a status bar with "Ready", "Telnet", "11, 8", "12 Rows, 59 Cols", and "VT100".

Creating a database with a database administration utility

It's probably more likely that you'll create your databases using an administration utility. Figure 2 shows Sybase Central, the database administration program that ships with the SQL Anywhere database.

Figure 2: The Sybase Central database administrator



As you can see from Figure 2, Sybase Central gives you an easier way to perform common database administration tasks. If, for instance, you choose the **Create Database** utility from the right-hand pane, you'll get a Create Database wizard. And the Sybase Central Create Database wizard presents a lot of options, such as transaction logging, encryption, page sizes, case sensitivity, and much more. But all of these options are just another way of creating a single SQL statement. Listing 1 is the full syntax for SQL Anywhere's CREATE DATABASE command:

Listing 1. The CREATE DATABASE command in SQL Anywhere v.6

```
CREATE DATABASE db-file-name
...
...
[ [ TRANSACTION ] LOG OFF | 
  [ TRANSACTION ] LOG ON [ log-file-name ]
  [ MIRROR mirror-file-name ]
]
...
[ CASE { RESPECT | IGNORE } ]
[ PAGE SIZE page-size ]
[ COLLATION collation-label ]
```

```
... [ ENCRYPTED { ON | OFF } ]
... [ BLANK PADDING { ON | OFF } ]
... [ ASE [ COMPATIBLE ] ]
... [ JAVA { ON | OFF } ]
... [ JCONNECT { ON | OFF } ]
]
```

Each step in the Sybase Create Database wizard corresponds (more or less) to an option on the CREATE DATABASE statement.

By comparison, the full syntax for MySQL's CREATE DATABASE command is considerably less complicated:

```
CREATE DATABASE [IF NOT EXISTS] db_name
```

Modifying database space

If you locate the actual physical file or files that hold your SQL data, you may be surprised to see that there's no correlation between the SQL data file and the tables your database contains. (One notable exception is MySQL, which by default stores each database in its own directory, and each table as a file in that directory.) Since SQL servers are responding to SQL requests, and do not allow the application to read/write the data directly, the server can store the data on disk in whatever way is most convenient.

If your database is small, there's a good chance your SQL server will store the entire database as a single physical file. Larger databases may be spread over several physical files, which can even be located on other computers. One of the DBA's many responsibilities is to manage these different files (often called dataspaces or tablespaces) to ensure that there's enough space available, and that performance remains good. In most cases a SQL server can be set to grow its data files as needed, but there may be predefined limits on how large a given file can get, or you may need to pre-allocate new dataspaces to accommodate anticipated growth.

Database log files

SQL databases typically let you create log files which record all changes to the database. This can be particularly useful in a development environment, when you want to keep track of everything your application actually does with the data. You may also be able to use the log file to re-apply changes to a database which you've had to restore from a backup.

Backing up a database

Just as is the case with TPS files, when backing up a live SQL database you run the risk of creating an inconsistent data copy. It's best to use the backup utility that comes with your database, as this program (hopefully) knows how to get a clean copy of the data. For instance, the backup program may wait for all transactions currently in process to commit before taking a snapshot of the database.

Database replication

I regularly see messages in the SV newsgroups from developers who need to manage one database across multiple physical locations, such as branch offices. This is a common enough requirement in the business world that most SQL servers offer some form of replication.

In a database replication scheme, one SQL server manages the central database, and additional SQL servers manage local copies of that database (or portions thereof). These servers communicate with the central SQL server on a regular basis, exchanging information as necessary.

There are various scenarios for database replication, from simple duplication of data to read-only clients to allowing all servers to fully participate in updating data. As you can imagine, database replication can introduce a whole new set of design problems. Still, it's better to let the server manage this kind of problem than to embody the necessary logic in the user's application.

Choosing a server

The single most important choice you'll make in going to SQL is your choice of SQL server. Although all SQL servers share a common set of SQL statements, vendors typically add their own proprietary enhancements. You should look at a variety of SQL servers before making your choice.

Once you've decided on, obtained, and installed a SQL server, your next task is to create at least one database. You can do this with a simple SQL statement, and you'll probably also have the option of using a utility to create the database.

Creating tables

As you'll recall from earlier in this chapter, the SQL language isn't just for retrieving and updating data; you also use SQL to create databases, tables and indexes. Here's a simple SQL statement that creates a table with a few fields:

```
CREATE TABLE ArticleGroups
(
    ArticleGroupID    INT AUTO_INCREMENT NOT NULL,
    Title             VARCHAR(255),
    PublicationDate  DATE,
    Value             TINYINT,
    LastModified      TIMESTAMP
);
```

For the sake of readability, I've put each field (or to use the SQL term, column) definition on its own line. This is common practice as it makes the statement more readable.

You will have to get used to some different data types when you switch to SQL. For a complete listing of available data types, start with the ODBC driver help, ODBC Data Types section in the Clarion online help. You'll see a chart mapping ODBC data types to Clarion data types. Some of the most common type comparisons are as follows:

ODBC Data Type	Clarion Data Type
CHAR (fixed length string)	STRING
VARCHAR (variable length string)	CSTRING
DECIMAL	PDECIMAL
TINYINT	BYTE
SMALLINT	SHORT
INTEGER	LONG
DATE	DATE
TIME	TIME
DATETIME	GROUP (see below)
TIMESTAMP	GROUP (see below)

Conversion between data types isn't always straightforward, and there are often alternatives. For instance, you can represent an ODBC fixed length CHAR string with either a Clarion STRING or a CSTRING, but if you use a CSTRING you won't be able to use Clarion to create that table. Or perhaps your design calls for a BYTE field but your SQL

database doesn't have a TINYINT type. In that case you'll need to use a SMALLINT or other data type. You can also represent many SQL data types as a Clarion STRING.

Dates and times

One of the more significant differences in data types between TPS and SQL tables has to do with date and time handling. If you're a long-time Clarion developer, then you're familiar with storing dates and times as LONG values, and displaying these values with pictures. A Clarion standard date is the number of days since December 28, 1800, so March 13, 2001 is 73124.

If you're viewing a TPS (or DAT) file with a Clarion program or utility, you can specify an appropriate date picture to translate the date number into something meaningful. But if you're using a program that doesn't understand a Clarion standard date (say you're browsing a TPS table in Excel, using the ODBC driver), then you're just going to have a bunch of numbers to look at. Quick, what day is 73087? And Clarion standard time is no better, since it's stored as the number of hundredths of a second since midnight, plus one.

In a mixed environment, you need to store dates and times in a format non-Clarion programs can understand. That's the reason for the DATE and TIME types. Both of these are four byte variables. DATE uses two bytes to store the year (1-9999), one byte for the month (1-12), and one byte for the day of the month (1-31). Similarly, TIME uses one byte each for the hour (0-23), minute (0-59), second (0-59), and hundredths of a second (0-99). You don't have to do anything special in your code to use a DATE or TIME instead of a LONG; in fact, if you're working with TPS tables you can change all of your LONG fields to DATE or TIME fields, and just convert the files in the dictionary browser. Now you're one step closer to being able to switch over from TPS to SQL.

Not all SQL databases support individual DATE and TIME data types, but most have a DATETIME or TIMESTAMP data type, which is a combination of a DATE and a TIME. These two data types have different purposes. A DATETIME is usually set to a previously known value, while a TIMESTAMP is automatically set to the date/time the record was last modified. In some (but not all) SQL databases TIMESTAMP is guaranteed to be a unique identifier, although this requirement is not part of the ANSI SQL specification.

Clarion doesn't have a data type to correspond to the combination of date and time, but it's easy to fake. When you import a SQL table definition which contains such a field, you'll see a group structure in the table definition that looks something like this:

```
TimeStampField      STRING(8), NAME('TimeStampField')
TimeStampGroup     GROUP, OVER(TimeStampField)
TimeStampDate      DATE
TimeStampTime      TIME
END
```

In this example the date/time field in the SQL table is called `TimeStampField`, and it's declared as a `STRING(8)`. The fact that `TimeStampField` is a `STRING` isn't important. What is critical is that this field be eight bytes long, because that's the length of a date/time field in SQL. The second part of the declaration is a `GROUP` with an `OVER` attribute. `OVER` lets you assign one variable to the memory space occupied by another variable. When you request data from this table, and the driver returns the contents of `TimeStampField`, you can refer to those contents eight bytes at a time using the `TimeStampField` label or the `TimeStampGroup` label. In either case you'll get gibberish. But if you use `TimeStampDate` or `TimeStampTime`, you'll be working with just the four bytes corresponding to that half of the variable. The `OVERed GROUP` is just a trick to extract the individual `DATE` and `TIME` values.

If your SQL server doesn't support individual `DATE` or `TIME` data types, and all you need is one or the other, you'll have to use a `DATETIME` (which, confusingly, is sometimes just called `DATE`) and resign yourself to wasting the four unused bytes.

Primary keys

Take another look at that `CREATE TABLE` statement from the beginning of the chapter:

```
CREATE TABLE ArticleGroups
(
    ArticleGroupID    INT AUTO INCREMENT NOT NULL,
    Title             VARCHAR(255),
    PublicationDate  DATE,
    Value             TINYINT,
    LastModified      TIMESTAMP
);
```

Do you notice anything missing? There are no keys in this table definition. And this is a critical omission, because every table should have at least one key, called a primary key.

Any database system needs a way of uniquely identifying individual records within the database. In a flat file database, this record pointer is often the byte offset in the physical data file of the start of that record. That's the case with TPS tables; with Clarion DAT files the pointer (as returned by the `POINTER()` function) is the actual record number, which is why in DAT files you can use code like `GET(Control, 2)` to get the second record in a file.

SQL databases don't necessarily have a built-in record identifier, which is why you have to be sure to provide a way of uniquely identifying each record. Here I've modified the `ArticleGroups` table create script to include a primary key on the `ArticleGroupID` field.

```
CREATE TABLE ArticleGroups
```

```
(  
    ArticleGroupID      INT AUTO_INCREMENT NOT NULL,  
    Title               VARCHAR(255),  
    PublicationDate    DATE,  
    Value               TINYINT,  
    LastModified        TIMESTAMP,  
    PRIMARY KEY (ArticleGroupID)  
) ;
```

Also note that in this definition (which is from a MySQL database) I've added an `AUTO_INCREMENT` attribute and a `NOT NULL` attribute. Every time I add a record to `ArticleGroups`, the server will assign a unique, automatically incremented value to `ArticleGroupID`. The `NOT NULL` attribute means that `ArticleGroupID` must have a value; if I didn't use the `AUTO_INCREMENT` feature, I'd have to supply my own value for `ArticleGroupID`.

Keys and indexes

Clarion developers have had to deal with some confusing terminology in recent years. Is that an object or a class? A table or a file? And now, is that a key or an index?

In SQL parlance, a key is a logical definition, not a physical thing. Keys are how you relate two tables. In the *Clarion Magazine* database I have a table for groups of articles (`ArticleGroups`), a table for articles (`Articles`), and a linking table that manages the many-to-many relationship between `ArticleGroups` and `Articles`. This table is called `ArtGrpLink`, and its create statement looks like this:

```
CREATE TABLE ArtGrpLink  
(  
    ArtGrpLinkID      INT      AUTO_INCREMENT NOT NULL,  
    ArticleGroupID    INT      NOT NULL,  
    ArticleID         INT      NOT NULL,  
    LastModified      TIMESTAMP,  
    PRIMARY KEY (ArtGrpLinkID)  
) ;
```

In `ArtGrpLink`, the `ArticleGroupID` field's only purpose is to link a record in that table back to the `Articles` table. `ArticleGroupID` is what's called a foreign key - it's a key because it is used to link to another table, and it's foreign because the information it links to exists outside the current table. *But there is no separate physical structure that defines the key, as there is in a TPS table.*

When you create a TPS table, and you want to sort that file in a particular way, you define...a key. There is something comparable to a TPS key in SQL databases but a) it's called an index, not a key, and b) it's optional.

This may take a bit of thought. A key, in SQL, is just one or more fields that link to another table. When you define a key (I'll give an example shortly) you tell the server that a relationship exists between two tables. Optionally you may also tell the server that there are certain constraints on this relationship. Perhaps when the parent record is deleted you want to delete all the child records automatically, or you want to prevent the parent from being deleted if child records exist.

Is this starting to sound familiar? In SQL, defining keys is akin to defining relationships in the dictionary editor. On one side of a relationship you have a foreign key, and on the other side you have a primary key. Here's an example of an `employee` table with a foreign key (`dept_id`) that references the `department` table's primary key (also labeled `dept_id`).

```
create table employee
(
    emp_id          integer      not null,
    manager_id      integer      ,
    emp_fname       char(20)     not null,
    emp_lname       char(20)     not null,
    dept_id         integer      not null,
    primary key (emp_id),
    foreign key ky_dept_id (dept_id)
        references department (dept_id)
        on update restrict
        on delete restrict
);
```

This foreign key declaration says that `employee.dept_id` matches `department.dept_id`, and that the server should not allow anyone to delete a department record or change the department record's primary key (`dept_id`) if there's a related `employee` record.

This particular foreign key example only restricts updates and deletes, but you can also have foreign keys with no constraint, a cascade constraint (delete or update all records with a matching foreign key), or a constraint that doesn't change the related record but clears the foreign key or resets it to a default value if the primary key record is deleted.

SQL indexes

Just as SQL keys correspond to Clarion data dictionary relationships, so do SQL indexes correspond to TPS keys. In a TPS table a key is a physical construct that makes it possible to sort records by field contents. In a SQL server, an index is a physical construct that lets the server sort records efficiently.

Technically, TPS keys and SQL indexes are both optional, since Clarion applications use the View engine to collect data, and the View engine lets you sort on arbitrarily chosen

fields. If you don't have a key/index defined, performance will suffer to some degree, probably more with the TPS table than with the SQL table. And most of us who use TPS files wouldn't think of creating a TPS file without at least a few keys.

When it comes to SQL, however, the whole subject of creating indexes (which, of course, correspond to TPS keys) becomes a little more nebulous. That's because an application that wants to retrieve some data from a SQL database simply sends SQL statements like:

```
SELECT Name,Address,City,State FROM Names  
ORDER BY State;
```

The ORDER BY clause tells the server what order to use when sorting the resulting data, but there is no instruction on which index, if any, the server should examine for that order. Now, with some SQL databases you can give the server hints in situations where several indexes could be used, and you know one to be more efficient than another, but that's beyond the scope of this chapter. For the most part, how the server does its job is up to the server.

This abstraction of the query language from the server implementation has important consequences for the developer. With TPS files, the Clarion runtime library ensures that the definition you have for your file exactly matches the file itself; in SQL, all that matters is that the fields you have defined in the table have corresponding fields in the table on the server. Your definition may have only one field, while the actual table may have dozens.

You also don't need to define any indexes for your SQL tables in the data dictionary (okay, the dictionary editor calls these keys - life *can* be confusing). You can create a browse that displays in record order and add whatever additional sort fields you like, and if the server has corresponding indexes, you'll get good performance.

You probably will want to include index (key) definitions in your SQL tables in the dictionary, however. For one thing, you'll need indexes/keys to define relationships between tables, and for another it's a good thing if your data dictionary fairly closely models the actual database. But there's no requirement for your indexes to match the back end's indexes; you'll never get an invalid record declaration because of an index mismatch. You might get some abysmal performance because your code assumes an index that doesn't exist, but that's another story.

Keys and indexes

If you've never worked with SQL tables before, you'll first encounter some unfamiliar data types. Most of these have close Clarion equivalents, the exception being date/time combinations, which require a special GROUP structure. You'll also need to ensure that all

your tables have a primary key, which is a field or combination of fields guaranteed to be unique for each record.

For the most part, SQL tables aren't that different from TPS tables, at least until you come to the subject of keys and indexes. It can be tricky at first remembering that a foreign key defines a relationship between SQL tables, and an index is a physical construct which helps the server locate records more quickly. As well, because your SQL application works with data by using SQL statements, not by directly accessing data files, there can be significant discrepancies between the table and index definitions in your applications and those on the server, and your application can still work successfully.

I don't care if it's SQL

I've been asked on a few occasions just how SQL development differs from non-SQL (typically TPS) development. The answer to this question is "it depends." You can choose a development style anywhere on the continuum from "almost identical to TPS" to "radically different from TPS."

It's quite possible to create a Clarion application that can run on either a TPS database or a SQL database, and the only thing you have to change is the driver. You will need to stick with data types common to all the drivers you plan to use, but other than that you don't need to make any special accommodation. You don't need to think about your development in a different way, except to the extent that you need to learn how to create or maintain a SQL database. And you need to make sure that each of your tables has a primary key.

I suspect that a lot of Clarion developers who do SQL start off with this approach. Perhaps they're looking for better network performance, or maybe SQL is one of the client's requirements. In any case, the point is that you can treat SQL tables the same way you treat TPS tables (or files, if you prefer that terminology). In this situation your application assumes no intelligence other than its own is at work manipulating the database, and the SQL server functions simply as a repository for data. You ask for data, you get it. You update data, it's updated. You delete, it's gone. A SQL server used this way doesn't take any additional action based on what you ask it to do.

Your application will automatically take some minimal advantage of any SQL database server's special capabilities, primarily when you're dealing with a browse that uses related tables. In older versions of Clarion browses read files directly, using the file driver; in Clarion ABC all such file access is handled by a Clarion `VIEW` structure, which is a sort of logical table which can contain related tables. Here's an example of a `VIEW` structure that combines three tables using a `JOIN` to display authors and their articles:

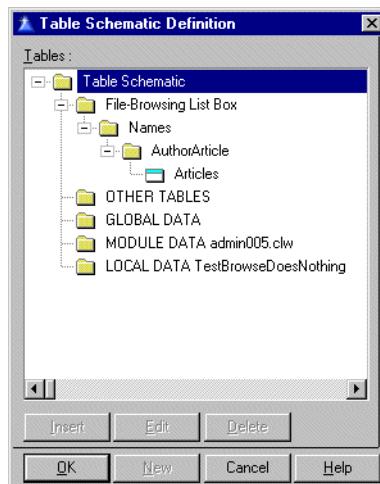
```
BRW1::View:Browse VIEW(Names)
    PROJECT(nam:LastName)
```

General SQL

```
PROJECT (nam:FirstName)
PROJECT (nam:NameID)
JOIN (aat:AuthorID,nam:NameID)
    PROJECT (aat:ArticleID)
        JOIN (Art:PRIMARY,aat:ArticleID)
            PROJECT (Art:Title)
            PROJECT (Art:ArticleID)
        END
    END
END
```

The primary table in this Clarion VIEW is the Names table. In the file schematic, this is the first table listed in the browse control, as shown in Figure 1. There are two additional tables in the VIEW: AuthorArticle is a linking table which manages a many-to-many relationship between Names and Articles.

Figure 3: The browse file schematic



If you use this VIEW with a flat-file database, Clarion will retrieve all of fields in each table record, even though only a few of these fields are actually listed in the VIEW. That means you get a lot more network traffic than you really need, and performance will suffer. If you use a SQL database, Clarion will generate a SELECT statement instead, and that statement will only retrieve the required fields. Here's a SELECT statement that corresponds to the above VIEW structure (I created this using the \c55\bin\trace.exe utility):

```
SELECT A.NameID, A.FirstName, A.LastName,
A.Company, A.Country, A.Email, A.UserID,
B.AuthorArticleID, B.ArticleID,
C.ArticleID, C.Title
FROM Names A
LEFT OUTER JOIN AuthorArticle B
ON A.NameID= B.AuthorID
```

```
LEFT OUTER JOIN Articles C
    ON B.ArticleID= C.ArticleID
ORDER BY B.AuthorID ASC, B.ArticleID ASC,
    C.ArticleID ASC
```

Although there are numerous fields in the Names table, only seven of these fields are named in the SELECT statement. You're probably wondering why seven, since just three are listed in the VIEW. As near as I can tell, ABC adds these fields automatically because they're key components. At least seven is better than 38, which is how many fields there really are in Names. Of course, when you bring up an update form, ABC will retrieve all of the fields in that row.

Note: In my tests with MySQL, ABC reports and processes, unlike browses, automatically retrieved *all* fields in the table(s), thereby removing the network performance benefit enjoyed by ABC browses running on SQL data.

In SQL, tables can be associated with a JOIN statement, such as this:

```
Names A LEFT OUTER JOIN AuthorArticle B
    ON A.NameID= B.AuthorID
```

There are several different kinds of joins. In a LEFT OUTER JOIN the SQL server will look for records for the left-side table, and find matching records on the right side table. If there are no matching records on the right side, the server supplies NULL values for the right side fields. This is the kind of join most Clarion programmers use, whether they realize it or not.

Notice that the Names table is defined in the SELECT statement as Names A, not just Names. The A is an alias for the Names table. Since you can have identical field names in different tables, you often need to prefix the field with the table name, as in Names.NameID. But that can lead to a lot of typing, so SQL allows the use of an alias. In this case, A.NameID is the same as Names.NameID. The Clarion view engine assigns these aliases alphabetically, beginning with A.

Finally, the JOIN has to specify which are the linking fields. The Clarion view engine uses the ON syntax:

```
ON A.NameID= B.AuthorID
```

All of the above code comes from a straight ABC application that would work with SQL or TPS tables. The only difference is the file driver. So even though you don't make any special allowances for SQL, you can still get some of the speed and performance benefits of SQL.

Tuning for SQL

Although stock ABC SQL applications work, there's a whole world of functionality out there for SQL developers. Typical server features include:

- Mass updates - why write a process to do something, when a single SQL statement will accomplish the same result?
- Server-side autoincrementing of keys
- Enforcing referential integrity
- Stored procedures - SQL code which can be called at any time
- Triggers - ability to execute a stored procedure when a particular event happens

I'll take a brief look at each of these areas, and point out some of the issues for Clarion developers.

Mass updates

Clarion developers are used to applying updates to one record at a time. With SQL, you can update large numbers of records with a single statement. For instance, let's say I've been inconsistent in storing country information in my Names table. In some cases, the country value for the United States of America is 'USA', in others 'US'. To change all instances of 'US' to 'USA' I can execute the following statement using PROP:SQL:

```
UPDATE Names SET Country='USA' WHERE Country='US';
```

This kind of capability doesn't necessarily have a bearing on how you design your applications, except that you can probably dispense with some of your own client-side code. Of course, you'd never allow this kind of inconsistency to appear in your data in the first place, right?

Server-side autoincrementing

Good database design requires you to have a unique identifier for each row in a table, and in most cases you'll accomplish this using an autoincrement key. Traditionally, Clarion applications autoincrement by retrieving the record in the table with the highest key value, incrementing that value by one, inserting a record with the new value (to reserve that auto-incremented number), and changing the current action from an insert to a change (even

though the form still appears to be inserting a new record). With a SQL database, you have the option of letting the server do the auto-incrementing, which is generally faster and more reliable. But this is not as straightforward as it may seem.

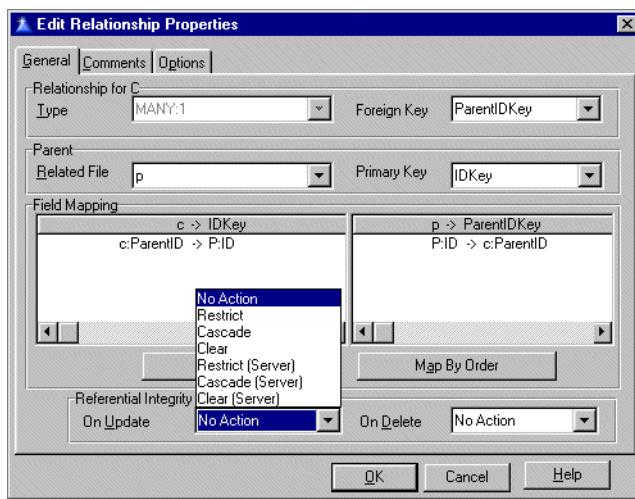
When you're doing a simple insert into a table, everything is fine - you may need to supply a NULL value for the primary key field, but the server will take care of the rest. The difficulty arises when you try to add related (child) records using the parent's update form. If you've just inserted the parent record, you won't have a value for the primary key field. That value exists, but your form has only inserted the record; it hasn't retrieved that record to find out the field value.

There are various ways around this problem. For MS SQL Server, Jim Kane has written some code to retrieve the @@identity variable, which contains the value of the last autoincrement identifier for the current connection. This way you can assign the correct parent id to the child record.

Enforcing referential integrity

I can't claim much experience with server-side referential integrity (RI), because most of my SQL work is with MySQL, which doesn't provide this capability. Most databases do let you set various update and delete check constraints in much the same way as you're probably accustomed to setting these constraints in the Clarion dictionary editor. If you decide to handle RI on the server, you should select the appropriate server side constraint in the Clarion dictionary editor, as shown in Figure 4.

Figure 4: Choosing a server-side RI constraint



Setting server-side constraints in the dictionary doesn't create any server-side code; this is just a way of documenting that the server will handle the RI issues.

Stored procedures and triggers

Closely related to RI issues is the use of stored procedures and triggers. SQL is a query language, but it's also a data definition language, and in many ways a full-fledged programming language. With most SQL servers you can store SQL code on the server, as a procedure, and call that code from your Clarion applications with `PROP:SQL`.

A trigger is similar to an RI constraint in that a particular event (such as an update or delete) triggers an action. In fact, you can implement RI constraints as triggers, if you like. The trigger can contain all the necessary SQL code, or it can call stored procedures instead of, or in addition to, its own SQL code.

Stored procedures and triggers have the most potential to radically alter your approach to database development. Much of the code in your application represents business rules, or standard approaches to handling certain kinds of data. The more of those business rules you move to the server, the simpler the client program becomes, and the safer it is to let other applications work with the data, since the server enforces the business rules no matter which client updates the database. You can make the database itself relatively bulletproof, with enough effort.

Moving all this code to the server can present disadvantages as well as advantages. You'll need to learn how to express your Clarion code as SQL code, and that takes some effort. If you move your application to another SQL platform, chances are you'll also have to rewrite some of your SQL code since, as Mike Gorman points out, there really is no firm and fast SQL standard. And you'll be creating a much more complex database which requires a greater level of understanding and, perhaps, administration.

Which way do I go?

The benefit you get from moving from a flat file database (like TPS files) to a SQL database is proportional to the degree to which you use the SQL server's capabilities. If you simply change drivers (assuming your datatypes are compatible with the SQL server), and your browses typically do not retrieve most or all of the fields in a table, then you should see better network performance. You may see better raw data access speed on the server as well, but I've never benchmarked raw TPS speed against any SQL server, so I can only guess that there will be some wide variation, depending on which SQL server you use.

Server-side processing, like check constraints for referential integrity and autoincrementing, further reduces network traffic, although auto-incrementing can cause some additional headaches, as I described earlier.

If you go all out and implement triggers and stored procedures, you can reduce network traffic by another notch or two. Although minimizing network traffic is an important goal for most developers, do keep in mind that you won't improve performance if the server doesn't have the processor speed and/or bandwidth to keep up with requests. You need to strike a balance between what the client computers are capable of, what the server can do, and how fast you can get data between the two. All other things being equal, however, there are significant benefits to moving business logic from the client machines to the SQL server.

Resources

- Whitemarsh SQL papers (<http://www.wiscorp.com/SQLStandards.html>)
- SQL.ORG tutorials (http://www.sql.org/online_resources.html)
- CCS SQL templates (<http://www.ccscowboy.com/>)

General SQL

GETTING INTO SQL ON THE CHEAP

by Tom Hebenstreit

Editor's Note: Prices and availability of the products described here may have changed.

Data grows. And grows.

And then it grows some more.

Sooner or later, your programs will outgrow the capabilities of the flat-file database drivers. When that happens, the solution is normally a move up to a full-fledged SQL-based Relational Database Management System (RDBMS).

Most RDBMS provide powerful features that your programs can take advantage of to both ensure greater data integrity and enhanced performance. Just as importantly, a well-designed SQL based application can keep scaling upwards with little or no further effort on your part (just throw some more hardware at it, or maybe add a parallel processing or clustering option to the RDBMS).

Even if you don't have a need yet and simply want to prepare yourself for future growth, SQL and RDBMS are the two terms you will want to keep in mind.

The problem is, Relational Database Management Systems are not only powerful, they can be expensive. Make that *very* expensive.

So how do you start getting your feet wet with SQL and RDBMS without spending a fortune?

Glad you asked.

That will be the focus of today's column - how to acquire a SQL based RDBMS at prices ranging from free (my perennial favorite) to a few hundred dollars (which is low cost in RDBMS terms).

Please note that the list is not totally inclusive, and I apologize if I've forgotten anyone's favorite RDBMS. Also, don't take the prices mentioned here as gospel; use them as the starting point for your own explorations. Prices and packages change rapidly in the software world, and who knows, you could easily find an even better deal with a bit of research.

Microsoft SQL Server 7

SQL Server 7 is Microsoft's current flagship RDBMS. A huge improvement over 6.5, it has won over a lot of Clarion developers with new features and enhanced reliability. More importantly, it has won over quite a few IT departments, so whether you like it or not, you will quite likely be faced at some point with either the desire or the need to get familiar with it.

Paradoxically, the cheapest and most expensive way to get SQL Server 7 is via a Microsoft Developers Network (MSDN) Universal subscription (about US\$2000 annually). If your company has one of these, just grab the SQL Server 7 disks and you are good to go. You can also get some good deals on it as part of the Back Office or Back Office Small Business Server packages.

On a less harrowing level, you can download a fully functional evaluation copy from Microsoft for absolutely free. This, of course, assumes that you have a big pipe and some time on your hands, as there are twenty-two separate files totaling 217 megabytes that need to be downloaded. Oh, yeah - you'll need about a gigabyte of free space for the process of recombining the files into the installable package, etc.

A more practical alternative is to order the evaluation CD for US\$9.95. Last time I got one, it was a kit with four CDs including SQL Server 7, some info and training CDs and a bonus trial copy of Windows NT Server. No guarantee that they still give you all of that stuff, but I was impressed with the package back then.

By the way, unlike most companies that have 30-day limits on evaluation copies, Microsoft gives you a full 120 days so that you can really delve into the product.

If you own Visual Studio, the best deal is to get what used to be called the Visual Studio Plus Pack – it includes both a test SQL Server 7 license (not time limited) *and* MSDE (see the next section), along with the Windows 2000 Developers Readiness Kit and some other goodies. It may be hard to track this down now, as they rename it quite often as various bits are added and removed. When I got it, I think it was around US\$15.

Resources

- MSDN: <http://msdn.microsoft.com>
- SQL Server Home Page: <http://www.microsoft.com/sql>

Microsoft Data Engine (MSDE)

What is MSDE? In a nutshell, it is SQL Server 7 without the administration tools. Designed for use in less demanding applications, it has a 2 gigabyte limit on database size and is not, according to Microsoft, quite as capable in a multi-user environment (after all, they don't want to cannibalize SQL Server sales.)

The bottom line, though, is that with MSDE installed on your machine you can develop fully SQL 7 compatible applications on a single machine under Windows 9x. Since it really *is* SQL 7, everything (including RI, triggers, stored procedures, etc.) scales right over to a full-blown SQL Server system.

How do you get it? Well, if you have Office 2000, look in the \Sql\x86\Setup folder on the first disk. Note that MSDE is not installed by default - the Jet engine is (yuk!).

If you own any Visual Studio 6 component (Visual C++, Visual Basic, Visual Interdev, etc.), you can download MSDE for free from the MSDN site listed below, or order a CD for a minimal shipping and handling charge. Please be aware that this offer is available *only* for owners of at least one Visual Studio component. If you get the CD, you also get the full-blown SQL 7 itself along with a developer license for it.

Editors' note: In 2005 MSDE will be replaced by SQL Server Express Edition

Resources

- MDSE Home page: <http://msdn.microsoft.com/vstudio/msde/default.asp>

Sybase SQL (SQL Anywhere Studio and Adaptive Server)

SQL Anywhere is one of the most popular RDBMS systems among Clarion developers, and with good reason. Reasonably priced, robust and filled with features that even some of the big boys don't have (or don't have working quite right), it has a vocal group of supporters on the Clarion newsgroups.

When I talk about SQL Anywhere Studio, you should know that what you actually get is a whole suite of programs and tools including the server itself, Sybase Central (a management tool), SQL Modeler (database design tool), Infomaker (a report writer), PowerDynamo (a web application server) and lot more.

A 60-day evaluation copy can be downloaded free or ordered on CD for a shipping and handling charge. Weighing in at a relatively slim 78 megabytes, I'd say the download version is still a bit big for most modems.

If you think you might actually use SQL Anywhere Studio, by far the better deal is to join their free Sybase Developer's Network (SDN). Once you are a member, you can order the SBN Subscription Package. This special deal for developers includes a developer license for *all* versions of SQL Anywhere Studio (and CD's for Windows and Unix in English, plus Japanese, French and German versions). Covering one year, it also includes a year of free updates, access to developer information, more tool downloads and other goodies. And yes, Linux is included as well.

The price? US\$99 plus some shipping and handling.

I joined SBN, ordered the package and had the CDs in my hot little hands within two days. Pretty awesome, I'd say. As an aside, I thought it was pretty cool that it even came with a 230 page book - until I looked at the book and found out that it was just the license agreement (in 35 variations and over 20 languages!)

By the way, although you don't hear about it much on the Clarion Newsgroups, you can also get evaluation copies of Sybase's flagship product, Adaptive Server Enterprise, at the Sybase product evaluation site.

Resources

- SQL Anywhere Studio info: <http://www.sybase.com/products/anywhere/>
- SDN SQL Anywhere home: http://sdn.sybase.com/sdn/mec/mec_home.stm

Oracle

Ah, yes - Oracle. The eight hundred pound gorilla of databases (and of downloads, as you'll see).

Oracle provides massively powerful databases (sometimes at a massively powerful price). It is their benchmarks and feature set that everyone is always trying to top, and their market share that everyone is always trying to grab.

For evaluation copies, you *could* try downloading the free 30-day trials they offer, but let me warn you that they are *BIG*. For example, Oracle 8i Standard is 450 megs plus. We aren't talking a big pipe here; we're talking the Channel Tunnel. And that is for just one product (or more accurately, one *version* of the product).

The more reasonable deal is to order one of their CD packs that contain all of the related Oracle products for a single operating system. For example, their CD pack for Windows NT includes the Enterprise, Standard and Personal editions of Oracle 8i, along with just about every possible option you can order for them. Thus, you have 30 days to try out the exact combination of options that you need, rather than just trying the database and hoping that add-on X will solve a problem.

CD-Packs are US\$39.95 plus shipping and handling, and are offered for seven different operating systems (including Linux). Apart from the current 8i version of Oracle, they also have packs for 8.0 and 7.3. A nice touch, if you have to develop for an older version.

Stepping up, Oracle also has (surprise, surprise) a developer network that has a better deal if you need more than a trial version, but less than a mega license. Called the Oracle Technology Network (OTN), one of the benefits is the ability to purchase one-year subscriptions to what they call "Technology Tracks." Each track is geared to one operating system and group of related products.

The one of most interest to Clarion developers would probably be the Internet Servers track for the Windows NT Platform. It includes developer licenses for Oracle 8i Enterprise, Standard, Personal and Lite, as well as WebDB and the Oracle Application Server. Other tracks are available for Linux, Unix, Sun Solaris, Netware and so forth. Each track must be purchased individually.

Like the Sybase subscription offering, Oracle tracks also include free updates for the year.

The price? US\$200 per track, plus some shipping and handling. Not bad for US\$5000 dollars worth of licenses.

Side note: With Clarion 5.5 Enterprise now including a developer license for the Oracle Connect driver, it is getting easier and easier to play around in the Oracle end of the pool without your wallet getting soaked (sorry, couldn't resist that one...)

Resources

- Oracle Databases main page: <http://www.oracle.com/database/>
- CD-Packs ('Trials' for info or Database->CD-Packs): <http://store.oracle.com>
- Oracle Technology Network home: <http://otn.oracle.com/index.html>

IBM DB2 Universal Database

Not an RDBMS you hear about a lot on the newsgroups, DB2 is nevertheless a force to be reckoned with. If you venture into an IBM shop, chances are very good that you'll need to deal with DB2 at some point.

I couldn't find any free time-limited evaluation versions, but that doesn't mean they aren't out there (I have one for the previous DB2 version on my desk right now). What I did find during my current search is that you can download an actual developers edition (i.e., not time limited) for free or obtain it on CD for a nominal charge.

Umm, did I happen to mention that the Oracle download was a bit on the large side? Well, hold onto to your modems, because the full download for the free DB2 Personal Developer's Edition is a whopping 542 Megs spread across eight files.

If you don't feel like spending the next year downloading DB2, you can also purchase the DB2 Personal Developer's Edition at the IBM online store for US\$39 plus the ever-present shipping and handling charges.

Note: If you decide to order DB2 on CD, just follow the various links to the DB2 product list and then choose the DB2 *Personal* Developer's Edition. Ignore the "specials" link on the main page that leads to the DB2 *Universal* Developer's Edition (unless, of course, you *want* to spend US\$500.)

Resources

- DB2 home page: <http://www-4.ibm.com/software/data/db2/udb/>

Pervasive SQL

Another database that has its share of vocal Clarion users, Pervasive SQL is somewhat of an odd child. Based on a hybrid system, it allows you to access data both directly (via Btrieve) and through the Pervasive.SQL engine (most RDBMS only allow access to the data through the SQL engine.)

Pervasive offers 30-day trial downloads of Pervasive.SQL 2000, and you can also order the trial versions on CD. Note that the Pervasive.SQL Server engine will not run under Windows 9x - it requires NT, Netware, Linux or a similar server OS. There is a Workgroup version, though, that should run under Win 9x.

For downloaders, Pervasive.SQL 2000 is the smallest of the bunch, weighing in at a mere 45 Megs or so.

In checking out how much it cost to order through their online store, I found out that the CD was free, but shipping and handling was US\$10. As a bonus for ordering trial software, there was a notice that I would also get \$10 off my order.

Well, after clicking "next" to get to what I figured would be some kind of order confirmation screen, all I got was a Thank You notice (uh, you mean I actually *ordered* it?). So, I can only report that there is now a totally free Pervasive.SQL CD wending its way to my door. Not bad, not bad at all.

By the way, even though a P.SQL developer license was included in the box with Clarion 5, I've included Pervasive.SQL here because the version on that CD is now obsolete. I don't know if Pervasive.SQL will still be on the 5.5 CD (or what version it will be) but, as you can see, it is *very* easy to get.

And yes, there is a Developer Zone, although I didn't see any special deals there (that doesn't mean that there aren't any, just that *I* didn't find them).

Links to downloads, ordering and more information are all accessible on the Pervasive.SQL 2000 home page.

Resources

- P.SQL 2000 home page: <http://www.pervasive.com/pgsql/>

Last Words On Licenses, Memberships, etc.

This may seem a bit redundant, but I should point out again that virtually every license obtained via any of these developer network packages is for *development* only - you cannot distribute it, use it in a production environment or install it for use at your clients.

In other words, they are giving it to you on the cheap so that people who want to use your fabulous program will also need to buy their database. The only exception is MSDE, which you can redistribute freely as long as you have a Visual Studio or Office Developer license (be sure to read the fine print, though). Even then, their ultimate aim is to have your users scale up to SQL Server.

General SQL

Also, be aware that in most cases you will be required to join that particular vendor's version of a developer network. *Nobody* will let you grab these RDBMS systems without extracting at least *some* information from you.

The good news is that the basic memberships are free in all cases. You'll also find them invaluable once you start digging into the products, as they give you access to special developer oriented sites chock full of information, tips, tricks, other downloads and a whole lot more.

On the documentation side, very few of these trial offerings come with any kind of paper documentation. PDF and help files are the norm here, folks.

One final point: Be sure and check the operating system requirements for the versions you download or want to install. Some RDBMS require Windows NT/2000, while others have versions that will run happily under Windows 9x as well as other operating systems.

Happy SQLing!

HOW TO CONVERT YOUR DATABASE TO SQL

by Scott Ferrett

This document is based on the talk given by Scott Ferrett at Euro Devcon '99 in Amsterdam on 22 April 99. It is based on the facilities available in Clarion 5a Enterprise Edition. Later versions of Clarion may require different (hopefully less) work. However, the information on file structure changes will probably still apply.

Reprinted with permission.

Throughout this chapter I will refer to TPS files. However, this chapter applies equally to any ISAM file format (Clarion, dBase, or Btrieve). I will refer to SQLAnywhere as the SQL driver. However, this can be replaced with Oracle, MS SQL, Scalable, AS400, ODBC or any other SQL file driver.

There are two things to do when creating an SQL database based on an existing set of TPS files: convert the data definitions and convert the data.

Converting the Data Definitions (Creating a new DCT)

The first thing to do is create a new DCT.

You then need to move the existing TPS tables into the new DCT and convert them to SQL tables. This can be done in two ways: copy the old DCT to the new DCT then change each table one at a time, or export the old DCT to TXD, edit the TXD and import the TXD into the new DCT.

I will use the second technique as it allows for bulk changes. However, this system is much more dangerous as you have no tools to assist you in making certain that everything you do is consistent and correct.

Driver Name

This is the most obvious change. You need to change **TopSpeed** to **SQLAnywhere**. *Do not do this yet.*

OWNER

All SQL tables require an OWNER attribute that indicates how to connect to the database where the tables live. *This should always be a variable.*

If you have any tables that already have an OWNER attribute, remove these.

Once you have no tables with OWNER attributes you can replace 'TopSpeed') with 'SQLAnywhere'),OWNER(GLO:Owner)

Key Component STRING => CSTRING

In most SQL databases trailing spaces are important when testing if a field is equal. So 'Smith' ~= 'Smith '. This is normally only important on key components as they are used in relational links and filters. If you use CSTRINGS instead of STRINGS, then Clarion treats trailing spaces in the same manner as the SQL system.

MEMO => STRING

SQL drivers do not support MEMO fields. These can be represented as very large strings. As you will be developing 32bit applications you do not have a 64K record limit to worry about.

LONG => DATE and TIME (sometimes)

In older systems date and time data is stored as a LONG. These should be converted to DATE and TIME fields so your data will be easily accessible from non-Clarion programs such as third party report writers.

No RECLAIM attribute

SQL drivers do not support this attribute.

Remove NOCASE

Every SQL system (except P.SQL) only supports the concept of case sensitive or case insensitive keys across the entire database. You cannot specify one key as case sensitive and another one as case insensitive.

Because of this global setting, SQL systems do not require the NOCASE attribute. In fact, specifying it can *significantly* impact on performance.

Change Referential Integrity to Server based

This is something you get for free. You do not have to learn any SQL to get the advantage of having the server do your referential integrity. To do this you need to change all your relational constraints to the equivalent server based relational constraints. If you are editing the TXD you do this by changing;

- CASCADE to CASCADE_SERVER;
- RESTRICT to RESTRICT_SERVER;
- and CLEAR to CLEAR_SERVER

You have now done all the quick TXD based changes. You can do the following changes in the TXD or after importing the TXD. I recommend the latter.

Make Sure All Files Have A Unique Key

There is no hidden record number in SQL. So for the driver to be able to update a record you must tell the driver how to uniquely identify a record. To do this you must have at least one unique key defined for a file. Even one record control files. The driver does not know that there is only one record.

Do Not Use GROUP IDs In Keys

It is fine to use fields that are within a group as components of a key. But you cannot use the GROUP field in a key. If you have keys that use a GROUP field, you will need to change the key to use each field within the key.

Converting The Data Definitions (Create an SQL Script)

To create an SQL Script you will need two dictionaries. One is the one you created in the earlier steps. The other is an SQL dictionary. You get an SQL dictionary by either creating a new SQL database or using an existing one.

Run The Dictionary Synchronizer To Create An SQL Script

Run the synchronizer. Select your other dictionary to be the SQL database you want to create the SQL tables in.

Set the Source DCT as the Clarion DCT you created in the previous section and the destination DCT as your SQL database.

Once you get to the synchronizer screen you need to copy all the files to your SQL database. The easiest way to do this is to highlight the top line. Press the right mouse button and select **Add**.

Run the script

To run the script you need to run your SQL's SQL executor. Load the script and run it.

Converting The Data

Having converted the data definitions, you now need to convert the data.

Create the Conversion Program

To create a conversion program you select **Create Conversion Program** from the **File** menu whilst you are in the dictionary editor.

You then end up back in the synchroniser. Don't panic. This is where you are meant to be. The synchroniser has many faces. One of these is to allow conversion programs to be created.

Choosing your dictionaries

To create a conversion program you need two DCTs. The original DCT that contains all the TPS files and the new DCT that contains the SQL table definitions.

The most confusing part of creating a conversion program is the screen that asks you which DCT is the source and which is the destination.

The **Source DCT** is the one with your **SQL** tables in it

Your **Destination DCT** is the one with the **TPS** tables in it

Once you get to the synchroniser screen you need to copy all the files. The easiest way to do this is to highlight the top line. Press the right mouse button and select **Copy**.

Press **OK**.

You now get the next confusing part of the conversion program creator. You find yourself back in the dictionary editor.

The system has actually done what you wanted. It just didn't tell you.

Edit the Conversion Program

A project convert.prj was created in the previous step.

Load this as the current project.

The first thing to do is change the properties of the project to create a 32bit program.

You will then need to edit the conversion program. The program is an object-oriented program designed to convert an existing set of ISAM tables to a new version of those tables. As such, it does not handle converting ISAM to SQL without a few modifications. Being an object-oriented program it allows you to make these modifications without having to hack the base code.

The file that needs to be edited is C5CVT__1.CLW.

Standard SQL Code Additions To Conversion Program

There are a few standard settings that need to be overwritten for all SQL tables. So the easiest way to do this is to create an **SQLDestTable** class derived from **DestTable**. You then change all your **DestTable** derived classes to be **SQLDestTable** derived classes.

```
SQLDestTable CLASS (DestTable)
CreateTable PROCEDURE (),RCCODE,DERIVED
AskName      PROCEDURE (BOOL _MustExist, <string FileLabel>)
```

General SQL

```
        ,BYTE,PROC,DERIVED
BuildKeys      PROCEDURE () ,RCCODE,DERIVED
    END
SQLDestTable.CreateTable PROCEDURE ()
    CODE
    RETURN RC:Ok

SQLDestTable.AskName  PROCEDURE (BOOL _MustExist, *lt;string
FileLabel>)
    CODE
    SELF.FileName = SELF.Label
    RETURN RC:OK

SQLDestTable.BuildKeys PROCEDURE ()
    CODE
    RETURN RC:Ok
```

Setting the OWNER

The conversion program creates an owner variable for every table. This is a pain. You need to change every OWNER() attribute to refer to just one string. You can then either add code to get a user ID and password from the user, or hard code it.

Set Tasks

The conversion program generator does not set the right tasks to be performed when doing this conversion. So you need to edit the SELF.Task = line to be:

```
SELF.Task = TASK:DefaultSQL + TASK:OpenSrc - |
    TASK:Backup + TASK:UpdateDest
```

There is one of these lines for each file being converted.

Convert Does Not Handle MEMO=>STRING

There is a bug in the conversion generator where it does not generate the necessary code to convert memos to strings. So in each table that had a memo you will need to edit the Assign procedure and add the line

```
NewFile.StrField = OldFile.MemoField
```

Modifying Your Applications

The only thing you will need to do is add a logon screen to the start of your program. You will need to ask the user for a User ID and Password.

You can hide this from the user by having a hard coded User ID and Password. Even in this case you will want to add a dialog indicating the program is connecting to the server as this can take some time (up to 20 seconds is not unusual).

The connection to the server is done when you first open a file. So you need to either open a file in your Logon screen, or on the frame. You can close the file immediately. The connection will be kept open until the application terminates or PROP:Disconnect is called.

Doing This For Client Data

This chapter has been written from the viewpoint of the developer having access to the database. If you want to convert an existing customer's data to an SQL system you will need to:

- Get the SQL system installed.
- Either copy over an empty database that already contains the table definitions, or create a new DCT and run the creation script.
- Run the conversion program

General SQL

CONVERTING TPS To MS-SQL

by Stephen Mull

This document relates my experiences and findings while performing a conversion of a large C5b Legacy app from TPS to Microsoft SQL Server. This document is based on Clarion 5b Enterprise Edition and MS-SQL Version 7.0. Using different versions of Clarion or MS-SQL Server may require different approaches, but information relating to file structures should still apply. I do not cover the details associated with MS-SQL 7.0 setup in this document, except where it relates to my Clarion application.

I have been using Clarion for Windows since the first release, and feel it is the finest RAD tool available anywhere! This was my first application involving the usage of a SQL based back end. While the initial learning curve has taken some time, I would certainly recommend the usage of SQL whenever possible.

My initial learning experiences began with reading the Clarion documentation, searching the newsgroups, and reading two really good articles relating to using Clarion with SQL. One is Rick Hoffman's "MS-SQL Tips and Tricks and C5" (Rick's original site is no longer up, but you can still get this article via the Wayback Machine: <http://web.archive.org/web/20020816013033/http://home.tampabay.rr.com/rhoffman/MSSQL-C5.document.htm>), and the other is a summary of the presentation by Scott Ferrett's presentation at Euro Devcon '99 in Amsterdam (see "How To Convert Your Database To

SQL,” p. 175). While all of the available information was indeed helpful, there was no single source covering the details of my specific needs. I felt my experiences might be of some assistance to other developers, thus this document.

It is important to understand that you, the developer, must use your best judgment regarding the usage of this information. While I feel I am relating accurate information based on my experiences, your situation may be different, thus requiring different approaches. On we go!

Which MS-SQL?

What works and what to use? There are a variety of choices. After investigating the options available, I chose to use the Clarion MS-SQL Accelerator instead of ODBC. I also decided that MS-SQL Server 7.0 was the best choice. I would not recommend you use prior versions as Version 7.0 is far superior and free from headaches for the most part. The finished application will also work fine with MS-SQL Server 7.0 Desktop Edition.

Other approaches, including using the MSDE or ODBC, probably work fine, but my choices seemed to be the most reliable and straightforward approach for my project. Whatever choices you make, if you use Windows 95 as your platform you will need to install the DCOM updates first, and then the new MS-SQL ODBC driver (3.7x). The new ODBC driver is recommended with MS-SQL 7.0. These are all included with the MS-SQL 7.0 CD.

Changes To The Dictionary

The first thing to do is create a backup of *everything!* I suggest you start with a new directory for the SQL app, and copy your existing APP, DCT, etc. to the new directory. You do not need to move the existing TPS tables into the new directory. You may use them later to copy the data to the SQL server, should you wish to do so. This will be discussed later in this chapter.

The first thing I had to do was make changes to the existing field definitions. Refer to the following table, which worked perfectly for me, excluding dates (to be discussed shortly).

This table of field type equates is excerpted from Rick Hoffman’s paper. Some changes and additions have been made to the original content.

SQL Field Type	Clarion Field Type
CHAR (20)	STRING (20)
VARCHAR (20)	CSTRING (21)
INT	LONG
BIT	BYTE
DATETIME	STRING (8) GROUP(Over String(8)) DATE ! you modify these fields TIME ! you modify these fields END or Date or Time
SMALLDATETIME	STRING (8) GROUP(Over String(8)) DATE ! you modify these fields TIME ! you modify these fields END
DECIMAL (18, 4)	DECIMAL (18, 4)
FLOAT	REAL
IMAGE	STRING (2048)
MONEY	DECIMAL (19, 4)
NUMERIC	DECIMAL (18, 4)
REAL	SREAL
SMALLINT	SHORT
SMALLMONEY	DECIMAL (10, 4)
SYSNAME	CSTRING (31)
TEXT	STRING (2048)
TIMESTAMP	STRING (8)
TINYINT	BYTE
VARBINARY	STRING (255)
STRING	MEMO *see note below

Global And Local Data Definitions

Don't forget to update all of your global and local definitions as well – they are easily overlooked! You will also need to add something along the lines of GLO:Owner for the owner name of each table (discussed below).

STRING To CSTRING

Look at the following string comparison: 'Dog' ~= 'Dog '. With MS-SQL 7 tables, the trailing spaces become an issue when comparing field values. In most cases, this is only an issue of importance with key components as they are used with relational links and filters. I still recommend you change all of them accordingly. If you use CSTRINGS instead of STRINGS (highly recommended), then Clarion treats the trailing spaces in the same manner as the SQL 7 system. I encountered no problems doing this. Just remember that STRING(20) = CSTRING(21) in Clarion, so add the extra length in your field definitions or you may end up with truncated data!

MEMO To STRING

The MS-SQL driver does not support MEMO. Instead, you may create your memo fields as a very large string. As your SQL app is 32bit, you do not have a 64K record limit to be concerned with. After you make this change, you will find that you will no longer be able to display and update the “memo” contents in the form you used with MEMO. What you need to do to correct this is go to the DCT, go to that field's **Display Properties** tab and change the control type from **Entry** to **Text**. Then go to the form, repopulate the field onto the form, and it will work properly once again.

DATE - TIME To DATETIME

My DCT contains fields which store both date and time as Date and Time. I left these “as is”, and upon sync with SQL 7.0 server, they were automatically converted to DATETIME and continued to function perfectly. Do not confuse DATETIME with DATETIMESTAMP on the SQL 7 server; they *are* different. I store my date values from within the app. This was appropriate for my application, but leaves the possibility of the data being incorrect if the workstation's date and time are not correct. Having SQL 7 insert date values is also possible, and will insure the correct date and time are used. If you do this, the SQL 7 Server will try to populate a DATETIME field with the date *and* time. It is possible to get around this, but I avoided it completely.

As I mentioned above, I store my dates and times separately. Clarion supports the DATETIME via a Group, as illustrated in the chart above, but reporting and filtering with it is a hassle I chose to avoid. Avoid storing various date and time data as a LONG. If you

store as a DATETIME, your data will be usable with third party products such as Crystal Reports, Access, Excel, etc.

Field Names And Definitions

I left my field names and key names the same, and all worked perfectly. There is a maximum field name length on MS-SQL tables, so be aware if you have excessively long field or key names. Regarding the use of external names, try to maintain identical field names. If for any reason the Clarion dictionary and the MS-SQL field names differ, you should set the External Name in the Clarion dictionary to the MS-SQL field name. Since you will sync from the DCT to the SQL 7 server, you should not encounter any issues in this area. Remember to update all field definitions to match the above list. One note about initial values and case: initial values will be set correctly using legacy code, even using recursive entries, but will not be set correctly using ABC, except on an initial insert. As far as case goes, it works fine, but keep in mind how this will relate to keys and NOCASE support, as mentioned below.

Of Keys And Indexes

Make sure all tables have a unique key. *This is very important!* Also realize that unlike Topspeed files, there is not a hidden record number in SQL 7 tables. You must inform the driver how to uniquely identify a record. To accomplish this, at least one unique key must be defined for each and every MS-SQL table. Do not use indexes with the MS-SQL driver, as they will not work properly.

Be sure to set all keys to either case sensitive or case insensitive. You must not attempt to use a mixture of both with your keys. The MS-SQL server's performance will suffer greatly if you do so. I learned this first hand! You must also not use GROUP fields as part of your keys. To restate this, fields within a group may be used as part of a key, but do not use any GROUP field in your key. I did not have any keys of this type, so I encountered no problems.

For files that required a unique record number or ID, I created a field called RecordID (@s18) for the file, and supplied its value from the app rather than the server. You may use the server to auto-populate this with an incremented number. To do this with MS-SQL, use an appropriate data type, most likely integer, and make it part of a unique key. Then mark the key as auto incrementing, exactly like you would in a Topspeed file. *NOTE: Topspeed and others claim this works, but I was never successful in getting this to work properly.* My method was to use Date() & Clock() on INSERT. Please don't call me

crazy; it works fine with over 100 users adding records every moment of the day, there has *never* been a duplication error, and it requires no interaction from the server!

Referential Integrity

MS-SQL 7 does not have a cascade delete declarative referential integrity feature. I understand this is planned for the next version of MS-SQL Server. Thus, SQL 7 will not enforce RI except when you ADD a record. To use the server to update and delete child records, you will have to create triggers and/or stored procedures to handle the process. I found this to be quite a pain with SQL 7. With all of this given, I would recommend you do not change RI to Server based! I left the dictionary “as-is” here, and all works perfectly! Additionally, this allows easier data manipulation via third party tools on the SQL 7 Server, but remember you can just as easily mess up your data using third party tools to manipulate data, so be careful!

File Relationships

None of the documentation has properly addressed file relationships, in my opinion. I initially used the sync tool in Clarion, created a SQL script, executed on my new database on the SQL 7 server, and voila, everything was created properly including relationships. I encountered all types of erratic problems with my app with the relationships defined both in my DCT and on the SQL 7 Server. Even with newsgroup and Topspeed tech support, all of these issues could not be resolved. Others may argue my final solution, but it is working perfectly. What I did was to leave the relationships intact in my Clarion DCT, and *did not* create the relationships on the MS-SQL server. Everything works perfectly, combined with leaving the referential integrity as stated above. Additionally, this allows easier data manipulation via third party tools on the SQL 7 Server, with the same caveat mentioned above.

Here is a helpful hint: *before* you sync your DCT with the SQL Server, save your DCT, then Save As a new DCT name, then remove file relationships from the new DCT, save, then use the Synchronizer to create the SQL script (covered below). Be sure to use your original DCT with your app, and not this new one without the relationships!

MS-SQL Driver Properties and Settings

Here are a few tips I learned from Rick Hoffmann’s paper and the newsgroups. See the Clarion docs for more information.

```
/SAVESTOREDPROC = FALSE
```

Note: sometimes the driver setting works backwards, so try both. You will notice the performance difference! What you want to do is not have the server save the temporary stored procedures that your app will create on the SQL 7 server.

- 1) /TRUSTEDCONNECTION = your choice, based on your situation. I did not use NT security, instead opting for SQL Server security.
- 2) /LOGONSCREEN = your choice, based on your situation. See OWNER Attribute.
- 3) /GATHERATOPEN – Not used with MS-SQL Accelerator. This is for ODBC use only.

Driver Options

Change “Topspeed” to “MS-SQL Accelerator”. Do this *after* all other changes are complete.

RECLAIM attribute

The MS-SQL Accelerator driver does not support this attribute.

Enable Field Binding Option

I enabled field binding in my DCT, and would recommend this to be enabled in most cases.

Enable File Creation Option

This does work properly with MS-SQL 7. I would still recommend the use of a SQL script instead.

OWNER Attribute

SQL 7 tables require an OWNER attribute. This indicates how to connect to the database where the tables are located. Use a variable whenever possible. If you have tables which already have an OWNER attribute, change them to something resembling !GLO:Owner. What do you do with this you might ask? See the Changes to the Application section below for information on how I handled this one!

Create a SQL Script Using The Synchronizer

To create a SQL Script you will need to first save your dictionary. Make sure you have completed all of the above actions before creating the SQL script, or you will have to do it again. You will probably not get it 100% right the first time, so don't worry! You will need your DCT file and an MS-SQL database. You get an MS-SQL database by either creating a new MS-SQL database or using an existing one. Make sure you have your MS-SQL server setup and running, and make sure you have installed the appropriate client software on your workstation. You may also use MS-SQL Server Desktop Edition if you do not have a separate server; it works fine.

I recommend you create a new database, using all defaults in Enterprise Manager. Also, save yourself some trouble and change the default sa database on the MS-SQL server login to your new database for the time being, or create a new login and give full rights to the new database to the new login.

Should you decide to follow my advice on not creating the file relationships on the SQL Server be sure to use a temporary DCT without the relationships before creating the tables, as mentioned above.

You run the Dictionary Synchronizer to create a SQL script, so first open your DCT, save, then run the Synchronizer. Select your other dictionary to be MS-SQL, and then select the database you want to create the SQL tables in. You will have to log in. Once you get to the Synchronizer screen you need to copy all the files to your SQL database. The easiest way to do this is to highlight the top line. Press the right mouse button and select **Add**. Click **OK**, then click **Finish** when offered. You will be prompted for a script name and location, so answer accordingly. That's it! Run the script from Enterprise Manager's Query Analyzer, found on the tools menu of Enterprise Manager's menu bar. Either load the script, or copy and paste from your script, select the database to run the script against in Query Manager and run it. You should be notified that the command(s) completed successfully in short time, and you're done! Now migrate your data if you wish.

How To Migrate Existing Data

I decided *not* to create a conversion program for my data dictionary. I used MS-Access and the Topspeed ODBC driver, along with the MS-SQL ODBC driver from Microsoft to migrate all data, using append queries. It is quick and easy! Your existing data might require some massaging before appending to your new SQL 7 tables, so you could import the data into new Access tables, perform your data manipulation, then append the data to the SQL 7 tables.

Views

Please note my initial app did not utilize views, but they are easy to use in your app. Assuming you know how to create a view, do so on the SQL server, give it a name like v myfile. In your dictionary go to the Synchronizer and import the view(s). Make sure you create a primary key, because when you import views a key is not created. Make the key look identical to the primary key of the view's main table. Make sure the following attributes of the key are set:

- Require Unique Key = On
- PrimaryKey = On
- Case Sensitive = On
- Exclude Empty Keys = Off

You may now use your view in your app!

PROP:SQL And Stored Procedures

Please note my conversion did not involve using any stored procedures. It is my understanding that working with stored procedures and other cool advanced functionality available from the MS-SQL server is best utilized via the CCS Client Server SQL Template Sets, available from Icetips (<http://www.icetips.com/>) . I intend to purchase and add to my app very soon!

Team Topspeed recommends that when possible you should write your own PROP:SQL statements to process data. Please note I *did not* do this for my initial conversion, and all works perfectly, so I may decide to leave everything as-is. The **Process** templates *do work!* The MS-SQL driver creates stored procedures in the TEMP DB for all SELECTs and for all inserts, deletes and changes. PROP:SQL eliminates the stored procedures for INSERT, DELETE and UPDATE.

The following are some performance and usage hints for working with PROP:SQL, excerpted from Rick Hoffman's paper. Some changes and additions have been made to the original content.

Use performance hints such as NOLOCK, FASTFIRSTROW and INDEX = when using PROP:SQL.

- 1) NOLOCK – Please read MS-SQL online documentation for description.
- 2) FASTFIRSTROW – Please read MS SQL online documentation for description.

- 3) INDEX – Sometimes MS SQL will not pick the correct Unique Constraint or primary key. Hinting will force MS SQL to use the specified PK or UC.
- 4) Don't use the RECORDS (TableName) on large tables to find the record count. Use a PROP:SQL with a SELECT Count (1) FROM TableName (NOLOCK). Since the select statement returns only one value you need to create a new file in the dictionary with a single field typed as a LONG.
- 5) For batch processes you may use BEGIN TRAN and COMMIT. Works like a charm, but monitor the transaction log for sizing (this is not a big issue with SQL 7).
- 6) NORESULTCALL – If the stored procedure is not going to return a result set then use the prefix NORESULTCALL. For example:
MyTable{PROP:SQL} = 'NOTRESULTCALL
SP_UpdateWhatever (1234)'.
- 7) CALL – If the stored procedure is going to return a result set via a SELECT statement then use CALL. Prefix is used. For example:
MyTable{PROP:SQL} = Call ('SP_SecuritySelectMembers
(''FL0021002 '')').
- 8) The data returned via a SELECT within the stored procedure needs to match the structure of MyTable. C5 only supports returning information via a SELECT. There are two ways to return things from a stored procedure. One is via an embedded SELECT and the other is RETURN (or similar command). The MS-SQL driver does not support returning information via RETURN, only SELECT.

Special Notes on PROP:SQL

The following special notes are excerpted from Rick Hoffman's paper. Some changes and additions have been made to the original content.

When issuing a PROP:SQL, issue a BUFFER (TableName, 0) before the PROP:SQL. This will tell the Cursor Fetch to retrieve one record at a time. Example:

```
BUFFER(TableName, 0)
TableName{PROP:SQL} = 'SELECT Field1, Field2, FROM TableName'
NEXT(TableName)
```

You can also issue a BUFFER (TableName, 20) and then push the PROP:SQL twice. This will tell the Cursor Fetch to retrieve 20 records at a time. Example:

```
BUFFER(TableName, 20)
TableName{PROP:SQL} = 'SELECT Field1, Field2, FROM TableName'
```

```
TableName{PROP:SQL} = 'SELECT Field1, Field2, FROM TableName'  
NEXT(TableName)
```

Issue a BUFFER(TableName, #) before the PROP:SQL.

When writing PROP:SQL statements its very easy to make syntactical errors. Team Topspeed recommended creating a debug procedure that's passed the SQL statement and displays it in a window. If you use a text field as the displayed field then you can cut and paste the SQL statement between your application and MS-SQL ISQLW or Query Analyzer. Example: SQLDebugWindow(TableName { PROP:SQL }).

Changes To The Application

In most cases very little change is required to an application for it to work with MS-SQL. Despite Topspeed claims that ABC is better than Legacy for SQL apps, I have found that Legacy offers excellent performance with MS-SQL 7.0. Additionally, due to MS-SQL's ability to self-tune, you will find performance will increase with each usage! I actually converted my app to ABC from Legacy, and have yet to eradicate all the minor bugs. I have decided to stay with Legacy for now.

As far as third party templates go, I use several third party template sets without issue. These templates include the following: CPCS Reporting Tools 5.1x (www.cpcs-inc.com), many templates from Sterling Data (www.sterlingdata.co.uk), some from Boxsoft Development (www.boxsoft.net), LSPack from Linder Software (www.lindersoft.com), Princen-IT Sendmail, and several other templates. I also use some custom templates that some very talented third party developers have written specifically for me, and yes, they *all* work fine with MS-SQL! Be sure to read *all* Clarion docs referring to using MS-SQL Accelerator and the SQL accelerators in general. Also read any sections on Browses, Processes, Reports, etc. where SQL is referenced. Although the docs are geared towards ABC, most information still applies in Legacy as well. You should actually do this *before* taking any of the steps described in this document.

Moving on to the app, one thing you will need to do is add a logon procedure at the beginning of the application. You will need to collect the SQL user id and password as a minimum. You *could* hard code this, hiding it from the user. If you decide to do this, remember to add some sort of message window telling the user to wait while the app connects to the server, as this could take a short bit of time.

Sample logon source is available in the Clarion docs, or you may create a Wizard app against any SQL database and the Wizards will automatically create the appropriate source code . Just cut and paste into your existing app. You may feel free to use the source in my sample app if you wish. An initial connection to the server is completed when any data file is opened, so open a file with your initial start. After opening any file, feel free to close it if you wish, as your connection to the server would remain until your app closes.

As mentioned earlier, MS-SQL 7 tables require an OWNER attribute. This indicates how to connect to the database where the tables are located. I created GLO:Owner as Global, CSTRING(255), and use an INI file to save and or retrieve and supply this string at runtime. I created a Clarion procedure that offers default SQL login definition, and store in the app's .ini file (this information could be stored in the registry for security or a TPS file with encryption if you wish). You must prompt your user to define the default login info using proper syntax. Refer to the Clarion docs for the Owner string format.

Embed source to retrieve the .ini entry and initialize GLO:Owner with this info in your main procedure before opening your SQL 7 tables. This way, if the string is empty, you will get the MS-SQL login. If it is populated correctly, the MS-SQL login will be skipped and login will occur automatically!

Another thing you will need to do is make sure that on any browse or process you set the Quick-Scan Records (Buffer Reads) option on. This will result in great performance increases! Any place you are prompted for an approximate record count, enter a number much greater than you actually ever expect to see with regards to records retrieved. If you leave this blank or zero, the process will first attempt to get a record count from the MS-SQL Server, which usually means a big performance hit!

Changing all browses to fixed thumb appears to enhance performance a bit, but records displayed in a browse sometimes disappear or appear twice. Your call on this one, as the book also said to do this. I did, but finally changed back. Also, make sure for any and all browses you have selected a key for the main file in the file schematic. This also applies to reports and processes.

If you find in a report or process that fields are coming up blank where you know data exists, check that all fields have the BIND attribute. This one tripped me up bad a few times! Be sure to check your embedded source or hand coded procedures if they involve data access, as these may require modification.

The browse and process templates access data through a Clarion view. This is why any field needed within a browse or process must have the BIND attribute. The easiest way to check things after making your changes is to run the app and see how it performs. Don't forget to address any auto-increment issues (mentioned above); these may throw you off as well! Also don't forget to bind any field used as part of a filter in your procedures.

Third Party Books And Reference For MS-SQL 7.0

- *Clarion Magazine* (<http://www.clarionmag.com/>) published by Dave Harms.
Fantastic Clarion resource!

- *SQL Server Magazine* (<http://www.sqlmag.com/>) published by Duke Press.
This too is a great new resource.
- *Using Microsoft SQL Server 7.0* by Que (<http://www.quecorp.com/>) . This book is very complete.

Summation

These are the main things I had to address with my conversion. I did not have any other significant issues. I did have a bit of trial and error, and you probably will too! Hopefully this chapter will help you avoid most of my trial and error process! Remember that I am not an SQL expert by any means! My methods were the result of reading everything I could find, asking lots of questions, and the trial and error process. If you are unsure about anything, post your questions to the newsgroups. Everyone is very helpful, and you will probably get the answers you need. I will make available for download a sample APP, DCT, and SQL script, along with a copy of this chapter in RTF format on my web site at:

- <http://www.mullusa.com/demo/sqldemo.zip>

General SQL

AVOID MY SQL MISTAKES!

by Mauricio Nicastro

The aim of this chapter is to show all the problems that I had to go through when I began to work with SQL. I'm hopeful that this could be helpful to those who, like me, are still not experts at working with this kind of database. Many of the things that I describe here may seem obvious, but the fact is that after being in a fix over and over you realize that they are not *so* obvious. The difficulties I encountered were the following:

Slow browses

All SQL tables must have at least one unique index. Whenever it is possible, I define autonumber fields that will be part of the principal key. At one time, I had a table with this field in two different keys: Code and Description.

- KeyCode = Code + AutonumberField
- KeyDesc = Description + AutonumberField

I assured my client that the execution of the program would be faster with SQL! The browse had two tabs and when I changed from a tab to the other, I had to wait 20 seconds in

order for the browse to show me something. It wasn't a big table (approx. 20000 records), and as you can imagine, my client wanted to make a programmer martyr of me; I didn't want to be Saint Programmer, not at so young an age!

I revised everything, worked with SQL Query Analyzer, looked for some problematic embedded code, but I couldn't find any anomalies. Just by chance, I found a manual in my bookshelves which contained the solution: all Clarion's indexes must be defined with the Case Sensitive field checked. If you don't do this, the operation is slow because the view engine will turn all the fields into uppercase while it is collecting the data.

Sending commands to the server

I had a requirement to do some mass updates to the data. The selective replacement of prices in a products table, for example, can be done in two different ways. One way is to make a loop using an index that matches, as much as possible, the selective criteria (this is the usual way when working with TPS tables). The other is to let the SQL server do the work. For example, in a general price list, I might need to increase by 10% the prices of those products bought from provider X. Here's how I would write this code in SQL:

```
Products{ PROP:SQL } = |
    'update products set price = price * 1.10 ' |
    & 'where ProviderCode = ' & Pro:Code
```

Now, is that right? It seems so; however, there is a small mistake: If `Pro:Code` is a CSTRING field, like any string it must be in single quotes. Thus the query, in this case, must be:

```
Products{ PROP:SQL } = |
    'update products set price = price * 1.10 ' |
    & 'where ProviderCode = '''' Pro:Code & '''
```

Remember that it is necessary to write three single quotes, one to open, the real quote, and the last to close.

Date Fields

The Clarion dictionary editor is not able to recognize a SQL date-time field. Instead, you will see a STRING(8) field, followed by a group declared OVER the STRING(8). This group has TIME and DATE fields. Let's say you have to order the browse by date with the use of a locator.

The point here is this: how is this index created? What are the key components? As a way of trying to figure all this out, I used the trial and error method, and then came to learn that

it is necessary to use the `Field_DATE` in the index key, because the name of this field will be used in the `ORDER BY` statement sent to the database server. Date ordering won't work if you use the string or the group field.

Filtered locators

Filtered locators are very useful, because while you are typing the browse is filtering. Besides, if the "Find Anywhere" field is checked it is possible to find the string the user types if it matches anywhere in the corresponding table field, not just when the start of the locator value matches the start of the table field. In my case, I don't know if it was my mistake, a Clarion problem or what, but I couldn't get Find Anywhere working. In my browse I had to filter a hot field and when I typed in the locator, the same letters appeared in the field and never worked. I tried with a variable as locator, then with the field, and it was impossible. As a result, I decided that the engine would do the work for me. I created a variable where I typed the string I was looking for, and in the accepted embedded point, I wrote:

```
IF Loc:Description <> ''
    BRW1::View:Browse{ PROP:SqlFilter } =|
        'Description LIKE ''%' |
        & clip( Loc:Descripcion ) & '%'''
ELSE
    BRW1::View:Browse{ PROP:SqlFilter } = ''
END !IF
BRW1.ResetQueue( Reset:Queue )
BRW1.ResetFromFile()
```

This code works, and very well, but of course only with SQL databases, as these normally support the `LIKE` matching function. You can add another variable and make it work as the "Find Anywhere" check.

Refresh the window

Sometimes you will encounter the typical invoice browse: header and detail. You want to add a new invoice, so you open the form, accept the entries and when you go back to the header browse ... the invoice you have just added is not there! You can do this:

```
BRW1.ResetQueue( Reset:Queue )
BRW1.ResetFromFile()
```

In this way, you refresh the queue and force the program to retrieve the data from the table.

Process

When you work with SQL you have to tell the engine which fields you need, by listing those fields in the browse's **Hot Fields** tab. With TPS files the entire record is always available; with SQL only those fields the browse/procedure/report knows about, because they're listed in the file schematic or in the hot fields, will be used in the corresponding SQL statement.

Parent-child relationships

Parent-child relationships can also be a problem. Take the invoice case as an example once more. Sometimes you use an autonumber field that is in charge of maintaining this relationship. In this case the question is: which program will autonumber this field? Your Clarion application, or the database server? I have read that I should leave this work to the engine. The problem is that the engine does this job in the same moment it is adding the record; consequently, all the children do not have a number which enables them to keep the relationship. If you want this working properly, you may add some code to retrieve the number of the field, then put this value to each child record and, finally, save everything. But if you have read Stephen Mull's "Converting TPS To MS-SQL," p. 183, you may note that it is better to use a CSTRING (18) field to keep this relationship. In the Field Priming on inset button, you can do the following:

```
RecordID = today() & clock()
```

That does work, though many of you might argue that there is a risk of getting duplicate keys when working with a large number of clients simultaneously. My counter-argument to this point is that I have worked with more than 150 terminals at the same time and that has never happened. I cannot say that it is not possible, however, it is not likely to happen.

These are all the interesting setbacks I had to sort out so far. If you're just getting started with SQL, I hope you find this discussion useful.

SQL DATA TYPES COMPARISON

by David Harms

The following table summarizes data type differences between three popular open source databases – Firebird (<http://firebird.sourceforge.net/>), MySQL (<http://www.mysql.com>), and PostgreSQL (<http://www.postgresql.org>) – and Microsoft’s SQL Server (<http://www.microsoft.com/sql/default.asp>). Please note that is a guideline rather than a definitive reference – I don’t have personal experience with all of these databases.

As you can see from the table, there is really very little standardization on either type names or specifications. In fact, there is only *one* data type you can use identically across all four servers, and that is the BIGINT type. INT or INTEGER types come a close second, but MySQL allows you to use an unsigned form which changes the range of values.

In many cases similar data types are available, but have different names. If you’re storing a BLOB, depending on the size of the BLOB and the server, you might call that data type BLOB, BYTEA, BINARY, TINYBLOB, MEDIUMBLOB, LONGBLOB, or IMAGE. String data doesn’t fare that much better. Although all four servers have a CHAR data type, each has a different size limit. In MySQL, it’s 255 characters; MS SQL, 8000 characters; Firebird, 32767 characters; PostgreSQL, approximately 1 gigabyte. Variable length strings (VARCHARs) have the same limits in each database as CHAR.

General SQL

Each server also has its own implementation for autoincrementing unique values; for instance, MySQL's `AUTO_INCREMENT` attribute on keys won't do you any good in PostgreSQL, where you need to make use of a Generator object.

Things get really hairy when you start dealing with fields that are a combination of a date and a time. You'll want either a `DATETIME` field or a `TIMESTAMP` field. But be careful! MySQL and SQL Server have `DATETIME` types which fit this description. In Firebird and PostgreSQL you have to use a `TIMESTAMP`. PostgreSQL's `TIMESTAMP`, however, can include a GMT offset, something not available in any of the other `TIMESTAMP` or `DATETIME` types. And don't think you can use the `TIMESTAMP` field in MySQL or SQL Server the same way as in Firebird or PostgreSQL. A MySQL `TIMESTAMP` field is set to the current date/time whenever you update the record, so you can't use it to store an arbitrary value; SQL Server is similar, although its value is also guaranteed to be unique across the entire database. And if you're converting between databases, keep in mind that there may be incompatibilities between the default date/time formatting.

Even simple `DATE` or `TIME` fields can present problems. Again, PostgreSQL `TIME` fields can store time zone offsets from GMT, which no other `TIME` type listed here supports. And SQL Server doesn't have `DATE` or `TIME` types, just `DATETIME`.

The moral of the story? As Mike Gorman has said, there really is no such thing as an SQL standard. Unless you know you'll never have to support more than one SQL server, try to stick with the more commonly supported data types. Be wary of arrays, boolean and bit fields, MySQL's ENUMs, and Postgress's specialized and user-defined data types.

Type	Firebird	MySQL	PostgreSQL	MS SQL Server
BIGINT	- 922337203685 4775808 to 922337203685 4775807	- 922337203685 4775808 to 922337203685 4775807	- 922337203685 4775808 to 922337203685 4775807	- 922337203685 4775808 to 922337203685 4775807
BIGSERIAL	use a GENERATOR object	use <code>AUTO_INCREMENT</code> on primary key	auto-incrementing integer, 1 to 922337203685 4775807	use IDENTITY
BINARY	see BLOB	see BLOB, TINYBLOB, MEDIUMBLOB, LONGBLOB	see BYTEA	Binary data, max 8000 bytes,

SQL Data Types Comparison

BIT	n/a	see TINYINT	Bit masks	0 or 1
BIT VARYING	n/a	n/a	Variable length bit masks	n/a
BLOB	Segment size limited to 64K, no limit on BLOB size	BLOB or Text up to 65535 bytes	see BYTEA	see IMAGE, TEXT, NTEXT
BOOLEAN	n/a	(BOOL) see TINYINT	True values include: TRUE, 't', 'true', 'y', 'yes', '1' - false values include FALSE, 'f', 'false', 'n', 'no', '0'	see BIT
BYTEA	see BLOB	see BLOB	Binary string, no specific limit on size	see BLOB
CHAR (n)	1 to 32767 characters	0 to 255 characters	approx 1 GB limit	1 to 8000 characters
DATE	8 bytes, 1 Jan 100 to 29 Feb 32768	1000-01-01 to 9999-12-31	4713 BC to AD 1465001	see DATETIME
DATETIME	see TIMESTAMP	1000-01-01 00:00:00 to 9999-12-31 23:59:59	see TIMESTAMP	January 1, 1753, to December 31, 9999, accuracy 3.33 milliseconds

General SQL

DECIMAL (precision,scale)	2, 4 or 8 bytes, precision=1-18, scale=0-18. Scale is the decimal places, must be <= precision.	-1.79769313486 23157E+308 to - 2.22507385850 72014E-308, 0, and 2.22507385850 72014E-308 to 1.79769313486 23157E+30	user-specified precision, exact, no limit	-10^38 +1 to 10^38 -1
DOUBLE PRECISION	8 bytes, range 2.225 x 10-308 to 1.797 x 10308	-1.79769313486 23157E+308 to - 2.22507385850 72014E-308, 0, and 2.22507385850 72014E-308 to 1.79769313486 23157E+30	8 bytes, up to 15 decimal places precision	see MONEY
ENUM	n/a	An enumeration of allowed values (similar to a CHECK constraint)	n/a	n/a
FLOAT	4 bytes, range 1.175 x 10-38 to 3.402 x 1038	-3.402823466E+38 to -1.175494351E-38, 0, and 1.175494351E-38 to 3.402823466E+38	see DOUBLE PRECISION	-1.79E + 308 to -2.23E to 308, 0 and 2.23E + 308 to 1.79E + 308
IMAGE	see BLOB	see BLOB	see BLOB	Variable length binary data, max 2GB

SQL Data Types Comparison

INT INTEGER	4 bytes, range -2,147,483,648 to 2,147,483,647	-2147483648 to 2147483647 or 0 to 4294967295	-2147483648 to +2147483647	-2,147,483,648 to 2,147,483,647
INTERVAL (between two TIME or TIMESTAMP values)	n/a	n/a	12 bytes, resolution one microsecond, - 178000000 to 178000000 years	n/a
LONGBLOB LONGTEXT	see BLOB	BLOB or Text up to 4 GB	see BLOB	see BLOB
MEDIUMBLOB MEDIUMTEXT	see BLOB	BLOB or Text up to 16777215 bytes	see BLOB	see BLOB
MONEY	see DECIMAL	see DECIMAL	Fixed precision (two decimal places), range - 21474836.48 to +21474836.47	-922,337,203,68 5,477.5808) to +922,337,203, 685,477.5807
NCHAR	see CHAR	see CHAR	see CHAR	Unicode string, max 4000 characters
NTEXT	see BLOB	see BLOB	see TEXT	Unicode text, max 1 GB
NVARCHAR	see VARCHAR	see VARCHAR	see VARCHAR	Unicode string, max 4000 characters

General SQL

NUMERIC (precision,scale) (usually equivalent to decimal type)	2, 4 or 8 bytes, precision=1-18, scale=0-18. Scale is the decimal places, must be <= precision.	-1.79769313486 23157E+308 to - 2.22507385850 72014E-308, 0, and 2.22507385850 72014E-308 to 1.79769313486 23157E+30	user-specified precision, exact, no limit	-10^38 +1 to 10^38 -1
REAL	see FLOAT	-1.79769313486 23157E+308 to - 2.22507385850 72014E-308, 0, and 2.22507385850 72014E-308 to 1.79769313486 23157E+30	4 bytes, up to 6 decimal places precision, floating point	-3.40E+38 to -1.18E-38, 0 and 1.18E-38 to 3.40E + 38
SERIAL	use a GENERATOR object	use AUTOINCREMENT primary key	Auto-incrementing integer, 1 to 2147483647	use IDENTITY
SET	n/a	A string that can have zero or more values from the allowed list.	n/a	n/a
SMALLDATETIME	see TIMESTAMP	see DATETIME	see TIMESTAMP	January 1, 1900, to June 6, 2079, accuracy 1 minute
SMALLINT	-32,768 to 32,767	-32768 to 32767 or 0 to 65535	-32768 to +32767	-32,768 to 32,767

SQL Data Types Comparison

SMALLMONEY	n/a	n/a	n/a	-214,748.3648 to +214,748.3647
TEXT	n/a	n/a	Approx 1 GB limit	2 GB limit
TIME	8 bytes, 0:00 AM- 23:59.9999 PM	-838:59:59 to 838:59:59	range 00:00:00.00+1 2 to 23:59:59.99-12 (shown with optional timezone notation)	see DATETIME
TIMESTAMP	Combination of date and time	1970-01-01 00:00:00 to sometime in the year 2037, automatically set to the date/time of the most recent update of the row	8 bytes, can include time zone, range 4713 BC to AD 1465001	A database- wide unique number that gets updated every time a row gets updated
TINYBLOB TINYTEXT	see BLOB	Text or BLOB up to 255 bytes	see BLOB	see BLOB
TINYINT	see SMALLINT	-128-127, or 0- 255	see SMALLINT	0-255
UNIQUEIDENTIFIER	n/a	n/a	n/a	Globally unique identifier (GUID)
VARBINARY	see MEDIUMBLOB	see BLOB	see BLOB	Max 8000 bytes
VARCHAR (n)	1 to 32,765 bytes	0 to 255 bytes	Approx 1 GB limit	1 to 8000 characters
YEAR	n/a	A year in 2- or 4-digit format	n/a	n/a

General SQL

Arrays	All datatypes except BLOBS	n/a	All built-in or user-defined data types	n/a
Geometric types	n/a	n/a	PostgreSQL includes a number of geometric data types such as line, point, lseg, box, path, polygon, and circle.	n/a
Network address types	n/a	n/a	PostgreSQL includes the following network address data types: cidr, inet, and macaddr	n/a
User defined types	n/a	n/a	You can create additional types with the CREATE TYPE command	n/a

THE SQL ANSWER COWBOY

by Andy Stapleton

Question: AS-400

Can AS-400 files be accessed from a Clarion 5 Professional application, with the only addition being the AS-400 drivers?

Ken Castleberry

Answer

From everything I gather this is true, as long as you have the interface from the PC to AS400. This used to be PC-Talk and was quite slow. Now a more native form of communication is available and speed has increased dramatically.

Before jumping into *any* client/Server arena check the communication layer associated with the systems. This includes any non-NT to NT/Windows platform. Some will perform quite well (usually those on TCPIP), others will run badly.

Question: Capitalization

How do I force a field in the table to uppercase or capitalize a word? Declaring an attribute in the data dictionary does not seem to have any effect. I am using MS SQL 7

Scott Jordan

Answer

After searching everywhere, I finally asked a good friend Ben Williams, since he has been saddled with MS SQL for quite a while. He confirmed my suspicions. In MS SQL 6.5 or 7.0 field attributes are unavailable, and the only method to force upper case or capitalization is via a trigger or your program. This is a major shortcoming in MS SQL in my opinion.

Here is a trigger that will force uppercase on a Name/Address/City/State:

```
CREATE TRIGGER UppercaseAddress
ON Names
FOR INSERT, UPDATE
AS
If Update(Fname) or Update(Lname) or update(Address)
or Update(State) or Update(City)
    update Names
    set LName = UPPER(Lname),
        Fname = UPPER(Fname),
        Address = Upper(Address),
        City = Upper(City),
        State = Upper(State)
Where Namsysid = any(select Namsysid from Inserted)
GO
```

Now you should also use UPPER on all your Clarion screens...

Question: Sybase, MS SQL, Oracle & Informix

I was interested to read your comparison of Sybase and MS SQL, especially as I believe that the latter is derived from Version 3 of the former. My question follows on from this. In your informed view, how do Oracle and Informix fit into the picture? How would you rank these in terms of effectiveness? Especially in terms of ease of use with Clarion? I'm assuming that the back end platform is NT. But what about scalability? Is it still true that MS SQL will only work on NT whereas Oracle will work on anything including Linux? What about the others?

James Fortune

Answer

Yes, MS SQL does require NT. Another reason to prefer Sybase over each of these is that Sybase is scalable and also has versions for Unix and Linux. I can speak more on Oracle rather than Informix so here is my best answer.

Oracle and Informix are more of a mainframe type technology. Both are platform independent and expensive in cost and maintenance. Oracle can be difficult to say the least; a lot of the convenience we enjoy is lost in Oracle. At the moment the reference manuals that I have had to purchase to know the changes for Oracle 8.0 are somewhere around 30lbs in four books.

One of the pros of Oracle is it's quite scalable. You can continue with Oracle throughout terabytes of data with quite excellent performance. If you are going into an Oracle shop and working with Oracle, here is a list of books that I find essential:

- Oracle 8 Tuning ISBN:1-57610-217-3
- Oracle8 The Complete Reference (Oracle Press) ISBN: 0-07-8822406-0
- Oracle8 DBA handbook ISBN: 0-07-882396-x

Question: Browse Speed

We use your templates for some of our browses but in other places it just isn't practical (switching DCT properties from TPS to SQL, etc. where your browses wouldn't work with TPS). Lately we tried having Clarion generate a browse pointing at an SQL file and it worked fine, except the browse takes forever to switch between tabs (primary key is a single long and unique (takes 30 seconds), where others can take up to 160 seconds). Is there some simple PROP or switch we could use to speed these browses up even just a little? Your template browses, when browsing the same file, take less than a second switching from any sort order we have tried!

Ivan Faulkner

Answer:

Using PROP:SQLFILTER or PROP:ORDER you can reset the ORDER BY. Here is some testing you can do: Turn on the driver trace program (drvtrace.exe) so you can monitor the SQL statements that Clarion is generating during the execution of the program. You will more than likely find that several fields are in the ORDER BY that you don't wish to have.

General SQL

Using the PROP:ORDER or the PROP:SQLFilter to modify the statement that is generated will alleviate some of your problems. Under certain circumstances, Clarion will generate an ORDER BY clause that has every field in the primary key, and also every field in the browse. Hence your delay.

Question

Can a Clarion application using Microsoft Data Engine (MSDE) be run successfully on a Macintosh network using SoftWindows 95? There won't be more than two or three concurrent users. (You didn't say they had to be easy questions, did you?)

Patrick O'Brien

Answer

The questions you have to answer first are:

- 1)** Does any Clarion program run on SoftWindows 95?
- 2)** Can I load and operate MSDE on the same platform?

What I would do is use a standalone Clarion program and try to run it under SoftWindows. If that runs then you have answered the first question.

Now run MSDE on the Mac and see if you can access the data in Northwind or Public. The last thing on your To Do list is to run a small standalone Clarion program that accesses one of the databases.

Question: MS SQL on Windows 95

Contrary to your [recent] column, MS SQL 7.0 does run on Windows 95. I have it installed on my notebook. I can connect other Windows 95 or NT clients to my notebook and frequently do for load testing.

Jim Kane

Answer

I appreciate your feedback, and you are correct, at least under these conditions: MS-SQL 7.0 will run on Windows 95 if DCOM is loaded or you have the patch that upgrades the Internet Explorer to 4.01. Without those, MS-SQL will not run on Windows 95. In effect you are upgrading to Windows 98 components. I have also loaded MS-SQL 7.0 to my 95 machine by doing the same thing.

Question: PROP:SQL

I understand that TS no longer recommends the use of {PROP:SQL}. What is the replacement and can you explain the rationale?

Answer

To my knowledge {PROP:SQL} is not going away but you have to be careful using it, which is why they don't recommend it. If you use {PROP:SQL} and NEXT() with a PUT() you must insure that you have the primary key fields in the SELECT statement.

The reason is that PUT and NEXT use the primary fields to insure the update of the record is accurate. If you don't provide this in the SELECT Statement both the PUT and NEXT will not be aware of the record position and subsequently give errant results or none at all. You can get around not using the primary in the SELECT, but you will have to create your own Update Statements.

TS is redeveloping the file drivers for additional functionality, but I cannot discuss this yet.

Question: Devcon 98 Examples

I have read your seminar materials for Devcon 98 where a sample file was downloadable. In that particular file called 'Devcon.doc', all the examples of stored procedures and triggers are shown using SQL Anywhere. Do you have an equivalent in MS-SQL?

Yeoh Eng Loke

Answer

The syntax of MS-SQL is different from the syntax of Sybase, but there are similarities. Here is an example of MS-SQL trigger:

```
[Trigger Definition ]
Trigger checks the format type in the setup table to see
if the user wants First then Last or Last then First,
Updates the FULLNAME field to match.

CREATE TRIGGER Names_FullName ON Names
FOR INSERT, UPDATE
AS
DECLARE @NameFormat varchar(10);
SELECT @NameFormat = NameFormat
      FROM SETUP WHERE setupsysid = 1;
IF UPDATE(Fname) OR UPDATE(Lname)
    IF @NameFormat = 'FirstLast'
        UPDATE Names SET FullName = Fname + ' ' + Lname
        WHERE Names.NameSysID =
              ANY(SELECT NameSysID FROM INSERTED)
              AND Fname IS NOT NULL AND Lname IS NOT NULL
              AND Fname <> ' ' AND Lname <> ' '
    ELSE
        UPDATE Names SET FullName = Lname + ', ' + Fname
        WHERE Names.NameSysID =
              ANY(SELECT NameSysID FROM INSERTED)
              AND Fname IS NOT NULL AND Lname IS NOT NULL
              AND Fname <> ' ' AND Lname <> ' '
```

The same trigger in Sybase is:

```
CREATE TRIGGER Names_FullName BEFORE INSERT UPDATE
    Order 1 ON Names
    REFERENCING OLD AS OLDESTUFF
    NEW AS NEWSTUFF;
BEGIN
DECLARE pNameFormat VARCHAR(10);
SELECT NameFormat INTO pNameFormat FROM SETUP WHERE SetupSysID = 1;
IF OldStuff.Fname <> NewStuff.Fname OR OldStuff.Lname <>
NewStuff.Lname
    IF pNameFormat = 'FirstLast' THEN
        SET NewStuff.FullName = NewStuff.Fname||' '||NewStuff.Lname;
    ELSE
        SET NewStuff.FullName = NewStuff.Lname||' '||NewStuff.Fname;
    END IF;
END IF;
END
```

Personally I think the Sybase code is cleaner and more readable than the MS-SQL version. Also with Sybase you have the option in the trigger to allow a WHEN condition which will eliminate two lines of code within the trigger and will also stop the execution if the WHEN condition is not met.

```
CREATE TRIGGER Names_FullName BEFORE INSERT, UPDATE
    ORDER 1 ON NAMES
    REFERENCING OLD AS oldStuff NEW AS newStuff
    FOR EACH ROW
    WHEN(OldStuff.Fname <> NewStuff.Fname
        OR OldStuff.Lname <> NewStuff.Lname)
    BEGIN
        DECLARE pNameFormat VARCHAR(10);
        SELECT NameFormat INTO pNameFormat
        FROM SETUP WHERE setupsysid = 1;
        IF pNameFormat = 'FirstLast' THEN
            SET NewStuff.FullName = NewStuff.Fname||' '||NewStuff.Lname;
        ELSE
            SET NewStuff.FullName = NewStuff.Lname||' '||NewStuff.Fname;
        END IF;
    END
```

All I did was move one line of code from the `IF` clause to the `WHEN` condition of the trigger.

All I can promise you at this time is the more I work with MS-SQL 7.0 the better it feels even with the changes, so there will be more to come...

Question: SQL Anywhere vs. MS SQL

I have been using SQL Anywhere 5.5. I now see that version 6 has been released. Also, I hear a lot of talk about MS SQL. What are your thoughts and experiences and would you recommend one above the other?

Austin Drum

Answer

The newer versions of Sybase (6.0 and better) use symmetrical multiprocessing now. This means that each query or stored procedure will use all of the processors in your server. For the most part the individual would not necessarily know the difference, but in a larger environment you can see a significant increase in speed.

MS-SQL has definitely came a long way since 6.5, but I still think the language needs to mature a bit more. The documentation of methods and examples is not very instructive. I recommend you read *The SQL Server 7 Developers Guide* (ISBN 0-07-882548-2).

Both databases are quite capable of doing an excellent job, both for the Enterprise and personal user. MS-SQL is more expensive per seat and from a Clarion standpoint it's a bit

harder to grasp the nuances. On the other hand the Sybase language is more like Clarion and easier to understand.

Question: PROP:SQL on Reports and Lookups

How do you get PROP:SQL to work on a lookup and report? I want to do a lookup from a SQL database so that only a subset of records is returned. I understand it will dramatically reduce network traffic.

Yeoh Eng Loke

Answer

If you are using the standard templates, the best method is using the PROP:SQLFILTER on the Lookup form. You can change the PROP:SQLFilter at a whim by using the browse with a passed parameter. To do this open your browse and insert (STRING) into the **Prototype** field, and (pfilter) in the **Parameters** field. Now when calling the lookup you do it like this:

```
MyLookup('FieldA like <39>TEXAS%<39>')
```

In the Lookup procedure you will in an appropriate embed point have the following code.

```
MyView{PROP:sqlFilter}=pfilter
```

Remember that MyView is the Clarion-created view in the lookup and pfilter is the passed filter you sent. This will add your filter onto the Clarion-generated SQL statement and reduce your record set as desired.

Also be sure to upgrade to C5 if you haven't already. C4A and C4B both return the entire record on a file select, and while this may not be an entirely bad thing it does cause unnecessary traffic.

Question: Views

Is a VIEW structure generally an efficient approach for accessing a SQL back end? I have manually coded a view, including ORDER and FILTER options, and I am simply going through the view, forwards, only once, reading the values and using them. The records themselves are not displayed, and the user cannot scroll backwards through the list or anything like that. (This isn't a browse - more something like a report). Sort of like a SQL

SELECT - I want to get a reasonably generic method (which a view is) but which is also reasonably efficient.

Bruce Johnson

Answer

The view structure is for more complicated items the best method to use, as it allows you to skip fields in the dictionary structure and also join additional tables together.

For example, think of Names and Address tables. The address table is a child table to names, and what you want to do is get the billing address on screen.

```
Create view NameAddr      view (NAMES)
  Project (NAM:Namesysid)
  Project (NAM:Fname)
  Project (NAM:Lname)
  Join (Addr:NameKey, NAM:Namesysid)
  Project (Addr:Address1)
  Project (Addr:Address2)
  Project (Addr:CITY)
  Project (Addr:STATE)
  Project (Addr:Zip)
  END

  open (NameAddr)
  IF Errorcode(); Stop (Error()) .

  NAMEADDR{PROP:SQL}=|
    'Select NAM.Namesysid,NAM.Fname,NAM.Lname,' & |
    'Addr.Address1,Addr.Address2,Addr,CITY,Addr,' & |
    'STATE,Addr:Zip' & |
    ' from '&NAME(NAMES)&' nam,'&NAME(ADDRESS)&'addr '&
    ' where Addr.Namesysid = NAm.Namesysid'
  If Errorcode(); Stop (FileError()) .
  Next (NameAddr)
```

Now you can be flexible; the only item you have to remember is the fields still have to be in file/field order for the Clarion record buffer to be accurate.

Question: SELECT

Andy, how do I count the number of records returned by a SELECT statement?

Perplexed in Peoria

Answer

You can return the number of records easily by using a dummy file. Create a dummy file in your database:

```
Create table dummy (
    dummycounter    integer
);
```

Create the same in your dictionary. Now when you wish the correct number of records you return it into the dummy file. For example, I might want to return the number of addresses in Texas for Zipcode 78833.

```
Dummy{PROP:Sql}='Select Count(*)  '&|
    'from '&NAME(AddressTable) &|
    ' where State = <39>TX<39>'    &|
    ' and zipcode = '&78833
If Errorcode();Stop(FileError());
Next(dummy)
ScreenCounter = Dum:DummyCounter
Display
```

The NAME (ADDRESSTABLE) is a Clarion function that returns the external name of the dictionary; this is needed if you use multiple accounts on the database for login. The <39> is the ASCII equivalent to a single quote. You can use the dummy table for more than one thing; it can also return any other integer you need back to the Clarion program.

Open Source SQL

USING CLARION WITH MySQL

by David Harms

There's a lot of interest in Linux these days among Clarion programmers. Some are even asking for a Linux version of Clarion, and while that's not likely to happen any time soon, there may still be a place for Linux in your Clarion toolset as a database or file server.

In this chapter I'll look at the popular MySQL (<http://www.mysql.com/>) database and how you can use MySQL on Linux from your Clarion application. Although MySQL is available for Windows, I'll only look at the Linux version. There is a license fee for the Windows version, but not for commercial use of the Linux version unless, and I'm paraphrasing here, you make money from the sale or service of the MySQL server.

My exposure to MySQL on Linux came about because of *Clarion Magazine*, which has always been served by a Linux box. For the first couple of years I used the Apache web server, and I relied on Apache's directory-based authentication to handle subscriber-only access. In this scheme, each directory is assigned an authentication group, and users (subscribers) are added to one or more groups. It's possible to use text files for the authorization lists, but once you get over a certain size of list these become quite slow as they aren't indexed. At first I used a flat file (DBM) database, but there were some problems with DBM files on RedHat Linux, and I wasn't happy with the results. It all came to a head when the user file reached 64k and the authorization database maintenance

program (a Perl script) refused to add further records! I had one very intense afternoon installing MySQL and rebuilding Apache to use the MySQL database for authentication.

Moving to MySQL solved the database reliability problems, but the Perl script was still quite slow - it could take 20 seconds to update a single user's access. Since the rest of the magazine subscriber administration is handled by a Clarion application, it made sense to update the MySQL database the same way. All of the authentication database updating is now handled by a Clarion application, and I've been very happy with the results.

Clarion applications talk to MySQL databases on Linux using the MyODBC (<http://www.mysql.com/products/connector/odbc/>) driver. The basic steps to getting this working are: install Linux; install MySQL on the Linux box; set up the MyODBC driver on your Windows machine; configure a data source, and then specify that data source in your dictionary; and configure MySQL on the Linux box to allow your Windows computer to connect. Provided you have a LAN/WAN or even an Internet connection to the Linux box, you're ready to use MyODBC tables in your application!

First I'll cover the Linux installation and setup, and MySQL installation, and then I'll look at using MySQL with Clarion.

Installing Linux

When I first set up a Linux box in late 1998 I didn't find it a particularly straightforward process. I muddled through the install without major problems, but in the finest *nix tradition, I had to recompile the kernel to add support for some of the features I needed.

Happily, Linux installations have come a long way since then. I recently set up a machine using RedHat Linux 6.1, and it was pretty easy. There are, however, a few concepts you should know.

Partitions, Directories And Mount Points

Linux file systems are set up a bit differently from Windows file systems. In Windows, you have a drive letter for each partition. In Linux you have directories instead of drive letters. But it's not quite that simple.

In the simplest Linux configuration you will need to create two partitions on your hard disk, typically during the installation process. One is a swap partition, which doesn't have a directory associated with it, and the other is the root partition, which is the top level directory. When creating this partition in the installation process you label it with a / (notice the forward slash rather than the Windows style backslash).

In most cases, however, you'll want at least one more partition, of no more than 16 MB, which will contain the operating system kernel and other boot files. This partition has the mount point /boot.

If you have a 500 MB drive, and a 100 MB swap partition, and a 16 MB /boot partition, that leaves 384 MB for the / or root partition. Both /boot and / are mount points, meaning when you switch to either of those directories (using cd / or cd /boot) you're switching to the corresponding partition.

But isn't /boot under the root directory? Logically, yes. But as soon as you drill down to /boot you're on a different partition because of the mount point. Any other directories under / which don't have a mount point specified will be on the root partition. Many Linux systems will have other partitions such as /usr, /usr/src, /tmp, /var, and so on. I use the following partitions on a 10 gig drive on the *Clarion Magazine* backup server:

Mount point	Size
/	1 Gig
/boot	16 Meg
/home	2 Gig
/tmp	1 Gig
/usr	3 Gig
/usr/src	1.2 Gig
/var	500 Meg

By using partitions and mount points you can restrict the amount of disk space available to any part of the directory tree. On a single user system that usually isn't a huge issue, however, so Linux installations for the desktop seem to be trending toward the single root partition (plus swap partition). Having a /boot partition is still a good idea when dealing with a very large root partition in the event that the BIOS is unable to handle the large partition size directly.

Window Managers

Another big difference between Windows and Linux is in the implementation of the GUI. In Windows the GUI is part of the OS; in Linux, and other *nix systems, the GUI is a separate system. The Linux GUI is itself layered. First there is the X Window windowing system, currently in version 11, which handles GUI basics. Then you have a window manager, and over that a desktop environment. There are a number of window managers

available, such as FVWM, AfterStep, and Enlightenment, and the two most popular desktop environments for RedHat Linux are Gnome and KDE. See <http://www.plig.org/xwinman> for more information on window managers and desktops.

RedHat 6.1 gives you the option of installing a Gnome Workstation, KDE Workstation, Server, or Custom configuration. I went through the process a few times, just for kicks, and in the end I settled on the Gnome Workstation install. You might think the Server install would be the logical choice, but that's intended more for a machine that's to be a mail/web/news server. If you want all those things, go ahead and use the Server configuration. Just remember that the more services you have running, the more potential security holes you have to deal with.

By installing the Gnome Workstation I avoided installing a lot of unneeded services, and I also got a GUI. That was a help as I needed to download the MySQL software, and while I could have used the text-mode Lynx browser, I find Netscape just a bit more user-friendly.

Security Issues

Now that I've admitted to hooking my Linux box up to the Internet, I suppose a few words about security are in order.

It's very easy to use a MySQL database across the Internet. By default, the MySQL server listens for connections on port 3306, and if you have MyODBC installed on your Windows machine and you know the name or IP address of the MySQL server, you can in theory access your data from anywhere. And because of this MySQL has a fairly robust access/privilege system to control who gets in.

If you are going to expose your MySQL connection to the world, however, you need to think about more than just MySQL's own security system. You really need to install, or get behind, a firewall.

Even though the MySQL server has an access control system, you may want to limit attempts to connect to that port to only certain trusted machines. As well, almost any Linux box will, by default, have other services which are listening on other ports. These can include telnet, ftp, http, news, mail, finger, name services, and more. You can strip out all the services which you don't want, but often a firewall is a better and simpler option.

The traditional way to create a firewall on Linux is with ipchains (or ipfwadm, in earlier versions). ipchains is a packet filter, which means it can inspect all packets which travel through any of your network adapters. You don't necessarily have to have a separate machine; you can run ipchains directly on the machine you're protecting.

Packet filters aren't the be all and end all of firewalling. All they do is inspect each ethernet packet and accept or reject it based on the rules you set up. These rules don't apply to the data in the packet, just to the packet header, in particular the packet type, the source, and the destination. You can use packet filters to allow/disallow packets from certain sources, or which are destined for particular ports.

The two common strategies are to allow all packets by default, and block packets on problem ports, or to deny all packets by default, and allow only the packets you know you want in. I recommend the latter approach. For more information on Linux firewalling see <http://linux-firewall-tools.com>.

Getting To Know The Penguin

There are a lot of little differences between Windows and Linux that will probably drive you crazy for a while, if like me you have little if any *nix background. Using forward slashes instead of backslashes when specifying directories was easy to get used to, but going back to Windows was maddening. Whose brilliant idea was it to use that pinkie-wrenching backslash anyway? The ls command, equivalent (loosely) to dir, isn't all that descriptive when used without any options. And the text mode vi (or vim) editor is a bit of a throwback. Get yourself a good Linux book and be prepared for a learning curve.

Installing MySQL

Once Linux is running and you're moderately comfortable with its use, you're ready to install MySQL. The easiest way to do this, assuming you have your Internet connection established, is to log in as root, run the window manager (startx) and load up NetScape. Go to www.mysql.com and, to be nice to the MySQL folks, click on the Mirrors link and go to a mirrored site close to you. (But be warned - not all mirror sites carry all files!)

From the mirrored site go to the Downloads page. You'll need to follow two links from here; one for MySQL, and the other for MyODBC. First, look for the Unix Platforms link

and click on the “recommended” release. As of this writing, it’s 3.22.32 (Editor’s note: the current version is now 4.0.20). A portion of that download page is shown in Figure 1.

Figure 1: Download options for MySQL on Linux

The screenshot shows a web page with a light blue header and a white body. The body contains several sections of text and links. At the top, there's a note about RPM installations. Below that, there are two main sections: 'Source downloads for 3.22' and 'Standard binary RPMs for 3.22'. Each section lists multiple links to various MySQL RPM files for different architectures and purposes.

Note for RPM installations: If you are using the **RPM** to install, you should install at least the server and client **RPMs**.

Source downloads for 3.22

- [Tarball](#) (Including MySQL benchmarks and MIT pthreads).
- [Source RPM](#).

Note that this RPM does not work on RedHat 5.x systems because of glibc incompatibilities. Download the source rpm and do a rpm --rebuild to get a 5.x RPM

Standard binary RPMs for 3.22

- [The server for alpha systems \(RedHat\)](#).
- [The server for i386 systems \(RedHat\)](#).
- [The server for ppc systems \(RedHat\)](#).
- [Benchmarks/tests using Perl DBI for alpha systems \(RedHat\)](#).
- [Benchmarks/tests using Perl DBI for i386 systems \(RedHat\)](#).
- [Benchmarks/tests using Perl DBI for ppc systems \(RedHat\)](#).
- [Client programs for alpha systems \(RedHat\)](#).
- [Client programs for i386 systems \(RedHat\)](#).
- [Client programs for ppc systems \(RedHat\)](#).
- [Include files and libraries for development for alpha systems \(RedHat\)](#).
- [Include files and libraries for development for i386 systems \(RedHat\)](#).
- [Include files and libraries for development for ppc systems \(RedHat\)](#).
- [Client shared libraries for alpha systems \(RedHat\)](#).
- [Client shared libraries for i386 systems \(RedHat\)](#).
- [Client shared libraries for ppc systems \(RedHat\)](#).

Figure 1 shows links to various MySQL RPM install files (RPM stands for RedHat Package Manager). You’ll need to download “The server for i386 systems (RedHat)” and “Client programs for i386 systems (RedHat)” (<http://dev.mysql.com/downloads/>) and then run

```
rpm -i package-name
```

Install the server first, then the client. If you’ve been successful, you should be able to type

```
mysql
```

at the prompt and see something like Figure 2. Spend a bit of time with the MySQL documentation before you go any further.

Figure 2: The MySQL command-line client

The screenshot shows a terminal window with a black background and white text. It displays the MySQL monitor welcome message, which includes the connection ID, server version, and a help command. The prompt 'mysql>' is visible at the bottom.

```
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 1 to server version: 3.22.32
Type 'help' for help.

mysql>
```

Last Resorts

If you have an older version of RedHat, or a non-RedHat Linux, you can look on the downloads page for a binary distribution or RPM. If that fails, you'll need to download the source code and compile MySQL yourself. It may sound a bit ridiculous to have to compile Linux programs before you use them, but in fact it's commonly done. Most Linux installations include a basic set of compile and make tools, and usually all you do is change to the directory where you installed the source, and type:

```
./configure  
make  
make install
```

Always check the README to be sure you're doing it right, however.

You have Linux and MySQL installed, and that's half the battle.

Administering MySQL

If you've installed MySQL from the binary RPMs, then the MySQL utilities will be on the path, probably in /usr/bin. These utilities include mysql, which is the Linux MySQL client, and mysqladmin, which as you might expect is used for a variety of administrative purposes. These are both text mode applications, and although there are some graphical MySQL client and utility programs available for Linux (one of the most popular is MySQL Manager from ems-hitech.com), you should familiarize yourself with mysql and mysqladmin.

Getting Started

Your first step to Clarion/MySQL success is to make sure the MySQL server is running. At the command prompt type

```
ps x | grep mysql
```

The ps (process status) command tells you what processes are active, and the pipe character runs the output through grep which only returns the lines containing the "mysql" string. You should see something like this:

```
404 ? S 0:00 (safe_mysqld)  
439 ? S N 0:05 /usr/local/mysql/  
libexec/mysqld  
489 ? S N 0:08 /usr/local/mysql/
```

```
libexec/mysqld
```

If MySQL isn't running, type

```
safe_mysqld &
```

at the prompt to start the server. (Actually you shouldn't run this command as root, for security reasons.)

TIP: If you can't find a file you're looking for, try the `locate` command, as in `locate safe_mysqld`. If that doesn't work, run `updatedb` and try again. `locate` uses a database that typically is updated automatically once per day (via a cron job), so if you've just installed some software and you don't know where it went, you may need to run `updatedb` manually.

To see what databases are currently available, type:

```
mysqlshow
```

You should see a listing similar to this one:

```
+-----+  
| Databases |  
+-----+  
| mysql |  
+-----+
```

The `mysql` database is the administrative database. This can get a bit confusing, so pay attention to the case. When I write MySQL I'm referring to the product as a whole. When I write about the `mysql` database, I'm referring to one particular database (collection of tables) which has the special purpose of controlling access to all databases available to MySQL on that server. Type:

```
mysqlshow mysql
```

and you'll see the tables in the `mysql` database:

```
+-----+  
| Tables |  
+-----+  
| columns_priv |  
| db |  
| func |  
| host |  
| tables_priv |  
| user |  
+-----+
```

As you can guess from the table names, the `mysql` database contains information on which users are allowed to access which databases, tables, and columns. All user data will be contained in other databases which you will need to create. You should *not* add tables to the `mysql` database.

Controlling Access

The key to using MySQL is understanding its access control system. The three main tables involved are `user`, `db`, and `host`. All are contained in the `mysql` database. These tables represent a hierarchy of access control. In most cases you'll probably only need to work with the `user` and `db` tables.

Here is the structure of the `user` table.

Field Name	Sample Data	Purpose
Host	localhost	computer the user is connecting from
User	root	User ID
Password		Password (usually encrypted)
Select_priv	Y	Select records
Insert_priv	Y	Add records
Update_priv	Y	Update records
Delete_priv	Y	Delete records
Create_priv	Y	Create tables and databases
Drop_priv	Y	Drop tables and databases
Reload_priv	Y	Execute mysqladmin reload, refresh, flush-privileges, flush-hosts, flush-logs, flush-tables
Shutdown_priv	Y	Execute mysqladmin shutdown
Process_priv	Y	Execute mysqladmin,

File_priv	Y	Read/write files on the server!
Grant_priv	Y	Give others same privileges as self
References_priv	Y	Not implemented?
Index_priv	Y	Create/drop indexes
Alter_priv	Y	Execute alter_table

Any connection to the MySQL server involves at a minimum a host name (the computer from which you’re connecting) and a user id. If you’re connecting from the Linux box, then these will default to localhost and whatever user you’re logged in as.

The root User

By default no password is set for the root user, as shown above. This means anyone who logs in as root on the MySQL machine has complete control over the MySQL database. Your first step in configuring MySQL permissions should really be to set the root password. Assuming you’re logged in as root, use the command:

```
mysqladmin password mynewpassword
```

This will update the password field in the user table entry for `root/localhost`. You’d better write that password down in a safe place, because without it you’ll have to reinstall the `mysql` database to get access!

Once you set the root password you’ll need to specify it whenever you connect to MySQL. You do this with the `-p` option. For instance, if your password is “mypassword” then you’ll need to call `mysqlshow` this way:

```
mysqlshow -p
```

and type the password when prompted, or type

```
mysqlshow -pmypassword
```

or

```
mysqlshow -password=mypassword
```

The last two options will display your password on screen, of course. Perhaps no one is looking over your physical shoulder. Even so, someone could be looking over your virtual

shoulder. The `ps` command lists any currently executing processes along with their command line parameters, so in theory someone could telnet in, run `ps`, and see your password.

The rest of the columns in the user table control what kind of access this user has to the database.

The db And hosts Tables

Privileges granted in the `user` table are effectively super-user privileges and apply to all databases. That means if you grant Update rights to a user, that user can now update all tables in all databases, *including the mysql database*. Such a user is effectively a MySQL administrator and can change any and all information including root passwords etc. So be very careful about which `user` records have database privileges. You wouldn't normally have an application logging in as the root user, for instance.

In most cases users other than the root user will have all their privileges at the user level set to N. When MySQL receives an SQL statement and can't find a suitable permission at the user level, it next looks for permission in the `db` table.

The `db` table specifies privileges for a specific database. It's field structure is as follows:

Field Name	Sample Data	Purpose
Host	192.168.100.1	Computer the user is connecting from
Db	testdb	Database
User	rocky	User
Select_priv	Y	Select records
Insert_priv	Y	Add records
Update_priv	Y	Update records
Delete_priv	Y	Delete records
Create_priv	Y	Create tables
Drop_priv	Y	Drop tables
Grant_priv	Y	Give others same privileges as self
References_priv	Y	Not implemented?

Index_priv	Y	Create/drop indexes
Alter_priv	Y	Execute alter_table

As you can see db is similar to user except that it stores a database name instead of a user password, and it lacks the administrative privilege fields.

Messing With The Data

As installed, MySQL is accessible from the machine on which it is installed. To access it from another machine (i.e. one running a Clarion app) you need to set up some permissions. You can do this with the mysql client.

On the Linux box log in as root and type

```
mysql
```

or

```
mysql -p
```

if you've assigned a root password.

You'll want to create a database:

```
create database testdb;
```

You'll get a response something like:

```
Query OK, 1 row affected (0.01 sec)
```

When you start the MySQL client no database is selected. At the mysql> prompt type:

```
use mysql;
```

Actually the ; isn't necessary for this command but as you'll generally need the trailing ; for SQL statements it's a good habit to get into. MySQL will tell you the database has changed. You can now update mysql and create the necessary records for your Windows machine to connect to the database you just created.

Let's say your Linux box and your Windows box are both on a local network, and the Windows box's IP address is 192.168.100.9. First, create a user record:

```
insert into user (host,user,password)
  values('192.168.100.9','rocky',
        password('squirrel'));
```

This insert statement specifies only three fields in the user file. All the remaining fields default to N, which again is a good idea unless you want to give this user administrative privileges. Note the use of the password function to do the encryption.

To add a db record:

```
insert into db (host,db,user,Select_priv,Insert_priv,
    Update_priv,Delete_priv, Create_priv,Drop_priv)
values ('192.168.1\00.9','testdb','rocky',
    'Y','Y','Y','Y','Y');
```

This record will allow the user rocky access to testdb when connecting from the specified host IP address (you can also use a host name). MySQL keeps the privilege settings in memory, so you have to tell it to reload this information from the database. In later releases you can use the FLUSH PRIVILEGES command, or you can use the quit command to exit mysql, then run:

```
mysqladmin reload
```

The following tables show the mysqladmin syntax:

Option	Description
-#, --debug=...	Output debug log. Often this is 'd:t:o,filename'
-f, --force	Don't ask for confirmation on drop database; with multiple commands, continue even if an error occurs
-?, --help	Display this help and exit
-C, --compress	Use compression in server/client protocol
-h, --host=#	Connect to host
-p, --password[=...]	Password to use when connecting to server If password is not given it's asked from the tty
-P --port=...	Port number to use for connection
-i, --sleep=sec	Execute commands again and again with a sleep between

<code>-r, --relative</code>	Show difference between current and previous values when used with <code>-i</code> . Currently works only with extended-status
<code>-s, --silent</code>	Silently exit if one can't connect to server
<code>-S, --socket=...</code>	Socket file to use for connection
<code>-t, --timeout=...</code>	Timeout for connection to the mysqld server
<code>-u, --user=#</code>	User for login if not current user
<code>-V, --version</code>	Output version information and exit
<code>-w, --wait[=retries]</code>	Wait and retry if connection is down

Command	Description
<code>create database name</code>	Create a new database
<code>drop database name</code>	Delete a database and all its tables
<code>extended-status</code>	Gives an extended status message from the server
<code>flush-hosts</code>	Flush all cached hosts
<code>flush-logs</code>	Flush all logs
<code>flush-status</code>	Clear status variables
<code>flush-tables</code>	Flush all tables
<code>flush-privileges</code>	Reload grant tables (same as reload)
<code>kill id,id,...</code>	Kill mysql threads
<code>password new-password</code>	Change old password to new-password
<code>ping</code>	Check if mysqld is alive
<code>processlist</code>	Show list of active threads in server

reload	Reload grant tables
refresh	Flush all tables and close and open logfile
shutdown	Take server down
status	Gives a short status message from the server
variables	Prints variables available
version	Get version info from server

MySQL should now be fully operational and ready to accept a connection from your Clarion app on a Windows machine.

Using ODBC

Your Clarion application will use the TopSpeed ODBC driver to communicate with the ODBC layer, which will connect to the MyODBC driver, which will talk to the MySQL server on your Linux box, over TCP/IP.

From the MySQL downloads page (www.mysql.com/downloads), or one of its mirrors, click on **Downloads for MyODBC**. Choose the full setup install for the version of Windows you are running. Unzip the archive to a temp directory and run setup.exe. You'll see the window shown in Figure 3.

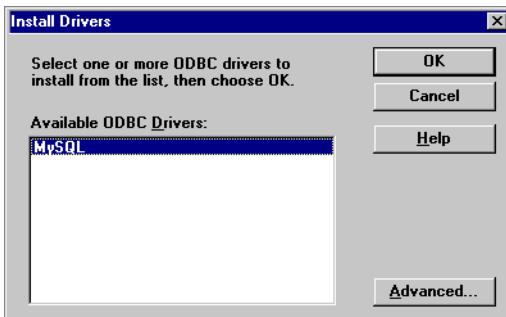
Figure 3: The ODBC setup program



Open Source SQL

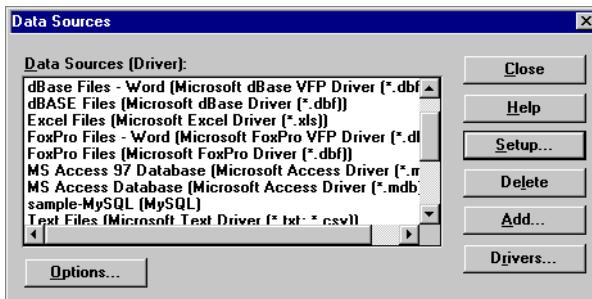
Click on **Continue** and you'll be presented with a list of ODBC drivers to install, which in this case is just MySQL (see Figure 4). Select the driver and continue.

Figure 4: Select the MySQL driver



The ODBC setup program will add the MyODBC drivers to your system and register them with the ODBC administrator (see Figure 5). Finally, the setup program will display the administrator. You can set up a MySQL data source here, or you can do it later by choosing ODBC Data Sources from the Control Panel.

Figure 5: The ODBC Administrator

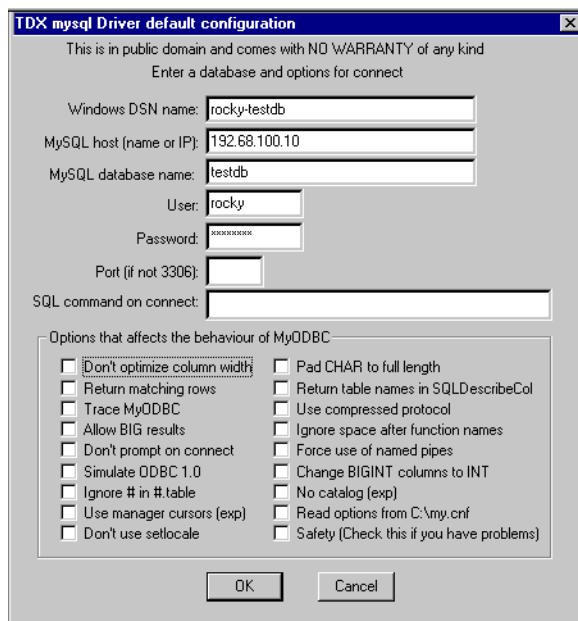


See the next chapter ("MySQL/MyODBC Notes," p. 241) for more on setting up MyODBC, including some important information on driver strings.

Data Sources

You'll need to set up at least one MySQL data source. Earlier I created a database called testdb. Figure 6 shows the configuration options for the MySQL ODBC driver.

Figure 6: Configuring a MySQL ODBC connection



Most of the fields are self-explanatory. The DSN name is the name you will need to enter in the **Owner** field of the Clarion file definition, and the MySQL host is the machine running the MySQL server. I've used an IP address here, but you can also enter the machine name if your DNS is able to resolve it. To find out if the name is recognized, you can try to ping the host, or you can try:

```
nslookup hostname
```

from the DOS prompt.

Enter the name of the database you're going to be working with, the user you will log in as, and the password. And then you're probably done. MySQL communicates with outside clients via port 3306 unless the server has been told otherwise, so presumably you'll know if that's the case.

Importing Tables

If you have existing tables in your MySQL database, you can import them into your dictionary. Open the dictionary, choose **File|Import File**. Choose the ODBC driver, and when you get the Data Sources window, click on the **32 bit** tab and choose the MySQL source you just created. You should see a list of tables on the MySQL server.

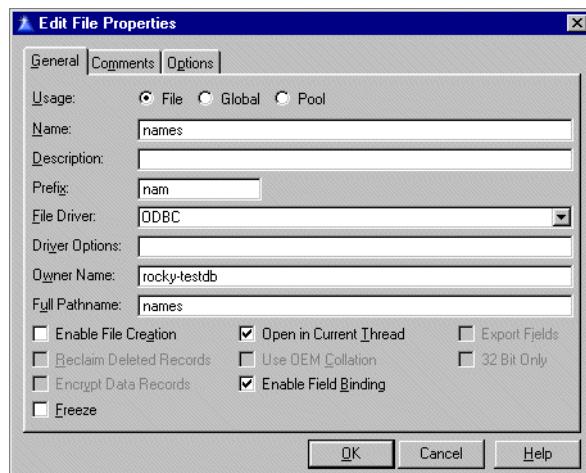
It's possible that you'll be unable to connect to the MySQL data source. Most likely this will be because you don't have sufficient permission to access the MySQL server. Remember that you need a valid user record as well as a valid db record in the mysql administration database. Also be sure you issue a `mysqladmin reload` command to make changes to the administration database take effect.

If you still can't connect, verify that you're using the correct user ID and password. If all your data looks right, then it may be that you have a firewall between you and the server which is blocking port 3306, or there's some problem with TCP/IP.

Assuming you have succeeded in connecting, you can choose a table to import. Unfortunately you can't do a multi-file import as this requires the synchronizer, which doesn't work with the ODBC driver.

Figure 7 shows the properties for a table imported from a MySQL server. The owner is set to the ODBC DSN, and the pathname is the name of the table.

Figure 7: File properties for a MySQL table



Creating Tables

If you don't have an existing MySQL database to work with, you'll need to create some tables in your own database. You can do this with SQL statements from the mysql client program on the Linux box (remember to issue a use command from within mysql before issuing table creation/modification commands). If you have C5 Enterprise Edition you can use Data Modeler to create SQL scripts for you, but these will need some massaging before they can be used in MySQL. And I suspect there are some SQL utility templates out there that will do the job also, with minor changes.

Another, often easier, approach is to create your tables using one of the commercial MySQL administration tools, such as my personal favorite, MySQL Manager, from EMS (<http://ems-hitech.com/mymanager>). EMS also sells database managers for MS SQL, PostgreSQL, and Interbase, as well as a number of useful utilities. A free trial version of MySQL Manager is available for download.

Some Basics

Each of your MySQL tables should have a primary key on a field of type INT. Just create a new field in the file, call it ID or something suitable, and check the **Primary Key** and **Autoincrement** options.

MySQL will take care of the autoincrementing for you if you choose that option, in which case you must not set the autoincrement option for this key in your Clarion dictionary. One important difference between MySQL and Clarion autoincrementing is that in Clarion the autoinc value is known while you're adding the record, but if it's done on the server it isn't. This will cause problems if you have an update form where you can add child records while adding a parent record.

If you're changing MySQL and dictionary definitions in parallel and you get invalid record declaration errors when you try to run your app, import the table(s) into a different dictionary, verify that everything works with a quickie wizarded application, and compare definitions.

Summary

In my experience the biggest obstacles to using MySQL are getting the permissions right and finding a working version of the driver.

With those problems solved I've found MySQL to be as fast and reliable as its reputation suggests it should be.

- Download MyODBC (http://www.mysql.com/download_myodbc.html)

MySQL/MyODBC Notes

by David Harms

Since I wrote the previous chapter on MySQL a few things have changed, and I've had some additional feedback from other Clarion developers using MySQL. I also need to make a few corrections to the comments I made about MySQL during my presentation at ETC 2000.

Errata

As I recall, I mentioned at ETC that the maximum database size in MySQL is two or four gigabytes, depending on the operating system. This is incorrect. The maximum *table* (not database) size defaults to four gigabytes, and there is no internal limit on the number of tables you can have in a database (although you can expect some performance degradation if you have thousands and thousands of tables in one directory).

The maximum table size is actually dependent on two things: the underlying MySQL file library system, and the operating system's file system. With the most recent builds (3.23 or later) MySQL has introduced the option of using the new MyISAM file system, which is based on the ISAM library which is at the core of MySQL. Yes, this is an SQL system, but

even SQL databases need to use some sort of lower level file library to actually store the data on disk.

In general, table size is limited by the operating system. On Linux-Intel, you're looking at two gigabytes per table, or four if you use Reiserfs (<http://devlinux.com/projects/reiserfs/>). On Linux-Alpha, however, you can have tables of up to *eight million terabytes!* That's 8,000,000,000,000,000 bytes, or four hundred million 20 gig drives. Big enough for ya?

If you're stuck at that two gig limit, you can use the merge library to treat multiple, identical tables as one table, but indexing support for merged tables has not yet been implemented.

Comparisons

If you'd like to compare MySQL features with other databases, have a look at <http://dev.mysql.com/tech-resources/crash-me.php>. This URL generates comparisons of MySQL (current alpha), MySQL (latest stable build), Access 2000, Adabas, IBM DB2, EMPRESS, Informix, Interbase, MS SQL, MIMER, mSQL, Oracle, PostgreSQL, SOLID Server, and Sybase. Areas tested include crashability, ANSI SQL 92 types and functions, ODBC 3.0 types and functions, other types and functions, constraints and type modifiers, order by, group by, join types, string handling, various kinds of limits, and more.

Miscellaneous

MySQL now supports transactions and replication, although this code is still in beta form (as of June 2000). Transactions can be enabled on a per table basis, which means that there's no speed penalty for tables that don't need transactions. Replication is of a fairly rudimentary sort. One server is a master, and one or more other servers are slaves. All updates to the master server are propagated to the slaves automatically. You'll need MySQL version 3.23.15 or higher. You can also set up two servers such that both are master and slave, and writes to either are replicated. So far this can only be done with two servers.

Fast gets faster: you can now create in-memory temporary tables. These are useful for things like lookups where the content of the table seldom changes.

I'm not sure if anyone asked me this, but I've recently learned that there are about a million MySQL server installations worldwide.

MyODBC

To use MySQL with Clarion you need the MyODBC driver, although the June 2000 SoftVelocity newsletter indicates that native support for some Linux databases is being considered, and I would hope that includes MySQL. There is a problem with the current releases of MyODBC involving the sequence of SQL statements. Normally, the Clarion ODBC driver issues the following sequence of statements:

```
SQLPrepare  
SetStmtOptions  
SQLBindCol  
SQLBindparameters  
SQLExecute
```

MySQL, however, expects the following:

```
SQLPrepare  
SetStmtOptions  
SQLBindparameters  
SQLExecute  
SQLBindCol
```

In the alpha builds following the release of C5.5 B2, the ODBC driver can be set to a modified statement order with the following driver strings:

Driver String	Statements
/BINDCOLORDER=1	SQLPrepare SetStmtOptions SQLBindparameters SQLBindCol SQLExecute
/BINDCOLORDER=2	SQLPrepare SetStmtOptions SQLBindparameters SQLExecute SQLBindCol

The /BINDCOLORDER=2 setting creates the statement order expected by MySQL. This may or may not still be required by later releases of the driver.

Other Driver Strings

If your browse involves more than one level of child records you'll most likely get an SQL error on the JOIN statement, because MySQL only has limited support for ODBC-style JOINS. The solution is to use the driver string:

```
/JOINTYPE=DB2
```

to force a style of JOIN that MySQL can cope with. You may also need to use the driver string

```
/USEINNERJOIN=FALSE
```

Thanks to Jim Gambon for pointing this out. Again, there have been some changes to inner join support in recent releases of MySQL so it's possible you won't need this.

With newer releases of MySQL and MyODBC, you may need to use the VERIFYVIASELECT driver string; without it some tables may be inaccessible:

```
/VERIFYVIASELECT=TRUE
```

Other Resources

You may also want to look at the following Icetips document on connecting to MySQL with Clarion, from a remote machine:

http://www.icetips.com/files/connecting_to_sql_databases_online.pdf

Summary

And the last word goes to Jim Gambon, who writes:

MySQL, as it is today, and for the foreseeable future, is not a Relational Database Management System. It is simply an ISAM file Database with table and key files similar to Clarion DAT files. Driving these tables is streamlined SQL engine for "atomic" database actions like "SELECT" "UPDATE", etc. The tables in a MySQL Database do not know how they relate to each other. Foreign Keys are not supported to restrict or cascade changes. Changes do not "trigger" any other action within the Database to keep things like "Inventory:Quantity_on_Hand" fields correct.

"Put it on the shelf." That is what Andy Stapleton said at the ETC 2000 conference when I told him that MySQL did not currently have "triggers" or "stored procedures". I understand exactly why he would say that. Having the back end database as robust as possible before the teams of programmers and web developers start beating on it saves him (and any other Database Administrator) headaches and sleepless nights.

So why use MySQL? (And don't say "because it's free.") MySQL has no RI on the server, but it is a very fast ISAM file system. That means that the RI will have to be on the client. But the client program is our Clarion created app, and we are blessed with an abundance of data integrity tools to use in development:

Foreign Key Constraints? Tell the dictionary about the restrictions and the templates take care of the rest. Stored Procedures? No, not in the server database (yet), but the ABC classes provide fine places to embed code that needs to know about the structure of a table (Filemanager) and its relations (RelationManager).

Clarion 5.5 now has File System CallBacks to be used as a sort of "client side trigger". Right before, and right after, a call to the file drivers (on ADD, PUT, SET, etc.) a procedure you define can be called. Do what you need in the Callback, and no matter where in the program the ADD or PUT happens, the proper "Quantity on Hand" update will happen. A future MySQL version will have simple stored procedures, but no triggers. You should still be able to use the Clarion CallBack (on the client) to run the MySQL stored procedure (on the server).

So this means that the same Template created RI code that works in our programs with TPS files will (or should) work when we switch to MySQL. MySQL, therefore, becomes the perfect first step away from shared ISAM files (like Clarion DAT or TPS files) in the small development shops where Clarion is often used.

So, if you have to share database maintenance chores with other groups of programmers, then definitely "put it on the shelf." You need something with more built-in data integrity. But, if you are in charge of the programming AND the database, MySQL seems to be a fine replacement for shared-file databases like TPS.

MySQL: INNODB TABLES AND TRANSACTIONS

by David Harms

MySQL is a fast, reliable, multi-platform, and free (under the GNU General Public License) SQL server. Developed specifically for speed and reliability, MySQL is missing some of the common SQL server features such as sub-selects, stored procedures, and triggers/server-side referential integrity. That may seem like a pretty long laundry list. But as Jim Gammon pointed out in the conclusion to “MySQL/MyODBC Notes,” p. 241, MySQL makes a more than adequate replacement for TPS files, as you can easily get Clarion to handle the critical RI issues.

Note: Sub-selects are slated for MySQL release 4.1.

Atomic operations

Among the missing features, the lack of transaction support was a more serious problem. MySQL in its native state uses atomic operations instead of transactions. An atomic

operation is any single table update. It's a bit like a mini-transaction: either the table update happens or it doesn't.

According to the MySQL documentation, atomic operations can be three to five times faster than transactions. Atomic operations also guarantee that you won't get any dirty reads (a read of data in the process of being changed). The downsides are

- updates are on a per-table basis, so you can't group a bunch of updates together under one atomic operation,
- there's no way to explicitly roll back an atomic operation (although you could manually undo whatever you just did), and
- because atomic operations prevent dirty reads, a table update will block a `SELECT` on the same table, which can have performance implications.

If this is how MySQL handles updates, then how do you get it to do transactions? Easy - just don't use MySQL tables.

Let me explain that. MySQL is a SQL DBMS, or database management system. Your application will (most likely) talk to MySQL using the MyODBC driver. However your app communicates with the server, MySQL receives those SQL statements, queries or updates data accordingly, and returns data if necessary. By default, MySQL calls its own low-level data access library to get at the data. But that isn't your only option.

Table types

Just as one Clarion application can use multiple file/table types via different file drivers, so a MySQL server can work with multiple table types by plugging in the appropriate back-end library. As of MySQL 3.23, there are seven table types (that I know of) available: MyISAM, MERGE, ISAM, HEAP, BDB, InnoDB, and Gemini. Each of these table types has its own advantages and disadvantages, and several originate with companies other than MySQL AB.

MyISAM tables

If you issue a `CREATE TABLE` statement in MySQL, without a `TYPE` attribute, MySQL will create a MyISAM table. MyISAM is an update to the original MySQL ISAM table type. MyISAM is, if you like, MySQL's native table type. Big (63 bit) files are supported on 64 bit operating systems, and all data is stored low byte first, which makes the data machine-independent.

ISAM tables

ISAM tables are the original MySQL “native” B-tree ISAM format tables. Data is stored in machine-dependent format. This table type is deprecated (replaced by MyISAM) and is expected to disappear entirely before much longer.

MERGE tables

New in MySQL 3.23.25, MERGE tables (also known as MRG_MyISAM tables) are collections of *identical* MyISAM tables which you can use as a single table. Keys and columns must match among the merged tables. Why would you want to do this? Perhaps for a log file, where you can split the data by month and get better performance when looking at just one table, yet still search all tables as one if you need to. You can also selectively create merged tables without altering the source table definitions. If you have very large tables, you can locate different tables on different disks, and treat them as a single table. As well, a merged table over a single table is effectively an alias. I haven’t tested MERGE yet, but I’m considering implementing the ClarionMag server log as a set of merged tables.

HEAP tables

HEAP tables are very fast tables created in memory. That means if you lose power, you lose the data, but you get terrific speed. HEAP tables are commonly used for temporary tables.

Gemini tables

The Gemini table types are available in NuSphere’s release (<http://www.nusphere.com/products/mysqladv.htm>) of MySQL. Its worth noting that there have been legal disputes (http://www.mysql.com/news-and-events/news/article_75.html) between NuSphere and MySQL AB. I don’t have a lot to say about Gemini because I don’t own a copy - NuSphere’s release of MySQL with Gemini is a commercial product.

BDB tables

The BDB, or BerkeleyDB table type is another of the MySQL table types which do not originate with MySQL AB. The Berkeley database system was created by Dr. Margo Seltzer and Keith Bostic in the early 90s; Selzter and Bostic later founded SleepyCat (<http://www.sleepycat.com>) Software.

A patched version of BDB ships with the MySQL source distribution; you can’t use the non-patched BDB with MySQL. Among other things, BDB tables provide transaction support, and are slightly larger and slower than MyISAM tables.

InnoDB tables

As with the BDB table type, the InnoDB table type does not originate with MySQL AB. InnoDB (<http://www.innodb.com>) is the brainchild of Heikki Tuuri, a Finnish developer who holds a PhD in mathematical logic from the University of Helsinki. Tuuri began development of InnoDB in 1994, and created the MySQL interface in collaboration with MySQL AB, between September 2000 and March 2001.

A relatively new table type for MySQL, InnoDB has been generating a fair bit of excitement among MySQL users. It offers a number of important enhancements, such as transactions, row level locking (as opposed to MyISAM table locking/atomic operations), and high performance on large volumes of data. InnoDB tables are kept in a tablespace, rather than in individual files for each table. InnoDB tables do take up quite a bit more space than MyISAM tables.

Choosing a table type

Although I've been relatively happy with MySQL's speed (I use MySQL primarily as the database underpinning *Clarion Magazine*), I have noticed occasional performance problems which I suspected were caused by updates blocking SELECT statements. This is a predictable situation given MySQL's use of atomic operations to maintain data integrity. InnoDB tables, on the other hand, lock at the row level, not the table level, and provide Oracle-style consistent reads. The InnoDB manual puts it this way:

A consistent read means that InnoDB uses its multiversioning to present to a query a snapshot of the database at a point in time. The query will see the changes made by exactly those transactions that committed before that point of time, and no changes made by later or uncommitted transactions. The exception to this rule is that the query will see the changes made by the transaction itself which issues the query.

(The manual also points out that consistent reads aren't always desirable; if your query needs to be absolutely sure that the records it sees have not been removed or altered since the query began, you can do a locking read by appending the phrase `LOCK IN SHARE MODE` to the `SELECT` statement.)

Consistent reads and row level locks certainly looked like useful improvements to my server installation, so I decided to go ahead and give InnoDB tables a try.

Installing an InnoDB-capable MySQL

My first step was to upgrade my existing MySQL server to a binary version (as opposed to getting, and building, the source) that supports InnoDB tables. This version is called MySQL-Max, and like “regular” MySQL, is freely available under the GPL. Just go to www.mysql.com (<http://www.mysql.com/>) and look for the Versions box on the home page. You’ll see a link to the current releases of key MySQL products, including MySQL-Max. The version I used for this article is 3.23.42.

Upgrading my MySQL installation proved quite simple. First, I unpacked the MySQL-Max distribution under the /usr/local directory. In my case, the command was:

```
tar -zxvf /tmp/mysql-max*
```

I used the wildcard in the gzipped tar filename to save myself some typing. The full command would be

```
tar -zxvf /tmp/mysql-max-3.23.42-pc-linux-gnu-i686.tar.gz
```

assuming the gzipped tar is in the /tmp directory and you’re already in the /usr/local directory. This command unpacked all of the mysql-max binary distribution files into the /usr/local/mysql-max-3.23.42-pc-linux-gnu-i686 directory.

The next step was to shut down the currently running MySQL server and switch everything to the new version. I executed this command:

```
mysqladmin shutdown -u root -p
```

Because I’d previously set a root password for mysql (always a good idea), I had to supply the root user id and the -p parameter. I could have supplied the actual password after the - parameter, but it’s better to let mysqladmin (and any other utilities) prompt you, as this is more secure.

With the MySQL server shut down I was now in a position to switch servers. When I first installed MySQL on this particular server, the files all went in the /usr/local/mysql-3.23.35-pc-linux-gnu-1686 directory. In keeping with the install instructions, I created a symbolic link to that directory, called mysql. That meant I could refer to the /usr/local/mysql-3.23.35-pc-linux-gnu-1686 directory as simply /usr/local/mysql. So the first step to the migration was to delete the old link:

```
cd /usr/local  
rm mysql
```

and add a new one:

```
ln -s /usr/local/mysql-max-3.23.42 -pc-linux-gnu-i686 mysql
```

If this was a new installation, I'd next run the `mysql_install_db` script to create the default `mysql` and `test` databases, but because I already have a database set up this isn't necessary, or desirable. Instead, I simply copied all the database files to their new location:

```
cp /usr/local/mysql-3.23.35-pc-linux-gnu-1686/data/*  
/usr/local/mysql/data -Rvf
```

The `-Rvf` options, respectively, tell `cp` to copy recursively, display file names, and force the copy where necessary. Actually I could also have moved the database to the new location, but I wanted to keep a backup copy in case something went wrong.

I started up `mysqld` with this command:

```
/usr/local/bin/safe_mysqld -- user=mysql &
```

And `mysql` loaded up, and everything worked! Wonderful. Just to make sure everything was copacetic, I rebooted the server. Unfortunately, the MySQL server did *not* start up on the reboot. After a little investigation I discovered that I'd started up the server with code in my `/etc/rc.d/rc.local` file, as follows:

```
/usr/local/mysql/support-files/mysql.server start
```

A quick check of that particular file confirmed what I suspected: it wasn't marked as executable, as it was part of the new install. So I ran this command:

```
chmod +x mysql.server
```

from the `/usr/local/mysql/support-files` directory, and tried again. This time when the server rebooted, the MySQL server ran automatically. To confirm that I had the right version of MySQL running, I started the command line interface (CLI), and executed this command:

```
mysql> select version();  
+-----+  
| version() |  
+-----+  
| 3.23.42-max |  
+-----+  
1 row in set (0.00 sec)
```

The next step was to create an InnoDB database. Like Oracle tables, InnoDB tables live inside a tablespace, which is one or more physical files. To create the tablespace you add some appropriate settings to the `my.cnf` configuration file, and then you start the MySQL server. I'm running this particular server on a RedHat 7.0 Linux box, and I used the following settings in my `/etc/my.cnf` file:

```
[mysqld]  
innodb_data_file_path = ibdata/ibdata1:1G  
innodb_data_home_dir = /  
set-variable = innodb_mirrored_log_groups=1
```

```
innodb_log_group_home_dir = /iblogs
set-variable = innodb_log_files_in_group=3
set-variable = innodb_log_file_size=50M
set-variable = innodb_log_buffer_size=8M
innodb_flush_log_at_trx_commit=1
innodb_log_arch_dir = /iblogs
innodb_log_archive=0
set-variable = innodb_buffer_pool_size=400M
set-variable = innodb_additional_mem_pool_size=20M
set-variable = innodb_file_io_threads=4
set-variable = innodb_lock_wait_timeout=50
```

I simply copied these settings from the MySQL online documentation, with a few exceptions. My `innodb_data_file_path` setting only specifies a single file for the tablespace, one gigabyte in size, while the example showed two tablespaces. I also changed the `innodb_log_group_home_dir` and `innodb_log_arch_dir` settings to suit my system, but I made the mistake of making these different directories. When I fired up MySQL, I saw the following in my error log (in `/usr/local/mysql/data`):

```
011004 12:54:43 mysqld started
InnoDB: Error: you must set the log group

      home dir in my.cnf the InnoDB: same as log arch dir.
011004 12:54:44 Can't init databases
011004 12:54:44 mysqld ended
```

As it turns out, in this release of MySQL you have to specify the same directory for the log and log archive files. I corrected this problem, deleted the `/ibdata/ibdata1` file (per the instructions regarding failed InnoDB initialization) and tried again. No joy - shortly after beginning to write the data file, InnoDB bailed on an operating system error.

```
011004 13:11:43 mysqld started
InnoDB: The first specified data file

      /ibdata/ibdata1 did not exist:
InnoDB: a new database to be created!
InnoDB: Setting file /ibdata/ibdata1

      size to 1073741824
InnoDB: Database physically writes the

      file full: wait...
InnoDB: operating system error number

      22 in a file operation.
InnoDB: Cannot continue operation.
011004 13:11:54 mysqld ended
```

I couldn't find any documentation explaining the nature of the error, so I deleted the one data file again, and this time rebooted. When MySQL came up on the reboot, it printed the following information to the log:

```
011004 13:17:53 mysqld started
InnoDB: The first specified data file
```

```
    /ibdata/ibdata1 did not exist:  
InnoDB: a new database to be created!  
InnoDB: Setting file /ibdata/ibdata1  
  
    size to 1073741824  
InnoDB: Database physically writes the  
  
    file full: wait...  
InnoDB: Log file /iblogs/ib_logfile0 did  
  
    not exist: new to be created  
InnoDB: Setting log file /iblogs/ib_logfile0  
  
    size to 52428800  
InnoDB: Log file /iblogs/ib_logfile1 did not  
  
    exist: new to be created  
InnoDB: Setting log file /iblogs/ib_logfile1  
  
    size to 52428800  
InnoDB: Log file /iblogs/ib_logfile2 did not  
  
    exist: new to be created  
InnoDB: Setting log file /iblogs/ib_logfile2  
  
    size to 52428800  
InnoDB: Doublewrite buffer not found:  
  
    creating new  
InnoDB: Doublewrite buffer created  
011004 13:19:58 InnoDB: Started  
/usr/local/mysql/bin/mysqld: ready for connections
```

Success! I had an InnoDB database running under MySQL, and I could begin creating InnoDB tables.

Creating the tables

For some time I've been using the 2.50.33 release of the MyODBC driver. I decided it was time to get current, and so I went to the MySQL home page (<http://www.mysql.com>) where I discovered that I was six revs behind the times! In fact, support for transactions was only added to MyODBC in release 2.50.37, so I didn't have a choice - I needed the updated driver to use InnoDB transactions. As I already had gone through the full MyODBC install in the past, I downloaded the "DLL only" install.

Note: If you're running Clarion 5.0 or earlier, you may not be able to use the current releases of the MyODBC driver (and therefore you cannot use transactions with InnoDB tables). The problem is that the Clarion ODBC driver uses a different sequence of ODBC statements than that expected by MyODBC.

In Clarion 5.5x, you can tell the driver to use a MyODBC-compatible sequence by specifying the /BINDCOLORDER=2 driver string.

To install just the new MyODBC DLL I extracted the zip contents to a temp directory, and then ran `INSTALL.BAT`. Actually I thought it a bit odd that the batch file was able to copy the files over, as I'd been using the MyODBC driver just prior to the update and expected the DLL would be marked as still in use. I checked the ODBC administrator and it reported that the DLL was still the old version. So I did what I usually have to do in a situation like this: I rebooted and tried again. The batch file ran again with no errors and still the ODBC administrator reported the old version.

At that point I checked the batch file and found it was copying files into the Windows `system` directory; I'm running NT, and those files needed to go into the `system32` directory. A quick search and replace on the batch file, one more attempt, and MyODBC was finally updated.

If you haven't used MyODBC before then you will want to download the full install - this will set up MyODBC in the ODBC administrator; as far as I know, that install doesn't suffer from the same problem as the DLL update batch file.

Creating tables

Once you have MyODBC installed, you're ready to create the tables. There are a number of ways you can go about this, including using Roberto Artigas' templates for converting from TPS to MySQL (Roberto regularly posts releases to the SoftVelocity `topic.thirdparty` newsgroup).

In this chapter I'll show how to create a few simple tables from scratch. These tables, called `Parents` and `Children`, will rather obviously contain parent and child records, in a one to many relationship. You'll also need a third table to support server-side autoincrementing (more on that later). Listing 1 shows the SQL code used to create a new database called `transact` and populate it with the three tables:

```
CREATE DATABASE transact;
USE transact;

CREATE TABLE Parents(
ParentID INT NOT NULL AUTO_INCREMENT,
LastModified TIMESTAMP,
PRIMARY KEY (ParentID)
TYPE=INNODB;

CREATE TABLE Children(
ChildID INT NOT NULL AUTO_INCREMENT,
ParentID INT NOT NULL,
SomeWeirdField INT DEFAULT 0,
```

```
LastModified TIMESTAMP,  
PRIMARY KEY (ChildID),  
INDEX ChildrenIdx _ ParentID (ParentID))  
TYPE=INNODB;  
  
CREATE TABLE lastinsert (ID INT) TYPE=INNODB;
```

After you create the tables, you need to tell MySQL to allow your application to access the database. Assuming the server and the application are running on the same machine (called localhost), and the user id and password are both transtest, the following code will do the trick :

```
GRANT ALL ON transact.* TO transtest@localhost  
IDENTIFIED BY 'transtest';  
FLUSH PRIVILEGES;
```

The GRANT statement will create entries in the user and db tables in the mysql database; MySQL uses these tables (and optionally several others) to control who do what to which tables and columns. By issuing a GRANT ALL you are allowing this user full control over the transact database; in reality you'd probably want to restrict rights a bit, particularly since GRANT ALL allows the user to read physical, non-MySQL files on the server!

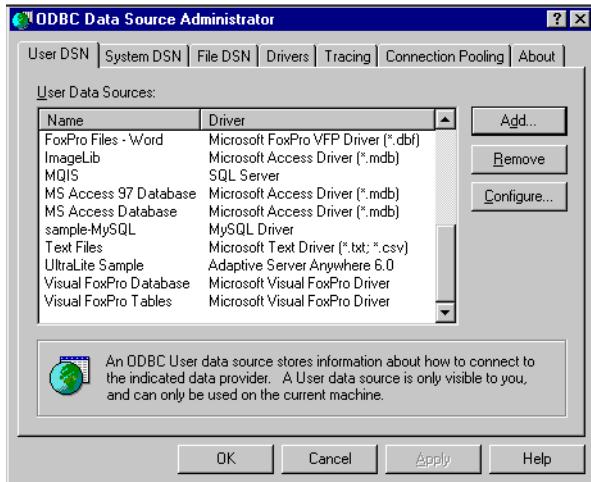
Setting up a data source

Let's say you have MyODBC installed, and you now have some tables. The easiest way to build an application around those tables is to just import them into your application, and to

MySQL: InnoDB Tables And Transactions

do that you need to define a data source. Go to the Control Panel and open the ODBC Administrator. On the **User DSN** tab (shown in Figure 1) click on **Add**.

Figure 1: The ODBC Administrator User DSN tab



From the **New Data Source** window (Figure 2) select the MySQL driver. As you can see, I have two entries. I don't know why - maybe it was that botched installation attempt.

Figure 2: Selecting the MySQL driver

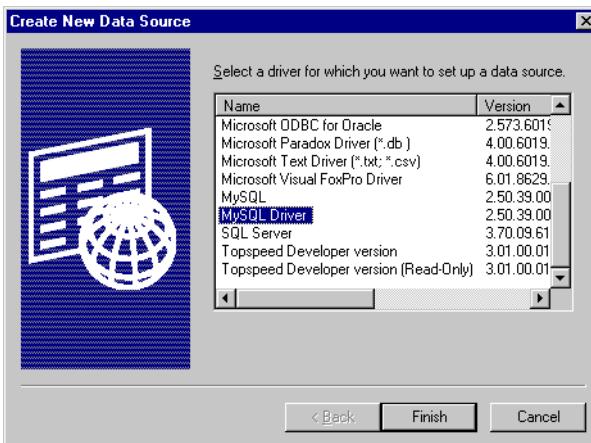
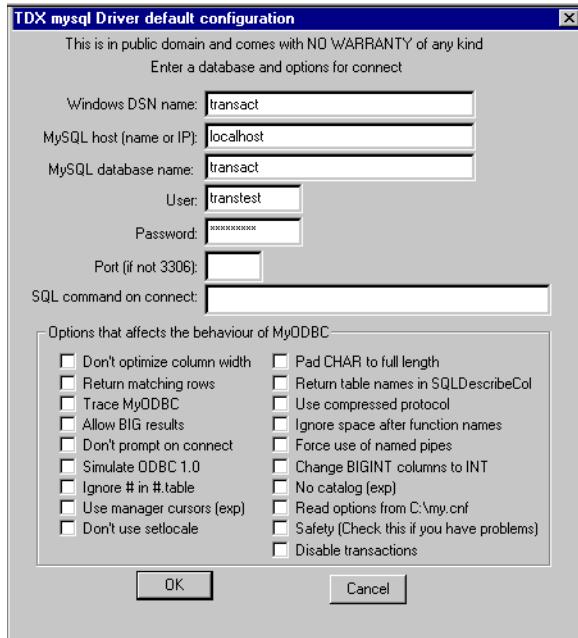


Figure 3 shows the configuration settings for the MySQL driver. The **Windows DSN name** is the name your application will use to access the database. The host is the name or IP address of the computer that is running the MySQL server.

Figure 3: MySQL configuration settings



Importing the table definitions

Now you're ready to import the table definitions into your Clarion application. Create a new dictionary, and with that dictionary open select **File|Import Table**. Choose the ODBC driver, and when you're presented with a list of data source names, choose the one you created for your MySQL database (in this example, it's the DSN called `transact`). Click on **Next** and you should see the list of available tables. Go through the import procedure three times until you have all three of the `transact` tables in your dictionary.

I have noticed occasional problems with importing MySQL tables, and the `Children` table is a typical example. When you import this table you have to examine the list of field names to make sure that all have imported correctly. On my system, the `ParentID` field does not. I have to correct the field name (its an unprintable character after import), and I have to go to the **Attributes** tab and set the **External name** to `ParentID` as well. If the external name of each field doesn't match the name of the corresponding table column,

you'll get an invalid record declaration error when you attempt to open the table with your Clarion app.

There's one other dictionary change to make. For at least one of the tables set the following in the **Driver Options** field:

```
/BINDCOLORDER=2
```

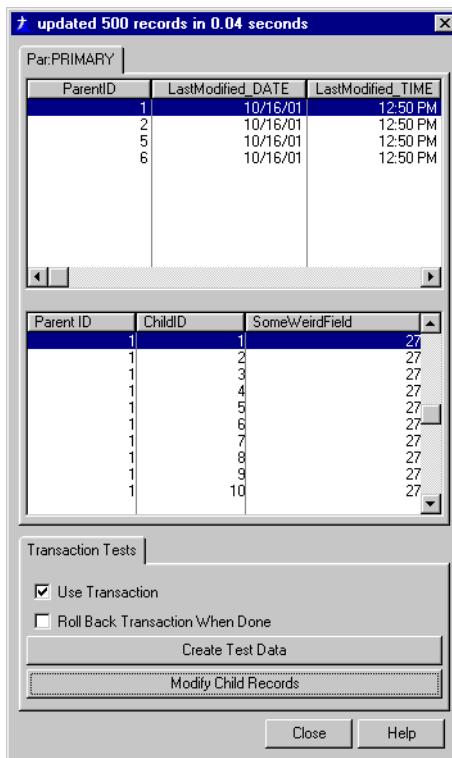
This statement tells the Clarion ODBC driver to use a sequence of ODBC commands that is compatible with the MyODBC driver.

Now you can wizard up an application based on the dictionary and start playing with the data. (Or you can just download the example app at the end of this chapter.)

Using transactions

Figure 4 shows the example application I used to test transactions with MySQL. Keep in mind that this application just demonstrates that transactions *work*. I'm not making any pretense of benchmarking here, although I have added some very rudimentary timings of basic operations.

Figure 4: The transaction test application



The example application lets you create some parent and child data inside a transaction. It also lets you decide whether you want to place that data creation inside a transaction, and if so, whether you want the transaction to complete successfully, or to be rolled back. Here's the data creation code:

```

starttime = clock()
if UseTransaction then logout(1,Parents,Children).
setcursor(cursor:wait)
setnull(Par:ParentID)
if access:Parents.insert() = level:benign
  lastinsert{PROP:sql} |
    = 'delete from lastinsert'
  lastinsert{PROP:sql} |
    = 'insert into lastinsert (id) ' |
      & 'values (last_insert_id())'
lastinsert{PROP:sql} = 'select * from lastinsert'
access:lastinsert.next()
ParentID = las:ID

```

```

y = random(1,100)
loop x = 1 to maxRecs
    !access:Children.PrimeAutoInc()
    Chi:ParentID = ParentID
    Chi:SomeWeirdField = y
    setnull(Chi:ChildID)
    !0{PROP:text} |
        = 'creating entry ' & x & ' of ' & maxRecs
    access:Children.Insert()
end
end
if useTransaction
    if AbortTransaction
        rollback()
    else
        commit()
    end
end
endtime = clock()
setcursor()
0{PROP:text} = 'processed ' & maxRecs |
    & ' records in ' & (endtime - starttime)/100 |
    & ' seconds'
ThisWindow.Reset(true)

```

And here's the data modification code:

```

starttime = clock()
if UseTransaction then logout(1,Children).
setcursor(cursor:wait)
y = random(1,100)
Children{PROP:sql} |
    = 'update Children set SomeWeirdField=' & y |
    & ' where ParentID = ' & Par:ParentID
if useTransaction
    if AbortTransaction
        rollback()
    else
        commit()
    end
end
endtime = clock()
setcursor()
0{PROP:text} = 'updated ' & maxKids |
    & ' records in ' & (endtime - starttime)/100 |
    & ' seconds'
BRW5.ResetFromFile()

```

There are several points of interest in both blocks of code. First, although I'm calling LOGOUT on two tables, only one is necessary, in my experience; this triggers the call to the ODBC SQLTransact function.

Second, the data creation code relies on server-side autoincrementing of the primary key values, which speeds up inserts considerably, as the Clarion application doesn't have to issue a SELECT before each insert to get the highest used key value. Server-side autoincrementing introduces a new problem, however. If you insert a parent record, you

don't immediately know what its primary key is, because that's assigned by the server. You need that value for the foreign key in the child record, but it isn't yet in the parent record buffer. In MySQL, the `last_insert_id()` function will return the most recently-created autonumber primary key for the current connection.

It would be nice if you could return the value of a MySQL function directly using `PROP:SQL`, but instead you have to store that value in another table, and then retrieve it like any other data. That's what the `LastInsert` table is for. The following code uses `PROP:SQL` to remove any existing record from that table, adds a new record with the just-assigned autoincrement id, and returns that value with a `NEXT()` on the `LastInsert` table. Of course, it's still possible this code could fail in a multi-user situation, since others could be trying to change the data in the `LastInsert` table at the same time. You could add a `UserID` field to the `LastInsert` table, and keep one record per user instead of just one record.

```
if access:Parents.insert() = level:benign
    lastinsert{PROP:sql} |
        = 'delete from lastinsert'
    lastinsert{PROP:sql} |
        = 'insert into lastinsert (id) ' |
        & 'values (last_insert_id())'
    lastinsert{PROP:sql} |
        = 'select * from lastinsert'
access:lastinsert.next()
ParentID = las:ID
```

So much for the code; how does the application run?

Testing transactions

In my test installation, with MySQL Max v. 3.23.42 and MyODBC 2.50.39, transactions work just fine. The database server is a Celeron 400 running RedHat 7.1 with two IBM 20 GB drives mirrored on a 3Ware Escalade controller; my test application is running on an NT box on a 100BaseT network.

A test insert of 500 records, with transactions enabled, takes almost exactly one second (including creating the parent record and getting the autoincrement ID). The time is the same whether you commit or rollback the transaction - this is, I assume, because the server caches the writes. If, however, you turn off transactions, InnoDB inserts take quite a bit longer, as the server flushes each write. Instead of one second, the operation takes almost eight seconds!

Mass updates (changing one field in all 500 child records to a new value) is speedy, as you'd expect. On my installation this operation takes about .04 seconds.

You can easily compare InnoDB tables to MyISAM tables: just run the following script against your transact database:

```
DROP TABLE Parents;
DROP TABLE Children;

CREATE TABLE Parents(
ParentID INT NOT NULL AUTO_INCREMENT,
LastModified TIMESTAMP,
PRIMARY KEY (ParentID)
TYPE=MYISAM;

CREATE TABLE Children(
ChildID INT NOT NULL AUTO_INCREMENT,
ParentID INT NOT NULL,
SomeWeirdField INT DEFAULT 0,
LastModified TIMESTAMP,
PRIMARY KEY (ChildID),
INDEX ChildrenIdx_ParentID (ParentID)
TYPE=MYISAM;
```

These definitions are identical to the original definitions except the table type is MYISAM instead of INNODB. Now it won't make any difference whether you enable or disable transactions, since MYISAM tables don't support transactions. MYISAM mass insert speed in the test application is comparable to INNODB with transactions enabled. Mass updates are several times faster than INNODB mass updates (on the order of .01 second to update all 500 records with one new field value instead of .04 seconds).

Michael "Monty" Widenius recently indicated on the MySQL mailing list that there is a benchmark page forthcoming (on the MySQL site) which will compare the MyISAM, InnoDB, and HEAP table types, and will interpret the results. He also indicated that if you have a lot of possible conflicts between writes/updates/selects, i.e. where any one statement could take a lot of time, InnoDB should be faster than MyISAM.

If you're doing just a lot of data retrieval, however, MyISAM should be faster. Each MyISAM table is stored in two physical files, one for the table's indexes, and another for the table's data. To retrieve a record, the library code looks at an index and then gets a record by data offset. In an InnoDB database the primary key and row are stored together, and additional keys are stored as that key plus the primary key. The result is that a fetch on just the primary key is probably faster with InnoDB, but slower for fetches on secondary keys since they require an additional lookup on the primary key. "Big rows" are also expected to give poorer performance than on MyISAM tables.

Which do I choose?

It seems clear that for business software development, InnoDB tables, with row-level locking and transactions, are a better choice than MyISAM tables. Although I haven't yet

done any real-world comparisons, I would also expect comparable or better performance out of InnoDB tables with the kinds of applications most Clarion developers create.

InnoDB tables are also growing foreign key support, which should be ready for prime time shortly. As of InnoDB version 3.23.43b you define foreign keys in the `CREATE TABLE` statement as follows:

```
CREATE TABLE parent (
    id INT NOT NULL,
    PRIMARY KEY (id)
) TYPE=INNODB;

CREATE TABLE child(
    id INT,
    parent_id INT,
    INDEX par_ind (parent_id),
    FOREIGN KEY (parent_id) REFERENCES parent(id)
) TYPE=INNODB;
```

There are no delete cascades at this time, so I assume that these definitions represent a restrict constraint, but I haven't tested any of this yet. See the InnoDB manual (<http://www.innodb.com/ibman.html>) for more information.

The downside to InnoDB is its relative youth within the MySQL environment. Although Heikki Tuuri began developing InnoDB in 1994, the MySQL integration only began about six years after that. Despite this, InnoDB has an excellent reputation, and is used live by a number of prominent sites, most noticeably Slashdot (<http://slashdot.org/>). You also don't need to convert your whole database - you can easily mix InnoDB tables with MyISAM tables in the same database.

Source code

See "Appendix A: Getting Support," p. 601, for information on how to get the source accompanying this book.

- v3n10inno.zip

LARGE TABLE PERFORMANCE IN MySQL

by David Harms

Recently there's been some discussion of MySQL/MyODBC performance with large tables. As most of the work I do is with tables of less than 10,000 records, I haven't really noticed the problems others have reported. But I decided it was time to do some testing with large data sets.

As it happens I do have one fairly large table, a web server log with over five million records. In order to test this table, however, I first wanted to move it to a test server.

Moving tables around

There are several ways to move MySQL tables from one server to another. If you're dealing with ISAM or MyISAM tables (these are the MySQL default table types for older and current versions) then you can simply shut down the MySQL server and copy the data files to a new location. On my Linux server, each MySQL database is stored in its own directory under the /var/lib/mysql directory. After copying the data files across I simply restarted MySQL, added permissions as necessary (using the GRANT statement), and started working with the tables.

There is one caveat to copying MySQL tables: if you're using a version of MySQL older than 3.23 then chances are your tables are the older ISAM rather than MyISAM tables. ISAM tables are only portable within platforms, i.e. you can copy MySQL ISAM tables from one SPARC machine to another, or one Intel machine to another, but not from a SPARC to an Intel machine (because of byte ordering differences). MyISAM tables, however, which appeared as of MySQL 3.23, are binary compatible between platforms. If you're using a more recent version of MySQL but began with a pre-MyISAM version then you may need (or want) to convert your tables to MyISAM format. You can do this with the ALTER TABLE statement:

```
ALTER TABLE tablename TYPE=MYISAM;
```

Note: In "MySQL: InnoDB Tables And Transactions," p. 247 I discussed the InnoDB table type, which supports transactions and row level locking; you cannot physically copy InnoDB tables around, or at least I wouldn't advise trying it!

If you're not sure what type your tables are now, you can issue the SHOW TABLE STATUS command. This will give you many columns of information including the average row length, data size, last autoincrement primary key value and more, but I've truncated the results here to just the first four columns:

```
mysql> show table status;
+-----+-----+-----+-----+
| Name | Type | Row_format | Rows |
+-----+-----+-----+-----+
| AccessHistory | ISAM | Dynamic | 1152525 |
| AccessLog | ISAM | Dynamic | 14063 |
+-----+-----+-----+-----+
```

TIP: When I get a really wide result back from the mysql client, the lines wrap making it difficult to read. I usually paste such results into Windows Notepad and turn off word wrap. The Clarion editor also works fine. I can easily copy and paste because I use a Windows terminal program (such as CRT (<http://www.vandyke.com/>)) to telnet or SSH to the Linux server.

If you get a syntax error trying to execute SHOW TABLE STATUS then you're using an older version of MySQL and you have ISAM tables.

If copying tables across whole isn't an option you can always use the mysqldump command line utility. Well, mostly always. I found I ran out of memory when I tried this on my five million record table, but I might have had better success with the -q (do not buffer) option. The mysqldump utility creates standard output that includes a table creation statement and INSERT statements for all of the table's data. You can tell

`mysqldump` to dump one table, selected tables, or one or more entire databases. Here are some of the more useful `mysqldump` options:

Option	Description
<code>--add-drop-table</code>	Add a drop table before each create statement.
<code>-A, --all-databases</code>	Dump all the databases.
<code>-a, --all</code>	Include all MySQL-specific create options (see CREATE TABLE syntax).
<code>--allow-keywords</code>	Prefix each column name with the table name to allow keywords as column names.
<code>-c, --complete-insert</code>	Use complete insert statements (with column names) - handy if you are using MySQL dump to move data to a table which has added columns.
<code>--delayed</code>	Insert rows with the INSERT DELAYED command - this assigns a lower priority to the insert so it doesn't interfere with other operations.
<code>-e, --extended-insert</code>	Use the new multiline INSERT syntax. (Gives more compact and faster inserts statements.)
<code>--help</code>	Display a help message and exit.
<code>-F, --flush-logs</code>	Flush the MySQL log file before starting the dump.
<code>-f, --force,</code>	Continue even if a SQL error occurs during a table dump.
<code>-h, --host=..</code>	Specify the server host name. The default host is <code>localhost</code> .

<code>-l, --lock-tables.</code>	Lock all tables before starting the dump. The tables are locked with READ LOCAL to allow concurrent inserts in the case of MyISAM tables.
<code>-t, --no-create-info</code>	Don't write table creation information (the CREATE TABLE statement).
<code>-d, --no-data</code>	Just dump the table structure, not the data.
<code>--opt</code>	Same as --quick --add-drop-table --add-locks --extended-insert --lock-tables. Should give you the fastest possible dump for reading into a MySQL server.
<code>-pyour_pass, --password[=your_pass]</code>	Password to use when connecting - if password is omitted you will be prompted.
<code>-q, --quick</code>	Don't buffer query, dump directly to stdout.
<code>-u user_name, --user=user_name</code>	The MySQL user name to use when connecting to the server. The default value is your Unix login name.
<code>-v, --verbose</code>	Verbose mode. Print out more information on what the program does.
<code>-w, --where='where-condition'</code>	Dump only selected records. Quotes are mandatory, i.e.: "--where=user='jimf'" "-wuserid>1" "-wuserid<1"

Whether you're moving data between servers, doing backups, or just need to export/import data, mysqldump is a most useful program. Get familiar with it.

The test table

For test purposes, I used a modified version of my `DetailHistory` table, a log file that records every request made to the server, whether for an article, an image, or any other file. Here's the relevant table status information for `DetailHistory`:

Name	DetailHistory
Type	MyISAM
Row Format	Dynamic
Rows	5441439
Average Row Length	54
Data Length	295372180
Maximum Data Length	4294967295
Index Length	62568448
Data Free	0
Auto Increment No	5441440
Create Time	2001-10-25 19:25:15
Update Time	2001-10-25 19:28:34
Check Time	2001-10-25 19:29:41
Create Options	pack_keys=1

Because this table gets a lot of inserts and has a lot of records, I didn't originally have any indexes declared at all. Actually I didn't even have a primary key declared - this isn't an absolute requirement in MySQL, apparently. The more indexes a table has (or the larger the fields being indexed), the slower inserts will be, since each insert has to also update the indexes. If you're doing a big (I mean *really* big) batch insert of data into a table you may find it faster to drop all the indexes, add the data, and then recreate the indexes after all the inserts are done.

In any case, I decided that a primary key would be a good idea, so I added one:

```
mysql> alter table DetailHistory add index
      > DetailHistoryIdx_ReqDateTime(ReqDateTime);
Query OK, 5441439 rows affected (4 min 19.69 sec)
Records: 5441439 Duplicates: 0 Warnings: 0
```

As you can see it took this server (a Celeron 400 running RedHat 7.1 with two IBM 20 GB drives mirrored on a 3Ware Escalade controller) just over four minutes to add an

autoincremented primary key value to a table with 5.5 million rows and almost 300 megs of data.

Judicious use of indexes becomes critical on large data sets. For instance, the DetailHistory had a user field which I suspected was unused, since the code that inserts these log files is completely separate from the authentication system. I decided to have a look:

```
mysql> select user from DetailHistory
      > where user <> '';
+-----+
| user |
+-----+
| testname |
| testname |
| testname |
| testname |
+-----+
4 rows in set (11 min 15.87 sec)
```

Because there was no index on the user field, the server had to look through all five and a half million records, and the query took a substantial length of time. To compare times, I created an index on the user field:

```
mysql> alter table DetailHistory add index
      > DetailHistoryIdx_User(user);
```

I then ran the query again:

```
mysql> select user from DetailHistory
      > where user <> '';
+-----+
| user |
+-----+
| testname |
| testname |
| testname |
| testname |
+-----+
4 rows in set (24.71 sec)
```

Not that speedy, but better. Of course, looking for a specific user value is quite fast:

```
mysql> select user from DetailHistory
      > where user ='testname';
+-----+
| user |
+-----+
| testname |
| testname |
| testname |
| testname |
+-----+
4 rows in set (0.00 sec)
```

A limited select in user order is similarly fast:

```
mysql> select user from DetailHistory
      > order by user desc limit 4;
+-----+
| user   |
+-----+
| testname |
| testname |
| testname |
| testname |
+-----+
4 rows in set (0.01 sec)
```

So much for testing. I don't actually need that user field, so I dropped it from the table.

```
mysql> ALTER TABLE DetailHistory drop column user;
Query OK, 5441439 rows affected (4 min 26.71 sec)
Records: 5441439 Duplicates: 0 Warnings: 0
```

I do want at least one other index on IP address, so I have something other than the primary key for browse testing:

```
mysql> ALTER TABLE DetailHistory
      > ADD INDEX DetailHistoryIdx_IP(IP);
Query OK, 5441439 rows affected (6 min 30.36 sec)
Records: 5441439 Duplicates: 0 Warnings: 0
```

Again, it's a few minutes to complete the change. Finally, it's time to do some testing!

Testing a large MySQL table

When it comes to really large MySQL tables and the MyODBC driver, I have bad news, a workaround, and a ray of hope. The bad news is that you really can't use large MySQL tables with Clarion/ABC at present (at least as of 5.5E), because the ODBC driver isn't setting the `LIMIT` clause on the `SELECT` statement. That means that if, for instance, you have over five million records in the table, the server will have to retrieve all of those records before your browse can begin to display! That is, of course, unworkable; I have filed a bug report.

The workaround is to use `PROP:SQL` to specify the `SELECT` statement, with `LIMIT` clause. When you explicitly set the SQL statement and issue a `NEXT()`, you can use `LIMIT` with MySQL tables. I would expect the CCS templates (www.icetips.com) would work fine with MySQL. Unfortunately, I haven't found an easy way to circumvent the ABC classes' communication with the driver.

The ray of hope is a bug fix - bug fixes spring eternal!

Another point worth noting is that if you have a filter on your huge file, and that filter uses keys and is therefore reasonably speedy, you can probably still use an ABC browse with MySQL. You'll be retrieving more records than you need to see, but if you have a

reasonably fast server and network, you might be able to *temporarily* get away with a select that returns a few hundred or a few thousand records, even though you only view 20 or so at a time. Yes, you'll be putting the server to way too much work, but if you have some capacity to spare this may get you by.

Summary

The MySQL server is quite capable of handling large files; unfortunately, Clarion ABC applications, at present, don't pass `LIMIT` clauses through to the back end, so if you don't have a filter on your data, any page browse will cause the server to retrieve *all* the records in the table. If that's your situation, you can use `PROP:SQL` in hand code, or you can use a commercial product that uses `PROP:SQL`, or you can wait for a bug fix.

GETTING STARTED WITH POSTGRESQL

by David Harms

In the world of open source databases, there are two long-standing rivals, MySQL and PostgreSQL, and one new contender, Firebird, which is the open source version of Interbase. In this chapter I'll be talking about PostgreSQL, and providing a diary of sorts of my attempt to migrate one database from MySQL to PostgreSQL.

PostgreSQL has its roots firmly in the University of California at Berkeley. Its ancestor is Postgres, an object-relational database developed at the university in the late 80s and early 90s. Postgres used a query language called POSTQUEL; in 1994-5, Jolly Chen and Andrew Yu, graduate students at Berkeley, added SQL capabilities and called the database Postgres95. In 1996, Marc Fournier volunteered to host the server for the source tree and the mailing list, and Postgres95 became PostgreSQL. Since that time, the PostgreSQL open source community has clearly flourished, and PostgreSQL has become quite popular.

But not as popular as MySQL, it would seem. MySQL has, for some time, been a darling of the computer press, an up-and-comer threatening the big database vendors. And this has gotten up the noses of some PostgreSQL supporters. Why, they ask, does everyone talk about MySQL, when PostgreSQL has more "real" SQL database features, like views, sub-selects, transactions (okay, MySQL has those now) and so forth?

Part of the answer, I think, is that MySQL has, for years, had a native Windows version, in addition to versions that run on the many Unix/Linux platforms. With PostgreSQL, you could run on just about any hardware, but if you wanted to run on Windows, you had to do so inside the Cygwin Unix environment (<http://www.cygwin.com/>) for Windows. Ugh. Can you say “emulation?”

I’m happy to report that a native Windows version of PostgreSQL has now arrived. Well, at least you can get a beta at ftp://209.61.187.152/postgres/postgres_beta4.zip. This version was developed by PeerDirect (<http://www.peerdirect.com/>), and will reportedly be contributed to the PostgreSQL project in December 2002. Interestingly, the beta is on a NuSphere server, and NuSphere (a subsidiary of Progress Software, as is PeerDirect) recently settled a lawsuit launched by MySQL AB involving NuSphere’s creation of a mysql.org web site, its use of trademarks, and its alleged failure to release the Gemini database handler under the GPL.

Open source database internecine warfare aside, the PostgreSQL folks say on their advocacy site (<http://advocacy.postgresql.org/advantages/>) that the Windows version will be part of the official distribution as of version 7.4. The current release is 7.2.3, and 7.3 is in beta.

Editors note: More recent information on the PostgreSQL Windows beta can be found at <http://techdocs.postgresql.org/guides/Windows>

Installing the PostgreSQL Windows beta

The beta is quite easy to install - just unzip everything into a directory, and then modify the setenv.bat file (which you’ll find in that directory) accordingly. My setenv.bat looks like this:

```
set PGHOME=d:\postgres_beta4
set PGDATA=%PGHOME%\data
set PGLIB=%PGHOME%\lib
set PGHOST=localhost
set PATH=%PGHOME%\bin;%PATH%
```

Run the batch file from a command prompt, not from Windows, because after you set the environment variables you’ll need to initialize the database by running the initdb utility. Here’s the output on my machine:

```
D:\postgres_beta4>initdb
The files belonging to this database system will be owned by user
"dharms".
This user must also own the server process.

creating directory d:\postgres_beta4\data...ok
creating directory d:\postgres_beta4\data\base...ok
```

```
creating directory d:\postgres_beta4\data\global...ok
creating directory d:\postgres_beta4\data\pg_xlog...ok
creating directory d:\postgres_beta4\data\pg_clog...ok
creating template1 database in d:\postgres_beta4\data\base\1
...creating configuration files...ok
initializing pg_shadow...ok
enabling unlimited row size for system tables...ok
creating system views...ok
loading pg_description...ok
Installing PeerDirect UltraSQL Replication Adapter Support
vacuuming database template1...ok
copying template1 to template0...ok

Success. I could now start the database server using:
d:\postgres_beta4\bin\postmaster -D d:/postgres_beta4/data
```

or

```
d:\postgres_beta4\bin\pg_ctl -D d:/postgres_beta4/data
-l logfile start
```

I put the start command into start.bat and gave it a whirl.

```
d:\postgres_beta4\bin\postmaster -D d:/postgres_beta4/data
```

Here's the output from the command:

```
DEBUG: database system was shut down at 2002-11-22 15:33:53 Central
Stan
DEBUG: checkpoint record is in pg_xlog/0000000000000000 at offset
2184988
DEBUG: redo record is at 0/21571C; undo record is at 0/0; shutdown
TRUE
DEBUG: next transaction id: 541; next oid: 16557
DEBUG: database system is ready
```

At this point the server is running. To shut it down, I issued a Ctrl-C in the command window. The server responded with a “fast” shutdown:

```
DEBUG: fast shutdown request
DEBUG: shutting down
DEBUG: database system is shut down
```

I could also have used pg_ctl with the stop parameter. Okay, after running start.bat again, I was ready to start mucking about with the database! From my minimal previous experience with PostgreSQL on Linux, I knew that I could use the psql client to get access. And sure enough, in the bin subdirectory, there was psql.exe and a bunch of other utilities. I tried

```
psql
```

and was rewarded with

```
psql: FATAL 1: Database "dharms" does not exist in the system
catalog.
```

I was reminded that psql defaults to the current user, and a database with the same name as the current user. This makes multiple user administration much easier, particularly if you change permissions so that users can only access their own databases. But I digress. There was no dharms database, so I tried to create one.

```
createdb dharms
```

and got

```
D:\POSTGR~1\bin>createdb dharms
psql: FATAL:  user "dharms" does not
createdb: database creation failed
```

As Jerry Pournelle would say, “Alas.” Time to refresh my memory on PostgreSQL security procedures. Clearly I needed to add myself to the user list, and to do that I needed to know the default superuser id. I tried postgresql, I tried root, I tried a lot of things, including pulling my hair out and emailing Val Raemaekers, who I knew had run the beta successfully. Then I ran psql on a Linux box on which someone else had installed PostgreSQL, and listed the databases. The owner of the default databases? It was *postgres* – not *postgresql*, which I’d tried, but *postgres*. Now I could create my dharms user:

```
D:\POSTGR~1\bin>createuser -U postgres -e dharms
Shall the new user be allowed to create databases? (y/n) y
Shall the new user be allowed to create more new users? (y/n) y
CREATE USER "dharms" CREATEDB CREATEUSER
CREATE USER
```

I did hear back from Val, who said “I installed pgadminII which I was using to administer PostgreSQL on FreeBSD, just pointed it to localhost with my Windows 2000 username and password, and it connected with no problems at all.” So you may not encounter the same problem I did.

With the dharms user in place I was able to run psql without any parameters, and have it default to my user id as the user and database name:

```
D:\POSTGR~1\bin>psql
Welcome to psql, the PostgreSQL interactive terminal.

Type: \copyright for distribution terms
      \h for help with SQL commands
      \? for help on internal slash commands
      \g or terminate with semicolon to execute query
      \q to quit
```

Take careful note of this first message. One, it tells you how to get out of psql – by using the \q command. And two, it tells you that there are a bunch of internal, non-SQL commands you can run, and you can find out about those with the \? command:

```

dharma=# \?
\a          toggle between unaligned and aligned output mode
\c[onnect] [DBNAME|-[USER]]      connect to new database (currently "dharma")
\c TITLE      set table title
\cd [DIRNAME]  change the current working directory
\copy ...     perform SQL COPY with data stream to the client host
\copyright    show PostgreSQL usage and distribution terms
\d TABLE      describe table (or view, index, sequence)
\d{t|i|s|v}... list tables/indexes/sequences/views
\d{p|S|l}     list access privileges, system tables, or large
objects
\da          list aggregate functions
\dd NAME     show comment for table, type, function, or operator
\df          list functions
\do          list operators
\dT          list data types
\e FILENAME   edit the current query buffer or file with external
editor
\echo TEXT    write text to standard output
\encoding ENCODING  set client encoding
\f STRING     set field separator
\g FILENAME   send SQL command to server (and write results to
file or |pipe)
\h NAME      help on syntax of SQL commands, * for all commands
\H           toggle HTML output mode (currently off)
\i FILENAME   execute commands from file
\l           list all databases
\lo_export, \lo_import, \lo_list, \lo_unlink
Large object operations
\o FILENAME   send all query results to file or |pipe
\p           show the content of the current query buffer
\pset VAR     set table output option (VAR := {format|border|expanded|}

fieldsep|null|recordsep|tuples_only|title|tableattr|pager})
\q           quit psql
\qecho TEXT   write text to query output stream (see \o)
\qreset       reset (clear) the query buffer
\s FILENAME   print history or save it to file
\set NAME VALUE  set internal variable
\t           show only rows (currently off)
\T TEXT      set HTML table tag attributes
\unset NAME   unset (delete) internal variable
\w FILENAME   write current query buffer to file
\x           toggle expanded output (currently off)
\z           list table access privileges
\! [COMMAND]  execute command in shell or start interactive shell

```

You won't use most of these commands on a day to day basis, but a few are particularly useful. One is the list databases command, \l:

```

dharma=# \l
List of databases
 Name   | Owner
-----+-----
 admin  | admin
 template0 | postgres

```

```
template1 | postgres
(5 rows)
```

To connect to a database, use the `\c` command. And once you're connected, you can use `\d` to list or describe the available tables, indexes, sequences, and views. There are also commands to modify the output of SELECT statements, write results to files, work with large objects, set variables, and more.

With the pending release of a native Windows version of Postgresql, and the immediate availability of a beta, one of the biggest obstacles to Windows developers using this popular and robust SQL database has been (or, if you're more cautious, will soon be) removed.

Setting up the database

As I said earlier, I ran into a bunch of trouble getting a working user name and password for the Windows version of `psql`, the command line interface to PostgreSQL (although not everyone who's run the beta has had the same difficulty). On Linux, if you're administering the server, you can log in as root and assume the `postgres` user's identity with the `su` command:

```
[root@ns root]# su postgres
bash-2.05$ createdb demo
Shall the new user be allowed to create databases? (y/n) n
Shall the new user be allowed to create more new users? (y/n) n
CREATE USER
bash-2.05$ createdb demo
CREATE DATABASE
bash-2.05$
```

If you don't have root access, then whoever does have root access can set up your database access for you.

Notice that in my example above the user and the database have the same name. This isn't absolutely necessary, but `psql` will default to the currently logged in user name for both the `psql` user and the database name. This makes for easy management where you want to give each logged in user their own PostgreSQL database.

Connecting

Now that I have created a `demo` database, and a `demo` PostgreSQL user, I can either log in to Linux as user `demo` and execute

```
psql
```

or as another user I can enter:

```
psql -U demo -d demo
```

In either case I'll get the following greeting:

```
Welcome to psql, the PostgreSQL interactive terminal.  
Type: \copyright for distribution terms  
\h for help with SQL commands  
\? for help on internal slash commands  
\g or terminate with semicolon to execute query  
\q to quit
```

```
demo=>
```

You can quickly check the version of PostgreSQL you're running by calling the `version()` function:

```
demo=> select version();  
          version  
-----  
 PostgreSQL 7.2.1 on i686-pc-linux-gnu, compiled by GCC 2.96  
(1 row)
```

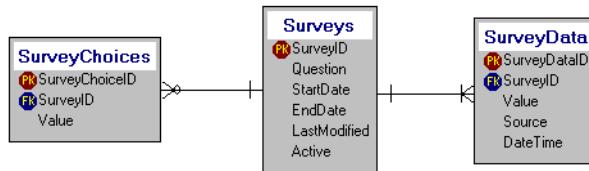
As you can see I'm one dot release behind the times. You can also check the `psql` client version with either the `--version` or the `-V` option:

```
# psql --version  
psql (PostgreSQL) 7.2.1  
contains support for: readline, history, multibyte  
Portions Copyright (c) 1996-2001, PostgreSQL Global Development  
Group  
Portions Copyright (c) 1996, Regents of the University of California  
Read the file COPYRIGHT or use the command \copyright to see the  
usage and distribution terms.  
[root@ns admin]#
```

Creating tables

Figure 1 shows the database I'll be creating in this installment. These are the tables that I currently use to store Survey data for *Clarion Magazine*, using MySQL. The titles are fairly self-explanatory.

Figure 1: The Surveys database diagram



Here's a CREATE TABLE statement for Surveys:

```

CREATE TABLE Surveys(
  SurveyID serial NOT NULL PRIMARY KEY,
  Question varchar(255),
  StartDate date,
  EndDate date,
  LastModified timestamp,
  Active bool DEFAULT False);
  
```

When I paste this statement into `psql` and execute it, I get the following result:

```

NOTICE: CREATE TABLE will create implicit sequence
'surveys_surveyid_seq' for SERIAL column 'surveys.surveyid'
NOTICE: CREATE TABLE / PRIMARY KEY will create implicit
index 'surveys_pkey' for table 'surveys'
  
```

Two additional structures have been automatically created, both because of this line in the declaration:

```

SurveyID serial NOT NULL PRIMARY KEY,
  
```

The second item is the index for the primary key, which is no surprise. But SurveyID also is my autoincrementing unique identifier for each row, as indicated by the SERIAL data type. SERIAL values are stored in eight byte integers, with a maximum value of 9223372036854775807 (big enough for ya?) unless the platform compiler doesn't support eight byte integers, in which case the maximum value is the same as a Clarion LONG. A SERIAL column causes PostgreSQL to create a sequence number generator, called simply a *sequence*, which is used to automatically increment that value. Here's the description of the table as shown by `psql` after I issued the CREATE:

```

demo=> \d surveys;
  
```

Column		Table "surveys"	
	Type		Modifiers
surveyid	integer		not null default nextval('surveys_surveyid_seq')
question	character varying(255)		
startdate	date		
enddate	date		
lastmodified	timestamp with time zone		
active	boolean		default 'f'::bool
Primary key: surveys_pkey			

As you can see the default value for `surveyid` references `surveys_surveyid_seq`. Sequences are actually single-row tables (think control files) with special associated functions: `setval()`, which lets you change the current value in the row; `nextval()`, which increments the current value (and first adds the row if the sequence has never been used before) and returns that value; and `currval()`, which returns the value last set by `nextval()` or `setval()` *for the current process*. This last point is an important one. If, for instance, I add the very first `surveys` record, `currval('surveys_surveyid_seq')` will return 1 even if another process (typically another user) inserts a `surveys` record after I inserted the first one, and before I called `currval()`.

The fact that sequences are actually tables gives you some flexibility in autoincrementing. You can create them separately from your tables and with specialized attributes, including reverse order, increment steps other than 1, and wrap-around. Here's a sequence that starts at 4, decrements by two, and when it hits zero starts over at 10 again.

```
CREATE TEMPORARY SEQUENCE increment_test
    INCREMENT -2 MINVALUE 0 MAXVALUE 10
    START 4 CYCLE;
```

Repeatedly calling `SELECT nextval('increment_test')` on this sequence yields the following numbers:

```
4 2 0 10 8 6 4 2 0 10 8
```

... and so on. Also note that I've used the `TEMPORARY` qualifier on this sequence - once the session is closed this sequence will be deleted. You can specify the name of an existing sequence when creating a temporary sequence. In that case the temporary sequence will replace the permanent sequence for the duration of the session. You would, of course, want to use something like this *very* carefully.

Why would you want a sequence to cycle? As Carl Barnes pointed out to me after reading the draft of this chapter, one application would be a Job/Order number that you don't want to exceed, say, five digits. Of course that would assume that you're archiving/deleting old orders so they can't cause duplicate key errors.

Because sequences are separate entities from tables, there is one side effect you might not expect. If you subsequently issue a `DROP TABLE` command, as in:

```
DROP TABLE Surveys;
```

the table and any indexes will be deleted, but the sequence will *not* be deleted. If you reissue the `CREATE TABLE` command, you'll get an error like this one:

```
ERROR: Relation 'surveys_surveyid_seq' already exists
```

So what do you do? Issue a `DROP SEQUENCE`:

```
demo=> DROP SEQUENCE surveys_surveyid_seq;
```

Now you can create the table, and its associated sequence will also be created.

Going back to the table description reported by PostgreSQL, you'll notice a `::` operator used in two places. This is the cast operator, and it first shows up in the primary key column that uses the sequence:

```
surveyid | integer | not null default
          nextval('surveys_surveyid_seq)::text)
```

Since PostgreSQL supports function overloading, it's conceivable that there could be versions of the `nextval()` function which take parameters of data type other than text. The `::text` operator ensures that the parameter to `nextval()` is interpreted as text. Seems a bit paranoid, but there you are.

There is also a cast on the boolean field named `active`, which is cast to a boolean value (`'f'::bool`). In PostgreSQL, the possible true values for a boolean are `TRUE`, `'t'`, `'true'`, `'y'`, `'yes'`, and `'1'`, while possible false values are `FALSE`, `'f'`, `'false'`, `'n'`, `'no'`, and `'0'`. The cast `'f'::bool` ensures that the default value is in fact boolean.

Because sequences are independent of the table they don't necessarily function the same way as a client-side autonumber, where the code looks at the highest value in the primary key, increments, and tries to add the placeholder record. For instance, imagine a sequence that's been used just once. You can see the sequence data with a `SELECT` statement like this:

```
demo=> select * from surveys_surveyid_seq;
      sequence_name | last_value | increment_by |      max_value
-----+-----+-----+-----+
surveys_surveyid_seq |           1 |           1 |
9223372036854775807
      | min_value | cache_value | log_cnt | is_cycled | is_called
-----+-----+-----+-----+-----+
      |       1 |           1 |     32 | f         | t
```

The sequence has a `last_value` of 1 because I've already added a record in the `surveys` table, without specifying a value for the `surveyid` column. Here's the resulting data:

```
demo=> select * from Surveys;
 surveyid | question      | startdate    | enddate     | lastmodified |
 active
-----+-----+-----+-----+-----+
 f           1 | First survey | 2003-02-03 | 2003-03-05 |
```

Now I add a record specifying a primary key value of 4:

```
INSERT INTO surveys (surveyid,question,startdate,enddate)
values(4,'Fourth survey',now(),now()+30);
```

There are now two records in the table:

```
demo=> select * from surveys;
 surveyid | question      | startdate    | enddate     | lastmodified |
 active
-----+-----+-----+-----+-----+
 f           1 | First survey | 2003-02-03 | 2003-03-05 |
 f           4 | Fourth survey | 2003-02-03 | 2003-03-05 |
(2 rows)
```

The sequence, however, remains unchanged because a value was supplied for the `surveyid` field:

```
demo=> select * from surveys_surveyid_seq;
 sequence_name | last_value | increment_by | max_value
-----+-----+-----+-----+
 surveys_surveyid_seq | 1 | 1 |
 9223372036854775807
 | min_value | cache_value | log_cnt | is_cycled | is_called
-----+-----+-----+-----+-----+
 | 1 | 1 | 0 | f | t
```

So what happens now? If I continue adding records, I'll end up with a duplicate key error when I hit surveyid 4:

```
demo=> INSERT INTO surveys (question,startdate,enddate)
values('Second survey',now(),now()+30);
INSERT 506966 1
demo=> INSERT INTO surveys (question,startdate,enddate)
values('Third survey',now(),now()+30);
INSERT 506967 1
demo=> INSERT INTO surveys (question,startdate,enddate)
values('Fourth survey',now(),now()+30);
ERROR:  Cannot insert a duplicate key into
unique index surveys_pkey
```

```
demo=> INSERT INTO surveys (question,startdate,enddate)
values('Fifth survey',now(),now()+30);
INSERT 506969 1
```

Although the attempt to insert the record with `surveyid` 4 failed, the sequence did increment, so subsequent inserts will again work. The moral of the story, however, is that you really have to be careful if you import a bunch of records with existing primary key values. You might want to set the sequence number to a value higher than any of the primary key values you're importing, leaving a window for the existing values. All subsequent calls to `nextval()` will start at that number plus one. You do this with the `setval()` function:

```
demo=> select setval('surveys_surveyid_seq',1000);
setval
-----
1000
(1 row)
```

And the result:

```
demo=> select * from surveys_surveyid_seq;
sequence_name | last_value | increment_by | max_value
-----+-----+-----+-----+
surveys_surveyid_seq | 1000 | 1 |
9223372036854775807

| min_value | cache_value | log_cnt | is_cycled | is_called
+-----+-----+-----+-----+
| 1 | 1 | 1 | 0 | f | t
```

One curiosity of sequences is that the minimum value of any sequence is 1, so you can't issue a `setval('sequencename', 0);` if you set the current value to 1, then the next available value for any serial data type is 2. That means you can't reset an existing sequence so it will start at 1 - you have to drop the sequence and recreate it. At least that's been my experience.

The SurveyChoices table

As shown in Figure 1. the `Surveys` database includes a table for survey choices:

```
CREATE TABLE SurveyChoices (
SurveyChoiceID serial NOT NULL PRIMARY KEY,
SurveyID int NOT NULL,
Sequence decimal(5,2) DEFAULT 1,
Value varchar(100),
FOREIGN KEY (SurveyID) REFERENCES Surveys (SurveyID)
ON DELETE CASCADE
ON UPDATE CASCADE);
```

And just to speed things up, here are a couple of indexes:

```
CREATE INDEX SurveyChoices_Idx_ID
    ON SurveyChoices (SurveyID);
CREATE INDEX SurveyChoices_Idx_ID_Seq
    ON SurveyChoices (SurveyID, Sequence);
```

`SurveyChoices` is linked to the `Surveys` table via the `SurveyID` column. Here's the output from the `CREATE` statement:

```
NOTICE: CREATE TABLE will create implicit sequence
'surveychoices_surveychoiceid_seq' for SERIAL
column 'surveychoices.surveychoiceid'
NOTICE: CREATE TABLE / PRIMARY KEY will create implicit
index 'surveychoices_pkey' for table 'surveychoices'
NOTICE: CREATE TABLE will create implicit trigger(s)
for FOREIGN KEY check(s)
```

As before, the `CREATE` statement has resulted in the creation of an implicit sequence for the `SurveyChoiceID` column, as well indexes for the primary key and for the foreign key. The `CREATE` statement also results in the creation of two triggers for the specified `ON DELETE CASCADE` and `ON UPDATE CASCADE` foreign key checks. These mean that if you attempt to delete a `Surveys` record, and there are related child `SurveyChoices` records, those child records will be deleted. Similarly if the parent primary key value changes, these changes will be rippled down to the child records (but that's irrelevant, because you always use values for primary keys that will never need to be changed, right? Right!).

So how do you find out what triggers are in place for an existing table? The answer lies in the system tables, and to get a listing of those you type `\ds`:

```
demo=> \ds
          List of relations
   Name        |  Type   | Owner
-----+-----+-----
pg_aggregate | table  | postgres
pg_am         | table  | postgres
pg_amop       | table  | postgres
pg_amproc    | table  | postgres
pg_attrdef    | table  | postgres
...
... Approx. 40 tables omitted for brevity
```

These system tables store everything from databases to tables to functions, check constraints, data types, and yes, triggers. It would've taken me forever to find out how to get a trigger listing from this database, but happily I found one on the web, posted by Michael Fork in the `comp.databases.postgresql.general` newsgroup on January 17, 2001:

```
SELECT pg_trigger.tgargs, pg_trigger.tgnargs,
pg_trigger.tgdeferrable, pg_trigger.tginitdeferred,
pg_proc.proname, pg_proc_1.proname FROM pg_class pg_class,
pg_class pg_class_1, pg_class pg_class_2, pg_proc pg_proc,
pg_proc pg_proc_1, pg_trigger pg_trigger, pg_trigger
```

```
pg_trigger_1, pg_trigger pg_trigger_2
WHERE pg_trigger.tgconstrelid = pg_class.oid
AND pg_trigger.tgrelid = pg_class_1.oid
AND pg_trigger_1.tgfoid = pg_proc_1.oid
AND pg_trigger_1.tgconstrelid = pg_class_1.oid
AND pg_trigger_2.tgconstrelid = pg_class_2.oid
AND pg_trigger_2.tgfoid = pg_proc.oid
AND pg_class_2.oid = pg_trigger.tgrelid
AND ((pg_class.relname='<<PRIMARY KEY TABLE>>'))
```

Find <<PRIMARY KEY TABLE>> in that listing and replace it with the name of your table. For the Surveys table, the output looks like this:

```
tgargs
-----
-----  
<unnamed>\000surveychoices\000surveys\000UNSPECIFIED\000surveyid\000
surveyid\000

| tgdeferrable | tginitedferred |      proname      |
proname
+-----+-----+-----+
-----  
| f           | f           | RI_FKey_cascade_upd |
RI_FKey_cascade_del
```

Fortunately there are tools available to make this kind of database administration easier. I use PostgreSQL Manager (<http://ems-hitech.com/pgmanager/>) from EMS (the same company that produces MySQL Manager (<http://ems-hitech.com/mymanager/>), and IB Manager (<http://ems-hitech.com/ibmanager/>) for Interbase/Firebird).

The SurveyData table

Finally, here's the table creation script for the SurveyData table, which holds the survey responses:

```
CREATE TABLE SurveyData(
SurveyDataID serial NOT NULL PRIMARY KEY,
SurveyID int NOT NULL,
Value varchar(100) NOT NULL,
Source varchar(30),
DateTime timestamp,
FOREIGN KEY (SurveyID) REFERENCES Surveys (SurveyID)
ON DELETE CASCADE ON UPDATE CASCADE);
```

And here's the result:

```
NOTICE: CREATE TABLE will create implicit sequence
'surveydata_surveydataid_seq' for SERIAL column
'surveydata.surveydataid'
NOTICE: CREATE TABLE / PRIMARY KEY will create implicit
index 'surveydata_pkey' for table 'surveydata'
```

```
NOTICE: CREATE TABLE / UNIQUE will create implicit index
      'surveydata_surveydataid_key' for table 'surveydata'
NOTICE: CREATE TABLE will create implicit trigger(s)
      for FOREIGN KEY check(s)
```

Add one index to make retrieving the survey data a bit more orderly:

```
CREATE INDEX IDX_SurveyData_Survey_Value ON
SurveyData(SurveyID,Value);
```

Table creation isn't that much different in PostgreSQL as compared to other SQL databases, although its use of sequences to handle autonumbered columns is worth some special attention.

Security basics

I said at the start of this chapter that this would be a sort of diary of my experiences. And this week I was reminded again that I find PostgreSQL's security measures a bit obtuse at times.

One caveat: I'm still using PostgreSQL 7.2, and there have been some security improvements in 7.3. I'll do my best to touch on those as I go.

Earlier, I described some of the troubles I had connecting with the native Windows beta on my development machine. I then briefly showed how to set up a user and database on a Linux box, and connect using the psql utility running in a terminal window, on that Linux box. This time around, however, I'll be using PostgreSQL on a Linux box over a LAN, eventually via the pqllibODBC driver.

This shift from connecting via a local application to a network connection is important, because PostgreSQL is, by default, set to only accept local connections. For instance, the Linux server I have running on my LAN is sporting a brand new RedHat (<http://www.redhat.com>) 8 install, including PostgreSQL. If I telnet (secure shell, actually) to the Linux box (which doesn't have its own monitor at the moment) and run psql, that application talks to the PostgreSQL postmaster program using a socket connection. It does not, however, use TCP/IP, and is not, in fact, configured to do so. That's important because any Clarion application will be using TCP/IP rather than a socket.

You can verify that PostgreSQL is not listening on the TCP/IP port (by default port 5432) by executing the following on the Linux server:

```
# telnet localhost 5432
Trying 127.0.0.1...
telnet: connect to address 127.0.0.1: Connection refused
```

If you see that message, then the first thing you'll need to do is tell PostgreSQL to start listening on the TCP/IP port. Locate the file `postgres.conf` on your system; on my Linux box it's in the `/var/lib/pgsql/data/` directory. Here's a partial listing of that file:

```
# PostgreSQL configuration file
# -----
#
# This file consists of lines of the form
#
#   name = value
#
# (The '=' is optional.) White space is collapsed, comments are
# introduced by '#' anywhere on a line. The complete list of option
# names and allowed values can be found in the PostgreSQL
# documentation. The commented-out settings shown in this file
# represent the default values.
# Any option can also be given as a command line switch to the
# postmaster, e.g., 'postmaster -c log_connections=on'. Some options
# can be changed at run-time with the 'SET' SQL command.
#=====
#
#       Connection Parameters
#tcpip_socket = false
```

As noted in the header, the commented out lines represent the default values, and `tcpip_socket` is set to `false`. Uncomment that line and change it to true (my thanks to Jeff Slarve for pointing this out):

```
tcpip_socket = true
```

Save and close the file. You will need to restart PostgreSQL to make the change take effect - on my Linux box I use this command:

```
/etc/rc.d/init.d/postgresql restart
```

Try the telnet test again. It still doesn't work? Ah, it turns out there's one other thing you need to do, and that's set the appropriate permissions. Locate the `pg_hba.conf` file, in the same directory as `postgres.conf`. At the end of that file you will see something like the following:

```
# Put your actual configuration here
# -----
#
# This default configuration allows any local user to connect with any
# PostgreSQL username, over either UNIX domain sockets or IP.
#
# If you want to allow non-local connections, you will need to add more
# "host" records. Also, remember IP connections are only enabled if you
# start the postmaster with the -i option.
#
# CAUTION: if you are on a multiple-user machine, the default
# configuration is probably too liberal for you. Change it to use
```

```
# something other than "trust" authentication.  
#  
# TYPE DATABASE IP_ADDRESS MASK           AUTH_TYPE AUTH_ARGUMENT  
  
#local all                      trust  
#host  all      127.0.0.1  255.255.255.255 trust  
  
# Using sockets credentials for improved security. Not available  
everywhere,  
# but works on Linux, *BSD (and probably some others)  
  
local  all      ident    sameuser
```

These are all comments except for the last line, which basically says that anyone logged on to the machine as a local user can access any database. Oh, and this document does also point out that you can use the `-i` option when starting the postmaster to enable TCP/IP connections, but it's more likely that you will have a startup script for PostgreSQL, in which case you should use `postgres.conf` setting. There's a lot more to `pg_hba.conf` than I've shown here - it really does give a lot of useful (if a bit cryptic) information on security settings.

In order to connect to PostgreSQL via TCP/IP, you must add something like the following to `pg_hba.conf`:

```
#TYPE  DATABASE IP_ADDRESS MASK           AUTH_TYPE AUTH_ARGUMENT  
host   all      127.0.0.1  255.255.255.255 trust
```

This line specifies the host access that is allowed. As written, it permits connection to all databases from the localhost IP address only, and does not require any authentication. If you can connect from localhost, you're good to go. In a way this is a fairly secure approach, assuming you have complete control over all applications executing on that machine. It does *not* allow an application to connect to the database from any other machine. Well, sort of.

In “Securing Remote Database Connections With SSH Tunneling,” p. 535, I describe one way to encrypt remote database connections. You could set up an SSH tunnel from a Windows box to the Linux server and connect to the PostgreSQL database using the above configuration. The SSH server is a local application, so it can talk to PostgreSQL, and your SSH client (on, say, a Windows machine) talks to the SSH server.

In any case, you'll probably want at least some level of authentication. In order to use authentication you'll need to set passwords for your users. Earlier I showed how to use `createuser` to create PostgreSQL users. With the `-P` parameter, you can get `createuser` to prompt for a password:

```
$ createuser -P cmag  
Enter password for user "cmag":  
Enter it again:  
Shall the new user be allowed to create databases? (y/n) n  
Shall the new user be allowed to create more new users? (y/n) n
```

CREATE USER

You can also create and modify users from within the `psql` utility. Typically you'll need to sign on as the `postgres` superuser. The easiest way is to log in to the Linux box as root and then `su` to the `postgres` user:

```
$ su root
Password:
# su postgres
$ psql template1
Welcome to psql, the PostgreSQL interactive terminal.

Type: \copyright for distribution terms
      \h for help with SQL commands
      \? for help on internal slash commands
      \g or terminate with semicolon to execute query
      \q to quit

template1=#
```

Note that you must supply a database name when connecting, as `psql` defaults the database name to the user name and there is (probably) no database with the name `postgres`. There is always a `template1` database, although it's not likely you'll ever modify it since it is, as the name indicates, a template for databases you create. Once logged in you can use the `CREATE USER` and `ALTER USER` statements to modify user settings.

WARNING: The `CREATE USER` statement, by default, creates a superuser, that is someone who can create databases *and* other users. So unless you're prepared to use the full syntax of `CREATE USER` you're probably better off with the command line `createuser` utility.

Now that you have a user with a password, you'll probably want a database with a matching name (or vice versa). On the face of it this seems like an odd requirement, and in fact there are ways around it. But one of the authentication options in the `pg_hba.conf` file is `sameuser`, which tells PostgreSQL to give access to a database only when the user name and the database name match. Here's a line for `pg_hba.conf` that restricts users to same-named databases on the local 192.168 network:

```
host sameuser 192.168.0.0 255.255.0.0 password
```

Creating a database for the `cmag` user is easy:

```
$ createdb cmag
CREATE DATABASE
```

It is possible to specify a list of users for a given database by setting up a separate file with user names and, optionally, passwords. But PostgreSQL is not nearly as configurable for security as its rival MySQL, where all permissions are table-based. Fortunately, PostgreSQL 7.3 adds some improvements - you can specify multiple databases on one line

in `pg_hba.conf`, and you can also specify multiple users or user groups at the end of the line as `AUTH_ARGUMENT` data, or so I've read.

If you're stuck with 7.2, then have a look in the PostgreSQL docs for the `pg_passwd` utility, which you can use to maintain password files for individual databases.

Summary

PostgreSQL's authentication system can be a bit quirky, and you can easily run afoul of the default settings when you're trying to configure the database for ODBC access. You need to ensure that PostgreSQL is set up to listen on its TCP/IP port (by default 5432) and you also have to configure the `pg_hba.conf` file to allow specific remote connections. Once you have those two things sorted out you're ready to start playing with data using Clarion.

Oracle

REFERENTIAL INTEGRITY IN ORACLE

by Jon Waterhouse

When you move an application from flat ISAM files to a client-server architecture using a database like Oracle as a back end, it is not just a matter of setting up the Oracle tables and pointing your dictionary at them. One concern is making sure that what you want to be treated as atomic transactions are actually treated that way. Another is making sure that your referential integrity (RI) is enforced in the best way possible. This chapter will look at two aspects of migration that you will have to deal with in short order: referential integrity constraints and autonumbering.

Referential integrity

The referential integrity constraints imposed by the ABC templates boil down to the following:

- When you delete a parent record: the delete is disallowed (RESTRICT) or child records are either deleted (CASCADE) or have their parent link fields set to NULL (CLEAR)

- When you change the primary key on a parent record (which is referenced by foreign keys on the child records); the change is disallowed (**RESTRICT**), changed in child records (and possibly child relatives too) (**CASCADE**) or set to NULL (**CLEAR**).

When working with a back end database, such as Oracle, rather than flat files, you have the option to have the database server enforce these restrictions, rather than have the Clarion application enforce them. When working with Oracle, I prefer to let the server enforce these restrictions. One major advantage is that if there are other applications, developed in either Clarion or some other language, that access the same data, the RI rules will be applied consistently by the database, regardless of the application actually changing the data.

When you set up a relationship between files in the Clarion dictionary, the drop-down list has the options above, plus the same options with (Server) added. When you choose the **Server** options for looking after RI, your Clarion application backs off. It tries to save the current record that it has been asked to save, and if that works, it is done. Your program expects any other RI activity to be handled by the server. If you choose to enforce RI locally, then after successfully saving the current record the application will go on to try and make the required changes to all of the related child records. A failure in any one of these child files will lead to a rollback.

So, if you choose to enforce RI in Oracle, how do you go about it? There are two things necessary. The first is setting up a foreign key constraint on the child table in Oracle. For example, say you have a table ORDER_LINE and a table ORDERS, where the order_no field in order_line has to match ORDER.ID. You would implement the constraint as:

```
ALTER TABLE order_line
ADD CONSTRAINT order_fk
FOREIGN KEY (order_no)
REFERENCES order (Id).
```

This will ensure that when someone enters an order line it has an order_no that can be found in the ORDER table. It will also prevent anyone deleting (or changing the key value of) rows in ORDER that are referenced in ORDER_LINE. This is the same as the “on delete restrict” setting in the Clarion dictionary. For the restriction on order_line to work consistently it is also necessary to put a NOT NULL constraint on the order_no column. If this constraint is not present, and you enter a NULL value, Oracle won’t try to enforce the constraint, since it knows it cannot find a value that matches – in Oracle, NULL is not equal to NULL.

You can also add to the foreign key constraint one of these two options: ON DELETE CASCADE or ON DELETE SET NULL. These take care of the two other possible options for RI on deletes (Cascade and Clear) in addition to the RESTRICT option for both updates and deletes.

Oracle, in its wisdom, does not provide a built-in method for cascading changes on update. The theory is that a primary key should not change, so there should never really be any need for cascading updates, unlike cascading deletes, which you might need when, say, you are archiving records. You can achieve cascading updates in Oracle, but it's a bit tricky. There are basically two methods, both of which involve triggers.

Triggers are SQL functions that can be set up to carry out one or more other activities when a certain event happens. In particular, they can be set up to do extra things before or after inserts, changes and deletes. They can also be used to carry out something instead of a particular action. For example, a before delete trigger could be set up to mark a record as deleted (using a column in the data) rather than actually deleting it. One of the things that is not possible in a trigger is a commit, or any Data Definition Language (DDL) statement that would cause a commit. Thus anything that is carried out by the trigger is logically part of the same transaction. A `CLEAR` constraint on update can be accomplished very easily using a trigger. Here's an example:

```
CREATE TRIGGER clear_on_id_change
BEFORE UPDATE OF id ON order
FOR EACH ROW
BEGIN
    Update order_line SET order_no=NULL where order_no=:old.id;
END;
```

This trigger will only fire when `ORDER.ID` is changed. For all the rows where it is changed, the child rows that reference that parent row will have their `order_no` column updated to `NULL`. Obviously for this to work the `order_no` column should not have the `not null` attribute.

Following on from the approach above, it seems like a similar trigger could be written to accomplish cascaded updates:

```
CREATE TRIGGER cascade_id_change
BEFORE UPDATE OF id ON order
FOR EACH ROW
BEGIN
    Update order_line SET order_no=:new.id where order_no=:old.id;
END;
```

Unfortunately, cascading updates are not quite that simple. The `:old` and `:new` refer to the values before and after in the row you are changing in the `ORDER` table. However, this trigger will only work if you do something else first. By default the Oracle database enforces constraints immediately. The trigger as written will fire before (it's a before update trigger) the initial change is made. However, under normal circumstances it will fail because it is not possible to change the children's link (foreign key) field to a parent ID that does not yet exist since this violates the foreign key constraint. The trigger would also fail if it was declared as an after update trigger, because as soon as the update changed the parent value, the children (in `order_line`) would become orphaned, causing a rollback.

The way to get around this is to make the foreign key constraint DEFERRABLE, and to set the constraint to DEFERRED for this transaction (SET CONSTRAINT order_fk DEFERRED). This tells Oracle not to enforce the constraint until the commit happens, by which time you expect to have everything sorted out. Deferred constraints have been available since Oracle 8 (Current version of Oracle is 9.2). Before then a trickier method had to be used.

Oracle wrote a little package (http://asktom.oracle.com/~tkyte/update_cascade/) to make this slightly tricky logic easy. The package uses a series of triggers and procedures so that the initial change (of the parent id) does not really happen. What happens instead is that a new row (with the new id) is inserted into the parent table. The child rows are all updated to become children of this new row, and then the old row (which no longer has any children) is deleted. This only works if you are updating the parent to a primary key value that does not yet exist in the table. In my experience about the only reason you would ever have to change a primary key is when you want to change it to a value that already exists (e.g. when one real customer has two data rows with slightly different names and you want to consolidate the two customers into the one they should have been). This is not something that can be accommodated by this package. However, changes of this sort can still be accomplished using the deferred constraint method (assuming you are working with Oracle 8 and up).

In summary, the equivalent in Oracle of a RESTRICT RI constraint is accomplished by setting up a foreign key using a NOT NULL column. The Cascade and Clear constraints for deletes can also be built into the foreign key constraint. Update constraints, if you really need them, require triggers.

Next, I'll continue this discussion with a look at autonumbering.

AUTONUMBERING IN ORACLE

by *Jon Waterhouse*

In the previous chapter I began a discussion of Oracle's Referential Integrity (RI) capabilities. Now I'll continue that discussion with a look at autonumbering.

There are two benefits of having the database enforce the RI constraints. The first is that your data is protected from some other application (or user armed with SQL*Plus) making changes that are inconsistent with referential integrity. Secondly, Oracle can carry out the required operations many times faster than a Clarion program can. For cascaded deletes, for example, a Clarion program will retrieve each of the child records one by one and then ask Oracle to delete each one. Oracle will get rid of all the unwanted records at the same time.

The downloadable example at the end of this chapter demonstrates the advantage of using Oracle constraints rather than relying on Clarion to enforce RI. The program first sets up a PARENT table by selecting 100 rows from the ALL_OBJECTS view. The table has a primary key of OBJECT_ID, which is numeric. It then creates a CHILD file by inserting 50 records for each key value in the PARENT table. The constraints on the tables are created by an `alter table` command. The program presents three methods of deleting all of the rows:

- A Clarion process using a Clarion “On delete cascade” constraint loops through the file deleting the parent records and their related children (the Oracle constraint is disabled)
- A Clarion process loops through the file deleting the parent records; the child records are deleted by the enabled Oracle foreign key constraint
- The records are deleted by a call to a single small SQL statement

Just to add a slightly more real-world dimension to the process the deletes take place based on the modulus of the `Object_ID`: first all IDs ending in zero are deleted, then those ending in 1, etc.

On my system the process using Oracle RI delete took about 8.26 seconds per 100 parent records (5000 child records). The single SQL statement with Oracle handling the RI took 5.7 seconds per 100 parent records. The Clarion method did not work at all. The child records related to the first parent ID were deleted fine, then Clarion got confused about which ID it was working with and failed with a “Record Not Found” error.

Autonumbering

Creating primary keys using autonumbering is something that Clarion programmers have become very comfortable with over the years. Many Clarion developers may not even realize that there are many SQL databases where primary keys are not autonumbered arbitrary values, but consist of one or several entered fields. Partly, Clarion’s autonumbering was a response to the quirkiness and poor performance experienced when dealing with multi-part keys in Clarion data files. In the SQL world autonumbering to produce the primary key by which a record is linked to other data is frowned upon by relational purists, but tolerated. The theory is that each table is an entity, and there should be attributes of each instance of the entity that make it distinct. Autonumbering is viewed as a bit of a cop-out. Still, most databases provide you with methods for autonumbering. In Microsoft’s SQLServer there is an `IDENTITY` column attribute that handles autonumbering. In Oracle autonumbering is implemented using a *sequence*.

What happens if you use Clarion to autonumber the column you use as the primary key of a table? The `ABC PrimeAutoIncServer` method (which is called when you enter a form in insert mode, or when you choose Insert on a browse) does the following: queries the database to find the highest existing key value, increments it by one and inserts a record with the new ID number and all the other fields “blank” or with default values.

There are two major problems with carrying out this process in Oracle. The first is that “blank” records often cause problems. If you design your database properly many fields in your tables will not allow NULLs. For example, your order table should always require a

customer; your order item should always have a price, etc. You *can* get around these constraints when you add your “blank” record by making up default values, but the price will be shoddy, error-prone data. You can save records with nonsense values for fields that should always have real values, but there is always the possibility that some of these crud-filled records will take up permanent residence in your table. The second problem with this technique is poor performance with a large number of users. If there are a lot of people inserting into a table, there are going to be a lot of requests to read one thing: the last value in the index. This is also an area of the disk that is going to be written to a lot: each time a new record is added there will be changes made specifically to that part of the index. Grabbing the next value from a sequence is, by comparison, very fast.

The Clarion method also has another built-in problem that will slow it down on busy systems. The insert is carried out in two separate operations: first the maximum value is retrieved from the table, then a record is added using the incremented maximum value. The possibility exists that another user can retrieve the same maximum value from the table *before* the first user gets around to inserting and committing its new record. This second user will then try to insert a record with the same key value as the first user has already added. This insert will be rejected. The `PrimeAutoIncServer` method gets around this problem by carrying out the whole process again if it fails the first time. By default it allows three tries. This is not just a theoretical problem: it happens in practice. In the example application I have the number of attempts allowed set at 6: try reducing that to 1 and see how many records fail to get added. On my system it was close to three percent. Although this scheme manages to get records numbered and added, in the real world, if you have an application that responds to heavy usage by increasing the work the database is asked to do, you are asking for trouble.

Theory therefore says that using the Clarion method of autonumbering will slow down your application rapidly as the number of users increases. The source that accompanies this chapter tests out this theory. Just for fun it also looks at what happens if you use Topspeed files for the same sort of load.

Before describing what the program does, I'll explain how you *should* do autonumbering in Oracle.

The way to handle autonumbering in Oracle is, as I said, to use a sequence. The sequence exists completely separate from your table. Its only function is to provide the `Nextval` in the sequence when it is asked. It may occasionally skip numbers, but it will never provide a duplicate value. When you want to insert a record you ask the sequence for the next value and put this value in the appropriate column. There are two ways of accomplishing this in Clarion. The first method is the one described in the Clarion help file. First your program asks for the `Nextval` in the sequence. When you save the record you use this value as the ID. This requires setting up a dummy table in Clarion with one field (a `ULONG`) to receive the value from the sequence. Your program gets the value passed to it using `PROP:SQL`, for example:

Oracle

```
Dummy_table{PROP:SQL}='SELECT order_seq.nextval from dual;'
```

The second method is to set a trigger up in Oracle, for example:

```
CREATE OR REPLACE TRIGGER AUTONUM_ORDER
BEFORE INSERT ON ORDER FOR EACH ROW
BEGIN
    SELECT order_seq.nextval into :new.id from dual;
END;
```

If you have more esoteric autonumbering needs (incrementing numbers for the second or third parts of keys), the Clarion method may start to look more attractive. However, the “add an empty record” Clarion method still has problems, as I noted earlier. The pure Oracle alternative would be to maintain a sequence for each of the numbering subsets. My guess is that any numbering scheme involving more than one field will cause more problems than it’s worth.

Testing the theory

The theory suggests that the larger the number of contending users, the worse the performance of the Clarion scheme will be compared to the Oracle sequence scheme. The example program is set up to test this by taking a fixed number of records to add and looping through a series of “users”. Each user gets to add their segment of the records. For example, if the number of records to add is 1000, the program first spawns one “user” to add 1000 records, then two users to add 500 records each, then three to add 333 records each, and so on.

The actual inserts into the database are carried out by addrecords.exe. This is a little project-generated executable that is passed the number of records to insert, and the scheme for generating the id number. The records are added in a tight loop with a YIELD statement. The main program calls RUN addrecords for each user. A statement in a timer checks to see if all of the (1000) records have been added for that round of the loop. If so it goes on to the next round of the loop (with an increased number of users). To be more precise the check is to see if at least 97% of the records having been added: the number that are actually inserted can be less than the full amount both because of integer division (three users would add only 999 records) and because you *may* have some records not added because of the duplicate value problem (in the Clarion scheme). In pseudo-code what happens is this:

```
Tot_recs_to_add=1000
Num_users=1
:start
User_recs_to_add=Total_recs_to_add/num_users
Loop k=1 to numusers
    Run addrecords(user_recs_to_add)
```

```
End !loop
Check 5 times a second
  If at least 970 new records have been added
    Num_users=Num_users+1
  If Num_users > 10 then finish else go back to :start
  End if
```

The slightly complicated structure is required because while you want some of the actions to run concurrently (e.g. four versions of addrecords running to simulate four users), you don't want to start the five user scenario until the four user scenario has terminated.

Times can either be checked by calls to `clock()` at the beginning and end of the outer loop, or by looking at the times of the records added (one of the columns stored in the test table is the add time). I have used the first method.

On my machine, the practical results are as follows:

- Using the Clarion autonumbering: 1 user takes 5.69 seconds to add 1000 records; 10 users take 9.34 seconds
- Using an Oracle sequence to handle the numbering: 1 user takes 5.58 seconds; 10 users take 8.33 seconds

The practical test does therefore validate the theory: while the sequence method is only 0.1 second faster for a single user, the advantage rises to 1.0 second when you reach ten users (which is a greater than 10% speed advantage). The tests were run on an HP clunker running NT4. Times are not that precise since the main timer only checks for completion five times per second.

Just for fun I set up a similar test to work with Topspeed files. If you need an example of how badly Topspeed files perform relative to a real database under multi-user access, this will do it for you. While using stream and flush is significantly faster than can be achieved using Oracle (about 1 second for 1 user), using `LOGOUT` and `COMMIT` around each transaction will kill performance when you add just the second user.

And not coincidentally, transactions are the subject of the next chapter.

Source code

See “Appendix A: Getting Support,” p. 601, for information on how to get the source accompanying this book.

- `v5n1oracle.zip`

TRANSACTIONS IN ORACLE

by Jon Waterhouse

In the previous chapter I discussed autonumbering in Oracle. Now it's time to look at another important issue in Oracle (and other) databases: transactions.

Transactions are units of work that take a database from one consistent state to another. Whether your back end is an Oracle database or Clarion TPS files, one thing you want to prevent happening is having a database end up with half a transaction in it. The classic example is a transfer between two bank accounts. One part of the transaction is to subtract the money from the first account; the second part is to add it to the second account. Either both these actions should take place, or neither of them. The second thing that is important in transactions is how they incorporate changes made concurrently by other users. For example, if you are trying to calculate and store bank balances, and there are transactions happening at the same time at banks and ATMs across the country, which information is incorporated and which is discarded?

Clarion has three commands that are designed to set up transactions, which are translated into different commands for each back end. The Clarion commands are LOGOUT, COMMIT and ROLLBACK. In Clarion the set of actions carried out between the LOGOUT and the COMMIT is the transaction.

Oracle

In Oracle, everything between one commit and the next commit is a transaction; there is no equivalent of a LOGOUT statement (this is not true in several other SQL databases). There are also some activities (e.g. involving the DDL, such as creating tables and altering table structures) that implicitly force a commit. Rollbacks occur either when errors are encountered or a ROLL BACK statement is encountered.

There are two other statements that control transactions in Oracle; SAVEPOINT allows you to break a transaction into parts so that if a rollback happens it the database is rolled back only to the savepoint (rather than the last commit), and PRAGMA AUTONOMOUS_TRANSACTION is a stored procedure directive that allows transactions within transactions. One use of this is to allow a logging procedure to document what someone attempted to do even if the attempt ultimately failed and was rolled back.

The points in a Clarion program when transactions are most important are when inserting, deleting and modifying data in the database. Client side autonumbering on insert is a potential additional source of data modification, separate from the “real” insert. Autonumbering managed by Clarion is not a good idea in Oracle databases, as I explained in the last chapter. In the browse-form paradigm inserts, changes and deletes mostly happen when a form is completed.

The other major place where data modification takes place is in batch transactions carried out in process procedures. If you are writing with an Oracle back end in mind you will likely never use process procedures: most processes will translate into a single SQL statement. In those instances where row-by-row processing is still required you will likely set up a cursor in a stored procedure to handle this rather than use a process in Clarion.

If the batch process is translated into a single SQL statement the transaction will include (at least) the entire batch process; if an error is encountered in one row, *none* of the changes will take place. The concept of putting a LOGOUT/COMMIT around an entire process is a bit of a foreign concept to a pure Clarion programmer. LOGOUT/COMMITS are sometimes inserted within process procedures to process (say) 100 records at a time, but this is normally done to improve performance. In Oracle, the rationale is mainly that it is a lot easier to deal with a transaction that fails entirely, than with one that alters half (or some of) the rows, and ignores others, leaving you to figure out into which category a particular row falls.

A second method of implementing the functionality of a batch procedure is to use a cursor. If you use a cursor you have a choice of whether to commit each row as it is processed, or to treat the entire batch as a transaction. There are still some difficult questions to work out. For example, suppose you are adding a new employee to your HR database. He has qualifications from some foreign university that you have to add to an existing list. When it finally comes time to save your new employee record in the database, you find you can't do it because you don't have his social insurance number (which is not allowed to be NULL). Should the qualifications you entered be considered part of the whole transaction and rolled

back, or should the addition to the qualifications list be considered a completely separate transaction? By default, in Clarion it would be a separate transaction; in this case, this is probably not a bad thing, in other cases, you may want to consider some of these ancillary changes as part of one multi-faceted transaction.

The ABC templates that ship with Clarion try to deal with two transaction-related issues: concurrency control and referential integrity. I began a discussion of referential integrity earlier; the following section will discuss concurrency issues.

Understanding how Clarion deals with basic concurrency issues will help you understand how more complex transactions can be managed.

Concurrency control

Concurrency control has to deal with two users making changes to the same record. Oracle will happily give any number of users a copy of a particular table row to look at. If more than one of those concurrent users decides she wants to make changes to the record and save those changes back to the database, you have a problem. There are two ways to deal with this. One is for the program to lock rows where it is likely that the user will update it (e.g. when the user opens a form) so that someone else who comes along is told the record is in use if they also try to update the same row. The second is to check just before changes are saved to make sure that someone else has not changed the row while we have been preparing our changes. I guess there is also a third way, which is let people save whatever they want without any checks, but users don't react very well to finding things unchanged that they *know* they have changed (but someone else wrote over their changes afterwards in changing something else in the row at the same time).

The two solutions described above are known as pessimistic and optimistic locking respectively. Pessimistic concurrency says it's fairly likely someone else will muck with my data before I'm ready to save, so I should lock it now; optimistic concurrency says it's unlikely anyone else will have done anything to my record, but I'll check just to make sure, and will accept the risk that I'll have to re-enter my changes if that happens.

The first method (pessimistic) is accomplished in Oracle by requesting data with a `FOR UPDATE` clause. Other people can still read the data, but they can't acquire the row `FOR UPDATE` themselves, nor can they update or delete the row without locking it first (which would be denied). In Clarion, to implement pessimistic locking, when the form opens the `SELECT` statement would have to select the row with the `FOR UPDATE` clause. You could accomplish this by using the Clarion `SQLCallback` feature, but that isn't something I've tried.

To implement optimistic locking, just before saving changes, the application should re-read the data, check to see that the row still looks like it did when it was originally grabbed (giving you an error message if not), then save the changed row. Ideally, it should do this all in one operation so that there is no chance of a change happening in between. In SQL this can be accomplished by a statement like:

```
UPDATE mytable SET changedfields=:newvalues  
WHERE changedfields=:oldvalues and allotherfields=originalvalues
```

This is what a standard Clarion application will do. Slightly less reliable, because there is the chance that a change will be made in between the check and making your change, is to carry out the check to make sure the row still looks the same first, and if that's okay, then make the change.

Up to this point I have been a bit slack in my use of the word *save*. Using flat files you think about things the following way: you grab the record off the disk into the record buffer in memory; you make the changes in memory; you save the changed buffer back to the disk. Game over. Anyone else comes along afterwards and reads from the disk, they see the changed record.

With Oracle, that is not true. If I (User1) grab (SELECT) a record for update and make a change using the UPDATE command (e.g. UPDATE Customer set firstname='Jon' where lastname='Waterhouse' and firstname='John') and you (User2) now come along and query the database to fill your browse with Select id, firstname, lastname from customer where lastname like 'W%' you will still see the John Waterhouse row. You will continue to see the John Waterhouse row, and not the Jon Waterhouse row until I (User1) COMMIT my change (and maybe not even then – see below). I on the other hand, filling my browse with the same query *before* I commit will see the row as Jon Waterhouse and *not* John Waterhouse. I made the change; I get to see the change. But no one else gets to see the change until I decide I'm happy with it (and any other changes I might have made), and COMMIT So, when I have used the word *save* in the previous discussion on locking, what I meant was update *and* commit.

With this extra bit of knowledge, let's revisit the discussion on locking. If my Clarion application does not use the FOR UPDATE clause to implement locking, and there is a large delay between UPDATE and COMMIT carried out by User2, then a simple re-query before the UPDATE accomplishes virtually nothing; I would only see the record had changed if my own session had changed it, or (possibly) if user 2 had got around to committing her work.

Twice now I have mentioned that even if another user commits, my session might not see those changes. Why is this? In some really complicated transactions (say the calculation of accounting balances based on all entries in an accounting system) you may try to read the same data twice. If the second time you read the data you get back different information

than the first time this may make it impossible for your calculations to produce a result that adds up. This sort of transaction needs to be protected from reads that are “non-repeatable” (i.e. because someone else has changed the data meantime) or “phantom” (someone has added a record that didn’t exist the first time we read the data). There is a third sort of read, called “dirty”, which allows you to see uncommitted data, but Oracle does not allow this under any circumstances. The Transaction Isolation Level determines what sorts of reads are allowed. The two choices in Oracle are Read Committed, which allows both non-repeatable and phantom reads, and Serializable, which allows neither.

That said, what does Clarion actually do? Does it issue FOR UPDATE selects? Is data committed immediately after any and all SQL statements that update data? What transaction isolation level is used?

In the standard browse-form application, when you select a record for update, the `TakeEvent` method of the `BrowseManager` first calls the `Window.Update` method, and then the `Ask` method calls the update form procedure. It is the `Windows.Update` method that makes sure that all of the buffers for the browses on the window are refreshed from the underlying views. This translates into a straight `SELECT * (all columns)` statement. The process is summarized in the following table:

In the browse, when you select Change on the browse>	
<code>BrowseManager.TakeEvent</code>	Chooses what to do based on which control has been accepted. In the case of the Change button, does the below:
<code>Window.Update</code>	Calls <code>UpdateViewRecord</code> for all the browses on the window, which GETs the current record
<code>BrowseManager.Ask</code>	Opens the form defined as the update procedure

In the form, when you click the OK button (for an update)	
<code>Window.TakeCompleted()</code>	Calls different routines depending on whether the form was called to insert, change or delete

RelationManager.Update	Initiates the logout (translates into setting the transaction isolation level to Serializable, and turns off auto-commit)
FileManager.TryUpdate/ UpdateServer	Tries to save the primary record using PUT. This translates into an UPDATE statement in SQL; passes errors, concurrency errors in particular, back to the caller
RelationManager.Update (continued)	If saving the main record succeeded, and RI is to be handled locally, calls the SecondaryUpdate method (which may be called recursively) to update the child records if the linking value has changed.

In the form procedure, when you press the **OK** button, a complicated sequence of Window methods is called. For purposes of this discussion, it's sufficient to note that if everything goes according to plan the TakeCompleted method calls a routine based on whether the form was called to update, insert or delete. In the case of an update, the method checks to see if any changes have been made, and if so it calls the SELF.Primary.Update() method, i.e. the RelationManager method for the primary file. This method in turn uses the UpdateServer method of the underlying FileManager. At the beginning of the RelationManager method, a LOGOUT is performed (what happens here is explained below), and in the FileManager method an SQL UPDATE statement is prepared as a result of the PUT() statement of the form:

```
Update mytable set changedfields=:newvalues
  where changedfields=:oldvalues
    and allotherfields=originalvalues,
```

That is, Clarion implements an optimistic concurrency check. If this statement succeeds the RelationManager method goes on to make changes to child files (if the RI enforcement is local and not on the server). Finally, at the end of the RelationManager.Update method, assuming that no errors have been encountered at any stage, the Clarion COMMIT statement changes the isolation level back to Read Committed and turns auto-commit back on, which effectively commits the transaction.

If that description was a bit much, then here is just the bottom line. Clarion uses optimistic concurrency checking. The scope of a transaction is from the beginning to the end of the RelationManager method; in the middle of the process a call to the FileManager

method actually writes the changes. Changes are committed at the end of the RelationManager method. The transaction isolation level is *serializable*.

One of the advantages of Clarion is that you can use the same syntax and have the database drivers convert your Clarion statements into statements appropriate for the particular file driver. And even with Oracle as the back end, you have (with the Enterprise edition) a choice of file drivers to talk to Oracle, the generic ODBC driver and the Oracle Accelerator.

ODBC, by default, sets auto-commit on. This means that every statement sent to Oracle is implicitly followed by a commit. With auto-commit on there is no way that you could implement a consistent multi-statement transaction. Thus, one of the most important things that the LOGOUT statement does is to change the ODBC Connect environment setting to turn auto-commit off. This is important to know. If, for example, you wanted a statement to be part of the same transaction as the main statement, and you placed it *before* the RelationManager.Update procedure, since auto-commit is, by default, turned on, this statement would auto-commit. If the main statement later rolled back because of errors, your initial, pre-logout change would still be stored in the database.

The second ODBC operation carried out by the LOGOUT is to set the transaction isolation level to SERIALIZABLE. By default, Oracle works with an isolation level of READ COMMITTED. In most cases the change in isolation level won't make any difference to you, as changes to the one record you are working with are dealt with by the concurrency check; it is only if your transaction includes reading other data that this will affect you.

Oracle aficionados will tell you that the reason ODBC defaults to auto-committing every update is that it's a Microsoft design, and that Microsoft's SQLServer quickly falls to its knees when transactions remain uncommitted because it can't handle large numbers of unreleased locks. They also maintain that pessimistic concurrency checking (which entails locks being held for longer) is a more sensible approach than optimistic checks, but again that optimistic checking is chosen because of the locking problems of non-Oracle databases.

When you are using Oracle Accelerator the only real difference in the processing of transactions compared to the ODBC method is that the transaction isolation level stays at read committed.

Similar events take place for inserts and deletes.

The important thing to understand about the way that inserts, changes and deletes are accomplished in Clarion is that if you want to make other changes as part of the same transaction, it has to be between the LOGOUT and COMMIT statements, and this means your code has to happen in the middle of the existing ABC RelationManager (RM) methods. The obvious place to add code is after the FileManager (FM) method that is called by the RelationManager method. The relevant methods are the TryUpdate,

`TryInsert` and `DeleteRecord` methods. Your additional code would first check to make sure the main action happened, then make additional changes.

It is quite possible that you want to add different things into the transaction depending on the context. For example, if the application maintains a transaction journal file, which is the main file that is updated by a few screens for varied purposes, the context somehow has to be communicated to the method. This might involve adding a parameter to the `FileManager` methods and the methods that call them.

I should note that the above approach is based on the C5.5 templates, which make the basic change (additions to all transactions on a file irrespective of context) relatively easy to accomplish. In the C5 templates things are not quite as easy. This is because the insert and delete methods call the `FileManager UpdateServer` and `InsertServer` methods, which are private, and the `RM.Delete` method issues its own `DELETE` command, while in C5.5 the new `FM.DeleteRecord` method is called.

Getting more complex

At the beginning of this chapter I brought up the classic transaction of the transfer of money between two bank accounts. I still haven't got beyond the simple transactions that Clarion manages itself, plus some alternatives in getting Oracle to manage some of the RI. Before going on to discuss multi-element transactions, I'd just like to mention one other Oracle feature that may help you avoid entering this realm. Clarion, in its documentation of views, says:

PUT only writes to the primary file in the VIEW because the VIEW structure performs both relational Project and Join operations at the same time. Therefore, it is possible to create a VIEW structure that, if all its component files were updated, would violate the Referential Integrity rules set for the database. The common solution to this problem in SQL-based database products is to write only to the Primary file. Therefore, Clarion has adopted this same industry standard solution.

However, if your view is declared in Oracle, you will find that you *can* directly update most of the columns in the view, whether they are in the primary file or not. There are some exceptions. For example, if your view contains a "virtual column" calculated from one or more table columns (for example, the view contains a `NAME` column that is based on concatenating the `FIRSTNAME` and `LASTNAME` columns in a table), you cannot directly update it. With an Oracle view set up, you can use the view as the "file" for a form, and Oracle can update the view fields you change on the form regardless of the underlying table they come from. In some cases this can dispense with the need to set up multiple data update statements in a transaction.

Here's an example. You have three tables: MACHINES, PARTS_LIST and PARTS. You could set up a view in Oracle like this:

```
CREATE VIEW v_machine_cost AS
SELECT m.machine_name, p.part_name,
l.num_parts,p.unit_cost,
p.unit_cost*l.num_parts extended_cost
FROM machines m,parts_list l,parts p
WHERE m.id=l.machine and l.part=p.id
and m.id=:machine_of_interest
```

A form based on this view would allow you to change how many of a particular part is required for the machine and the unit cost of the part. Oracle is smart enough to translate this into a change in the PARTS_LIST table for the first and change in the PARTS table for the second. This gets rid of the need to write two update statements arising from the changes on one window.

Summary

In these chapters I've covered some of the issues involved in moving an application from flat ISAM files to a client-server architecture using a database like Oracle as a back end. As should be clear by now, it is not just a matter of setting up the Oracle tables and pointing your dictionary at them. Both referential integrity and autonumbering are essential aspects of migration.

MS SQL

MIGRATING THE INVENTORY APPLICATION TO SQL SERVER

by Ayo Ogundahunsi

The amount of data being processed in corporate databases and over the Internet has created a demand for more powerful engines for data storage, access, and processing. While Clarion is a powerful RAD tool, a necessary complement is an excellent back end SQL Database.

In this chapter I will attempt to reinforce information contained in earlier chapters. The emphasis of this series is on portability, business rules, referential integrity (RI), and Clarion as an interface tool. More specifically:

- If you are the designer of the overall application, i.e. you are responsible for creating the original tables, the business rules and logic of the application must be moved to the back end so that a non-Clarion application will be able to use your logic. I call this approach *non-isolationist* in the sense that components, for instance your data types, will be directly accessible to other systems. For example, you do not use a LONG instead of a DATE data type

so that updates from within Clarion will not differ from updates outside of Clarion.

- If your application will be accessing an existing database, then you will still try to put most of your business logic/rules at the back end, i.e. in stored procedures.
- When RI is enforced at the back end, you greatly minimize the risk of bad data entering into your database. This becomes more evident where other applications input data to be used by your system.
- Clarion moves away from its role as the complete database application and into a role as an interface tool. Extensive data manipulation is done at the server using stored procedures called from within Clarion.

Some may argue that the points mentioned above reduce control you have over the customer, especially in the case of contractors. Nevertheless, it is a matter of how you view the services you are offering. Do you want to deploy a system where all the components can *only* run with Clarion? Or, you want to deploy a system for which someone can easily code a Visual Basic interface? It is a business decision. Note however that times are changing, and customers are starting to look at more portable systems, so you may not want to be overly proprietary in your design.

On the marketing side, the more you emphasize on the fact that the logic of your application resides in SQL, the easier it becomes for you to enter into the arena controlled by other languages like Visual Basic, PowerBuilder, etc.

Existing resources

There are many articles in *Clarion Magazine*, including several published in this book, that have treated similar conversions extensively; please make sure you refer to them since I'll assume some prior knowledge. In particular you should be familiar with the following material:

- Scott Ferret's "How To Convert Your Database To SQL," p. 175.
- Stephen Mull's "Converting TPS To MS-SQL," p. 183
- Rick Hoffman's MS-SQL Tips and Tricks and C5 (Rick's original site is no longer up, but you can still get this article via the Wayback Machine:
<http://web.archive.org/web/20020816013033/http://home.tampabay.rr.com/rhoffman/MSSQL-C5.document.htm>

Stephen Mull's chapter (a must read) covers a lot about conversion and contains most of the tips and tricks explained Rick Hoffman's article.

See the end of this chapter for some non-Clarion resources.

Getting Started

There is an Inventory example that comes with Clarion. It is located in:

C:\Clarion5\Examples\INVNTORY

for Clarion 5, or in

C:\C55\Examples\INVNTORY

for Clarion55. To follow along with this chapter, make a copy of this directory to C:\INVNTORY. All the examples will be based on this directory.

I will be using the terms Columns and Fields interchangeably, also Rows and Records. Columns and Rows are SQL terms; Fields and Records are the Clarion equivalents.

The conversion steps on the Clarion side are as follows:

- 1) Dictionary/Application Changes
- 2) Create a copy of INVNTORY.DCT, INVNTORY.APP.
- 3) Change the Table driver, properties.
- 4) Add Identity fields.
- 5) Change Data types (if needed. E.g. LONG to DATE).
- 6) Remove Initial Values from Clarion dictionary. If TODAY() is used and is needed, remember to define a DEFAULT in SQL Server as explained under the DEFAULT section in this chapter.
- 7) Remove GROUPS if used (in SQL tables you will only use GROUPS to translate the SQL DATE/DATETIME/TIMESTAMP data types to their Clarion equivalents).
- 8) You will also need to change STRINGS and MEMOS to CSTRINGS.
Remember to add 1 to the size of your field when using CSTRINGS. Do not use the LONG data type for date fields; use DATE instead.
- 9) Delete procedures in APP file.
- 10) Template/Classes
- 11) Auto Incrementing - The Jim Kane Solution

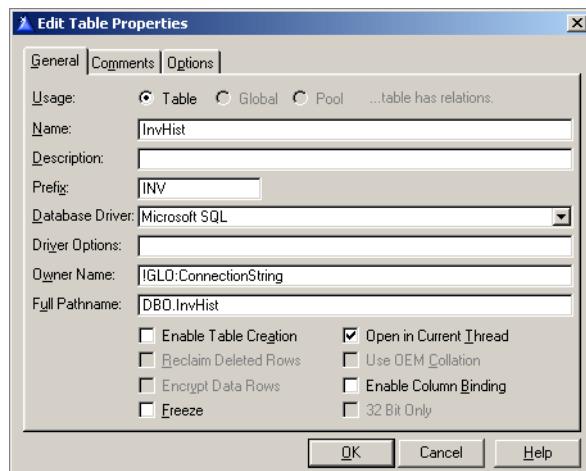
- 12)Connections**
- 13)Connection String**
- 14)Database Creation**
- 15)RI in Clarion Dictionary Fields**
- 16)Default values in Fields**
- 17)Script generation (Synchronizer)**

Step 1 - Dictionary /application changes

You have to load the Inventory dictionary located in C:\INVNTORY and save it as a new dictionary under the name INV_SQL.DCT. Repeat this step for the application file: INVNTORY.APP saving it as INV_SQL.APP. One thing you shouldn't forget to do is to change the dictionary in INV_SQL.APP to INV_SQL.DCT.

The next thing is to modify the table properties by changing the **Driver** to **MS SQL Server**, Specifying an **Owner**, and the way the table is named on the back end. Figure 1 shows what the InvHist Table looks like after this step is complete.

Figure 1: Changing the InvHist table settings



The fields that have changed are the driver, the owner, the full pathname, and the **Enable Table Creation** checkbox.

Owner Name: (!GLO:ConnectionString)

The owner name is the label of the connection string (see explanation on **Connection String** below). You will have to define this as a Global variable either in the Dictionary or in the application. The ! prefix tells Clarion this is a field and not a string literal.

Full Pathname: DBO.InvHist

The MS SQL back end recognizes the pathname like this. DBO stands for Database Owner, and InvHist is the name of the table. For now, I will stick to this simple approach to ownership. A complete understanding of database ownership and roles is beyond the scope of this chapter.

Enable Table Creation: Unchecked

Enable table creation is unchecked deliberately. It is better you create the tables with the back end than through Clarion.

System IDs and identity columns

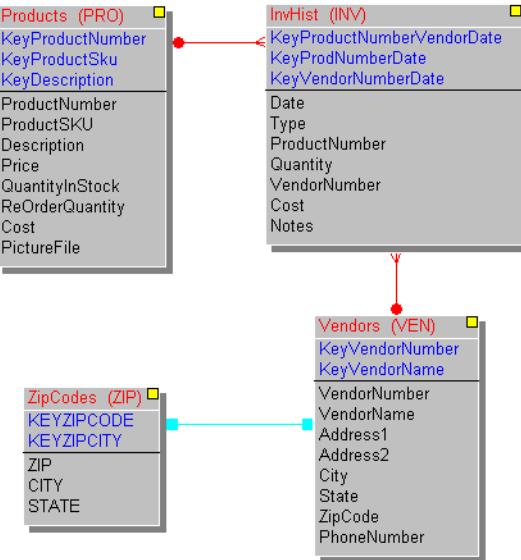
Properly designed tables usually have fields (columns) designated for storing the identity values of rows. The identity value is unique for each row in a table, and can be used to link child records in other tables. This subject has been already covered by Dave Harms in “Designing Databases,” p. 1).

If you load the Inventory dictionary into the Data Modeller, you can see that there are no identity fields defined. Products does have a ProductNumber, and Vendors has a VendorNumber, which are presumably unique in each table, but these data could also

MS SQL

change. It's better to have a unique, autoincremented number which will never need to be changed.

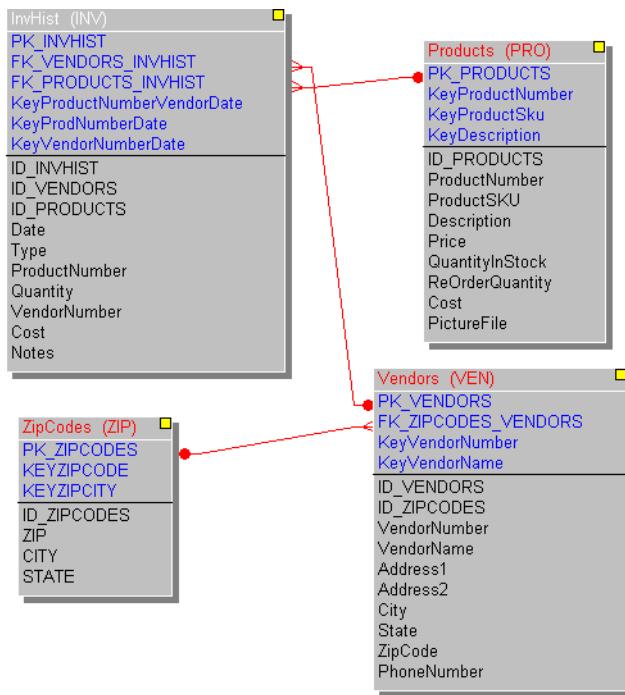
Figure 2: The Inventory tables without identity fields



Migrating The Inventory Application To SQL Server

After adding the identity fields, I have something like this:

Figure 3: The Inventory fields with identity fields added



A couple of things to note here about naming and key attributes. The naming convention is up to you. Some people use `SysID`, some `NameOfTable_ID`. I prefer using `ID_NameOfTable` for the simple fact that in a very big database, you can immediately see the linked tables, sort the IDs and put them together in a big table, etc. Also, if you are using the MS SQL Server View Designer (more about this later), once you populate the screen with tables, the designer automatically connects the tables together for you. This is quite helpful and makes your work a lot easier. I use Crystal Reports to print out my database structure, and I have the field names sorted, so I have all the link fields (those starting with `ID_`) all together.

I used the prefix “PK” to indicate Primary keys, and “FK” to indicate Foreign Keys. It is always better to use a naming convention like this because when you synchronize with the MS SQL Server, you will find the Table design interface in the Enterprise manager easier to use.

I can also go further and rename the other keys like `SK_VendorNumber` where “SK” means sort key, or `UQ_VendorNumber` where “UQ” means unique key. It is good to

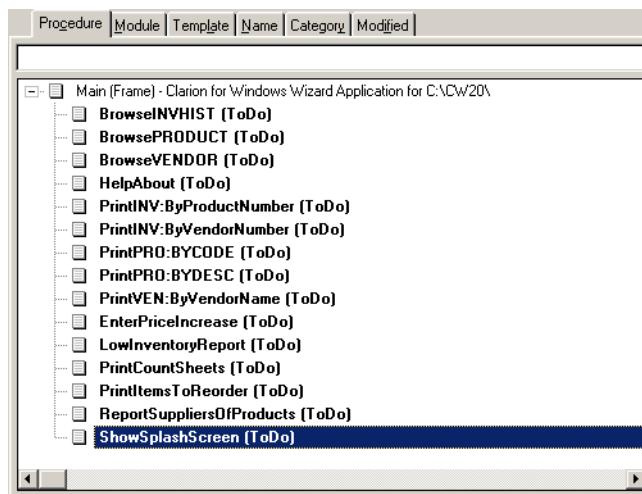
distinguish these keys in a way you can easily recognize because it quickly gives you an idea about how the key was created.

In setting the attributes of the keys make sure that all your keys are case-sensitive. If you don't do this, performance will probably suffer significantly.

Deleting the application procedures

All procedures in the application file `INV_SQL.APP` except the `MainFrame` procedure are to be deleted. Yes, deleted. Once you are done with initial stages of conversion, you can import the same procedures from the original application file (`INVNTORY.APP`). This makes conversion faster.

Figure 4: Deleted procedures



Auto Incrementing

In short, the problem with SQL auto-incrementing is that while it's safe and efficient to have the server create the auto-incremented ID for a new record, you won't have that ID available to you when you want to add child records to the record you've just created. Read the chapter "An Introduction To SQL," p. 141 for background on the auto-incrementing problem.

There are different ways to solve the auto-incrementing problem; I will use the solution provided by Jim Kane (which is an adaptation of an initial solution by Scott Ferret). You can download Jim's code here:

http://www.icetips.com/ftp/old_stuff/sqlan55.zip

From the downloaded file, follow these steps (culled from the readme.txt file) to install the code:

- 1) Copy SQLAN.TPL to the template directory and register SQLAN.TPL
- 2) Copy SQLAN.CLW and SQLAN.INC into the Clarion Libsrc directory
- 3) In ABFile.inc add two methods:

```
SetAutoIncDone Procedure (BYTE pAutoIncDone)
GetAutoIncDone Procedure (), BYTE
```
- 4) In ABFile.CLW, add the code for the two methods mentioned in 3 above.
The source code is in ABFix.CLW.
- 5) Add the Global Extension to your application.
- 6) You are to create a table called 'dummy' in SQL Server. (The script to do this has been added to the example SQL Script.)

```
CREATE TABLE DBO."Dummy" ("dummy_col" INT)
```

Connection String

The Connection String is a string sent to a database server that allows your client machine to access the database. This is separate from the normal network rights given to you to connect to the Server. In order to understand how the connection string is used, it is important to understand the security features in SQL Server.

Security in SQL Server can be implemented using Windows Authentication Mode, or Mixed Security Mode.

- Windows Authentication works only on operating systems that use Windows NT authentication; you cannot use this with Windows 9x or Millennium servers. As clients, these operating systems must have a trusted connection to the Windows NT/2000 Server. This is because SQL Server allows connections based on the user account name or group membership available in the Windows domain by mapping logins from a trusted NT Domain into SQL Server.

MS SQL

- Mixed Security Mode: In mixed security mode, a user is given access by Windows Authentication or SQL Server authentication. If SQL Server authentication is used, then the password and username is maintained by SQL Server.

Remember when you are installing SQL Server to choose Mixed Mode (not Windows Authentication Mode). As you do this, you will be required to enter a password; this password should be `sa`.

The connection string is made up of the server name, the database you are connecting to, your username, and password. It is in the form:

```
'AYO2000-OFF, INV_SQL, sa, sa'
```

where `AYO2000-OFF` is the name of the Server, `INV_SQL` is the Database name, `sa` is the username, and `sa` is the password.

If SQL Server is running on your local machine, you can use this connection string:

```
'(LOCAL), INV_SQL, sa, sa'
```

SQL Server 2000 was released with support for multiple instances. This means you can run multiple copies/different versions of SQL Server 2000 on the same machine at the same time. If you install SQL Server using the instance feature you have to qualify the server name. So, assuming you created an instance called `AYO_NV01` during setup, and your machine/server name is `AYODAHUNSI`, then your connection string will look like this:

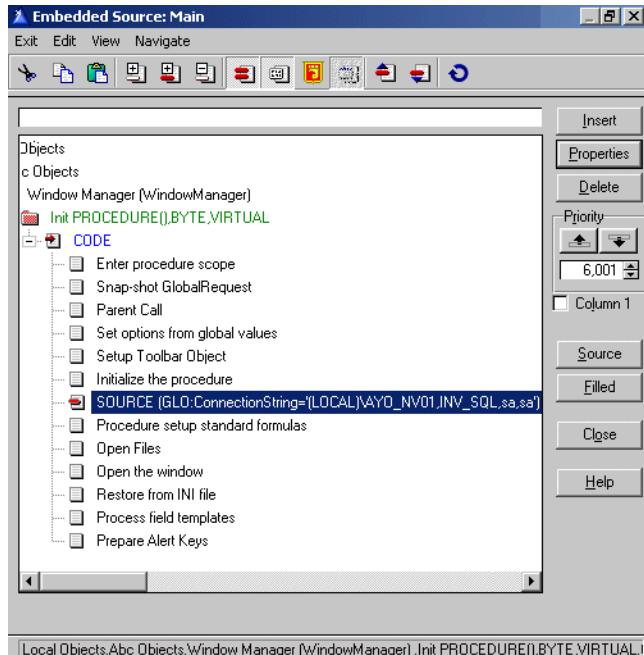
```
'AYODAHUNSI\AYO_NV01, INV_SQL, sa, sa'
```

From way the username and password is passed with the connection string, you can see that there are obvious security issues. For now, I will stick with this approach.

Migrating The Inventory Application To SQL Server

Create a global variable called GLO:ConnectionString to store the login information. Make it a CSTRING (128). Assign a value to this variable before the **Open Files** embed in the Main frame, as in Figure 5.

Figure 5: Embed point for connection string



This connection string is hard-coded in this example, but in a real-life situation, you can read this from an INI file, or, use a more secure means of building the connection string.

Resources	
Professional SQL Server 7	Robert Viera
SQL Server 2000 Developers Guide	Micheal Otey, Paul Conte
The Guru's Guide to Transact-SQL	Ken Henderson
SQL Server 7.0 Administrator's Pocket Consultant	
SQL Team	http://www.sqlteam.com
Microsoft SQL Server Home page	http://www.microsoft.com/sql/default.asp
Help, articles, and scripts for SQL Server	http://www.swynk.com/sql/

MS SQL

Interactive Product Guide	http://www.winntmag.com/Techware/InteractiveProduct/SQL2000/
SQL Server Magazine	http://www.sqlmag.com

You've learned how to set table properties, create identity fields, install the auto-incrementing code, and create a connection string. Now it's time to create the database and generate the table definitions.

SQL scripts

SQL Scripts are text statements that you execute on the back end to perform certain functions. These include creation of databases, tables, stored procedures; execution of scheduled tasks, and so many other operations that are normally done manually.

Clarion Enterprise Edition, version 5 and later, comes with a tool called the Synchronizer. This tool generates an SQL script from your Clarion dictionary, and stores that script in a text file. You then have to run this script on your back end in order to generate the database that matches your dictionary.

If you don't have Clarion Enterprise Edition, the alternative is to use third party, solutions most of which are free. Here are some programs you can use to generate SQL scripts:

- DumbDict (Developed by Tom Ruby). Can be downloaded at:
<http://www.tomruby.com/dumbdict.zip>
- Geoff Bomford's Templates. Can be downloaded at:
<http://www.comformmark.com.au/gwbsql.zip>

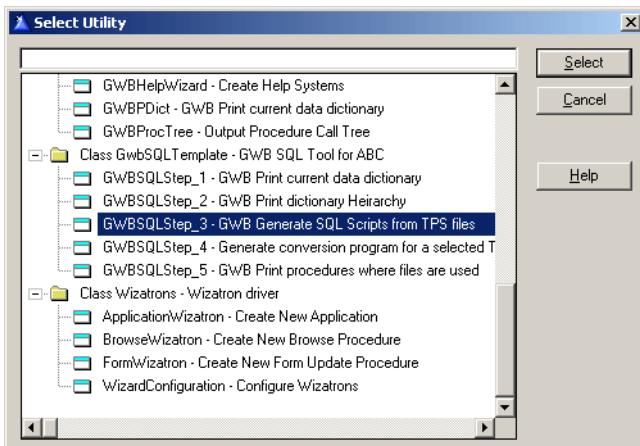
DumbDict generates SQL Anywhere code, so the script needs some modification before you can run it on SQL Server. However, Dumbdict comes with the APP file so you can make modifications in the source. On the other hand, Geoff Bomford's templates generates pure SQL Server code as well as Pervasive.SQL code; this utility is indispensable if you do not have the Clarion Synchronizer.

Generating scripts with Geoff's templates

After you've downloaded and installed Geoff's templates, open up the inventory application and choose **Application|Template Utility** from the main menu. Select

GWBSQL-Step_3 (see Figure 6) to generate the SQL Script. Remember, you need to create a sub-directory called `scripts` under the directory that contains the template.

Figure 6: Generating scripts with the utility template



Generating scripts with the clarion synchronizer

Synchronization in Clarion Enterprise terms means ensuring that the structure of the Clarion dictionary matches the database structure at the back end. This implies that if you make a change at the back end, the synchronizer can modify the dictionary to match back end, or vice-versa (referred to as two-way synchronization). Though this is a desirable feature, it does not work well all the time, so what I recommend is one-way synchronization, i.e. from the back end to the Clarion dictionary. More about that later.

Creating a blank database

Before synchronizing the Clarion Dictionary, you have to create a blank database. You can do this by running a script or SQL Statement in the Query Analyzer, or by using SQL Server's command line utilities like OSQL.EXE, ISQL.EXE.

The command line utilities are the only way to run scripts if you are deploying Microsoft SQL Server Desktop Engine (MSDE). MSDE is a trimmed down version of SQL Server optimized for not more than five users, and does not come with any of the tools mentioned under "Tools of the trade," p. 332.

MS SQL

The ISQL utility is an old command line utility based on SQL Server's DB-Library API. This has been replaced by OSQL which is based on the ODBC API.

To create a database from the command line, enter the following command:

```
OSQL -SAYODAHUNSI\AYO_NV01 -Usa -Psa  
-Q"CREATE DATABASE INV_SQL"
```

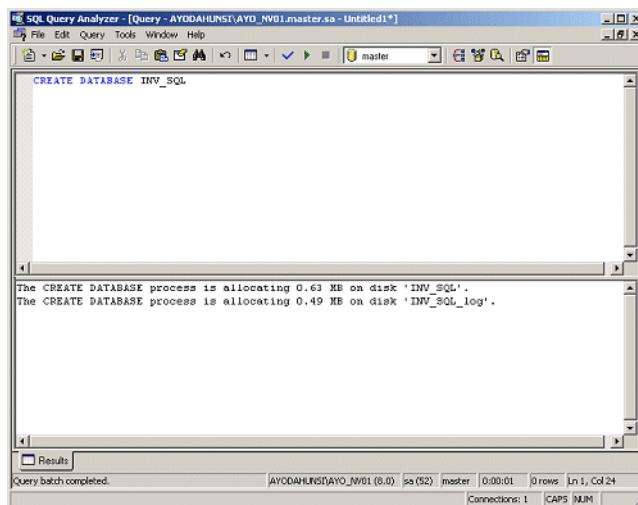
OSQL will respond with messages like the following:

```
The CREATE DATABASE process is allocating  
0.63 MB on disk 'INV_SQL'.  
The CREATE DATABASE process is allocating  
0.49 MB on disk 'INV_SQL_log'.
```

To get a list of more options available using OSQL, type OSQL /?.

If you want to create the database using the Query Analyzer, the statement to run is: CREATE DATABASE INV_SQL, as shown in Figure 7.

Figure 7: Creating the inventory database using Query Analyzer

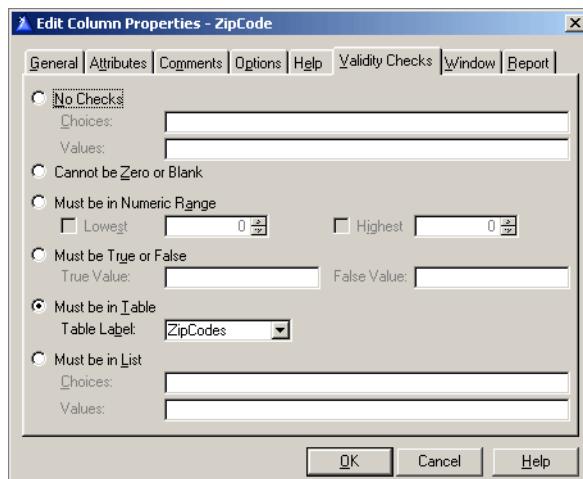


Different settings can be specified when creating a database for the first time. As you get more familiar with SQL Server, and you gain more experience, you can explore these options. For now, let the default values take effect.

RI in Clarion dictionary fields

It is common for some people to attempt enforcing some level of Referential Integrity in fields using the **Must be in Table** option under the **Validity Checks** tab. The validity check settings in the VENDOR table is a classic example (see Figure 8). If you leave these settings unchanged, you will not be able to use the Clarion Synchronizer to create your SQL script. You must remove any **Must be in Table** requirements from your table field edits. This validity check will also cause a GPF when you run the Clarion Synchronizer

Figure 8: Clarion RI enforcement in the VENDOR table



Default value in fields

You also have to be careful about how you set default values in your Clarion dictionary. For example, when you set defaults based on other fields, the Synchronizer will generate your SQL Script with these defaults, and SQL Server does not understand this. For example, the Default value for the Cost field in InvHist is PRO:COST, and the column is defined this way in the SQL script:

```
"Cost" DECIMAL(7,2), DEFAULT(PRO:COST),
```

Unfortunately, MS SQL does not know what PRO:COST means, so when you try to run a script created by the synchronizer you get a syntax error like this:

```
Line 46: Incorrect syntax near 'PRO:'.
```

When you use the Clarion Synchronizer to create your script, it is smart enough to use the equivalent SQL function for the Clarion TODAY() function as a default in the INVHIST table. The function is GETDATE(), and the definition for the Date column becomes this:

```
"Date" INT NOT NULL DEFAULT (getdate()),
```

But there is another caveat here. You can see that the data type used is INT, which is equivalent to LONG in Clarion! I'll cover this in more detail later. A reminder: do not use LONG for date types; use DATE instead.

Script generation

Assuming you've followed all suggested changes, you can now run the synchronizer and have it generate a script.

Tools of the trade

There are a couple of tools that come with SQL Server that have had a great impact in the way people work with SQL Databases. These tools make it quite easy maintain, test, and even deploy applications using SQL Server while the developer is still in the process of acquiring SQL skills. Though there are a lot of books available that describe how to effectively use of these tools, you can quickly find your way because of the intuitive interfaces.

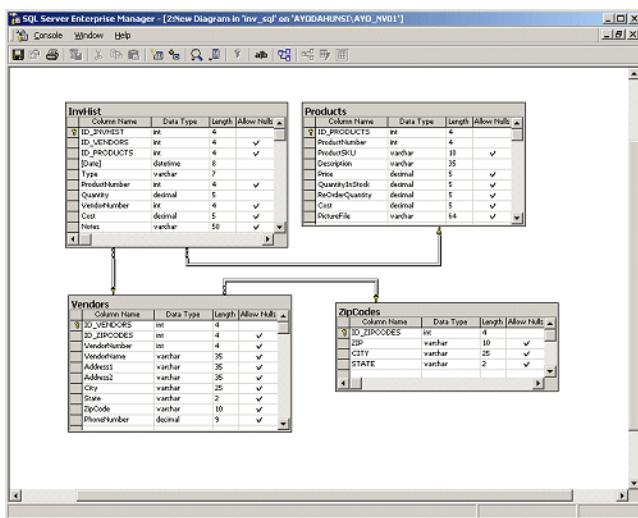
- Enterprise Manager (EM) - As the name indicates, this tool manages everything from databases to stored procedures, Alerts, Jobs, replication, etc.
- Query Analyzer (QA) - The Query Analyzer is used to run scripts or SQL commands. Commands can also be parsed (i.e. checked for accuracy) before they are executed. Press **F5** to run either the highlighted query, or every command in the QA window. Press **Ctrl-F5** to parse the highlighted query, or every command in the QA window. Almost all the tasks performed by pointing and clicking within the Enterprise Manager can be done within the Query Analyzer. The developer new to SQL Server should stick to the Enterprise Manager in order to get familiar with SQL.
- View Designer (VD) - The View Designer is seldom known to new users of SQL Server, and as a result these users may spend too much time figuring out how a complex SELECT statement should be written. When I am creating SELECT statements whereby I join many tables, I usually use the

Migrating The Inventory Application To SQL Server

View Designer to graphically build up the kind of join I want. Now, if your identity fields were defined as I suggested earlier, once you populate the View Designer the correct links/joins to your database are automatically created and scripted.

- Database Designer - The database designer shows you how tables are related/linked in your database. This tool can also be used in creating new tables, and modifying the properties of existing ones. Note that the relations in the tables (as shown in Figure 9) are not generated automatically when you synchronize with a Clarion Dictionary; you will have to add these yourself.

Figure 9: The Database Designer



However, if you followed the suggestions in creating ID fields and their corresponding keys as described under Step 1 - Table Changes, the links are generated automatically based on fields with similar names.

- DTS (Data Transformation Services) Designer - This is probably one of the most effective tools in SQL Server for data conversion, and beats anything I've seen on other database back ends. The DTS Designer can be used for converting data from one file format to another with a high degree of automation and sophistication. As an example, think of a scenario where you have to download customer invoices from an AS/400 mini computer with some massaging of the data required before it can be used; you can do this

with DTS. Or perhaps you want to do a parallel run between an old system and a new one, and data is still being entered through the old system; you can set up DTS to synchronize the two systems.

- Books Online (BOL) - This is the SQL Server help system. It is quite versatile and indispensable, especially for a starter.

I have showed how default values can be made an integral part of your tables at creation. You can also add default values later. Used carefully, defaults can help standardize constants especially when maintaining the bridge between Clarion and MS SQL.

Date discrepancies

A common example is the use of defaults in preventing the “Record has been changed by another station” error that occurs even on a single machine sometimes, and has been frustrating to a lot of users that have posted questions on the SoftVelocity newsgroups.

In the earlier discussion about default values I mentioned that even though the Clarion Synchronizer is smart enough to convert the `TODAY()` function to its SQL equivalent `GETDATE()`, this causes problems later. This is what happens:

There are two Date data types in SQL Server, namely `SMALLDATETIME` and `DATETIME`. `SMALLDATETIME` has accuracy up to one minute. e.g. 2001-05-03 03:57:00.000, while `DATETIME` has accuracy up to 3.33 milliseconds, e.g. 2001-05-03 03:57:26.480.

The problem with a `GETDATE()` default value is that if for any reason rows are added to your database outside Clarion, for example from a stored procedure, the column will be updated with a value of a higher precision (as indicated above) because `GETDATE()` returns the current date and time as a `DATETIME` data type. So, later, when a user tries to edit that row from a Clarion Update Form, she gets this message:

“This record was changed by another station. Those changes will now be displayed. Use the Ditto button or Ctrl+ to recall your changes”

The error occurs because Clarion is unable to handle the precision of milliseconds. In order to resolve this, you can change the data type on the back end from `DATETIME` to `SMALLDATETIME` if you do no need to deal with seconds. In most cases, this is unacceptable. The other option is to create a kind of global default in SQL Server that controls the precision of date values whenever updated . Fortunately, it is possible to populate a database with default values by the back end, no matter where the update is coming from, i.e. from within Clarion, or from a stored procedure. To do this, I will be making use of the following Transact-SQL functions:

Migrating The Inventory Application To SQL Server

DATEPART () - Return an integer value which represents the part used.

CAST () - Convert data from one data type to another.

CONVERT () - Similar to CAST () except with additional date formatting.

You can find a detailed explanation of these functions in the SQL Server's Books online.

What I want to do is to split the value returned by GETDATE () into parts, and rebuild the date, but this time with zeroes as milliseconds. Any default values will then be Clarion-compatible. Here's the code:

```
SELECT CONVERT(datetime,
(
CAST(DATEPART(yyyy, GETDATE()) AS varchar) + '-' +
CAST(DATEPART(mm, GETDATE()) AS varchar) + '-' +
CAST(DATEPART(dd, GETDATE()) AS varchar) + ' ' +
CAST(DATEPART(hh, GETDATE()) AS varchar) + ':' +
CAST(DATEPART(mi, GETDATE()) AS varchar) + ':' +
CAST(DATEPART(ss, GETDATE()) AS varchar)
),120)
```

Notice that I put a SELECT before the actual conversion. Get used to this; it's a commonly used way to get the result of expressions in SQL.

Now, copy this and run in the Query Analyzer, and you will get something like:

2001-05-03 03:57:26.000

depending on your time.

The Transact-SQL script to create a user defined Default called TODAY that I can use globally to populate any column is this:

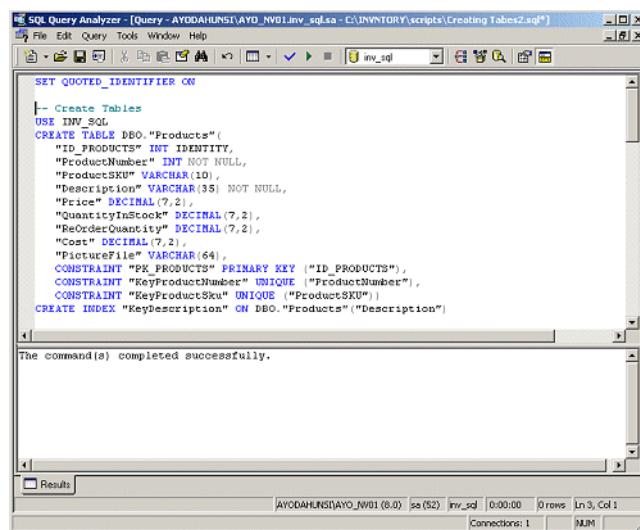
```
CREATE DEFAULT [TODAY] AS
CONVERT(datetime,
(
CAST(DATEPART(yyyy, GETDATE()) AS varchar) + '-' +
CAST(DATEPART(mm, GETDATE()) AS varchar) + '-' +
CAST(DATEPART(dd, GETDATE()) AS varchar) + ' ' +
CAST(DATEPART(hh, GETDATE()) AS varchar) + ':' +
CAST(DATEPART(mi, GETDATE()) AS varchar) + ':' +
CAST(DATEPART(ss, GETDATE()) AS varchar)
),120)
GO
```

Cut, paste, and execute this snippet in your Query analyzer, and your default will be created. Note that the default can also be created from the Enterprise manager. However, it is better you use a script to create this since it is more difficult to enter a formula in a one line prompt. Later, you will see how to use this DEFAULT in your tables.

Running scripts

The synchronizer generates the SQL table creation script from the definitions in your dictionary. You should load this script into your Query Analyzer. As you do this, you have to remember to change your active database from “master” to INV_SQL or else the tables will be created inside the master database. You really don’t want to do that.

Figure 10: Running script to create tables



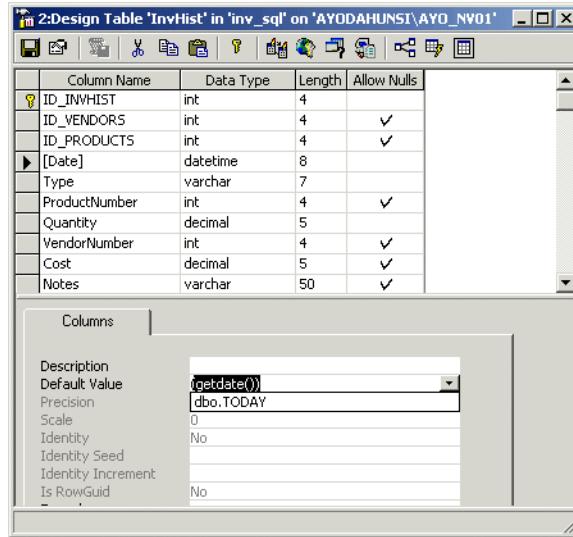
The screenshot shows the Microsoft SQL Server Management Studio (Query Analyzer) interface. The title bar reads "SQL Query Analyzer - [Query - AYODAHUNSI\AYO_NW01.inv_sql - C:\INVVENTORY\scripts\Creating Tables2.sql*]". The main window displays a T-SQL script for creating a table named "Products". The script includes columns for ProductID (INT IDENTITY), ProductNumber (INT NOT NULL), ProductSKU (VARCHAR(10)), Description (VARCHAR(35) NOT NULL), Price (DECIMAL(7,2)), QuantityInStock (DECIMAL(7,2)), ReOrderQuantity (DECIMAL(7,2)), Cost (DECIMAL(7,2)), and Picturefile (VARCHAR(64)). It also includes constraints: a primary key on ProductID, unique constraints on ProductNumber and ProductSKU, and an index on Description. Below the script, a message box states "The command(s) completed successfully." The status bar at the bottom shows the connection details: "AYODAHUNSI\AYO_NW01 (8.0) sa (S2) inv_sql 0:00:00 0 rows ln 3, Col 1".

```
SET QUOTED_IDENTIFIER ON
-- Create Tables
USE INV_SQL
CREATE TABLE DBO."Products" (
    "ID_PRODUCTS" INT IDENTITY,
    "ProductNumber" INT NOT NULL,
    "ProductSKU" VARCHAR(10),
    "Description" VARCHAR(35) NOT NULL,
    "Price" DECIMAL(7,2),
    "QuantityInStock" DECIMAL(7,2),
    "ReOrderQuantity" DECIMAL(7,2),
    "Cost" DECIMAL(7,2),
    "Picturefile" VARCHAR(64),
    CONSTRAINT "PK_PRODUCTS" PRIMARY KEY ("ID_PRODUCTS"),
    CONSTRAINT "KeyProductNumber" UNIQUE ("ProductNumber"),
    CONSTRAINT "KeyProductSKU" UNIQUE ("ProductSKU")
)
CREATE INDEX "KeyDescription" ON DBO."Products"("Description")
```

Now you can run the script. After the script has generated the necessary tables and indexes, constraints, etc., you need to change the default value of GETDATE () to dbo.TODAY

using the global default called TODAY you just created. This will help prevent precision errors discussed earlier.

Figure 11: Setting Date defaults as dbo.TODAY



You get to the window shown in Figure 11 by right-clicking on the table and selecting Design Table from the context menu.

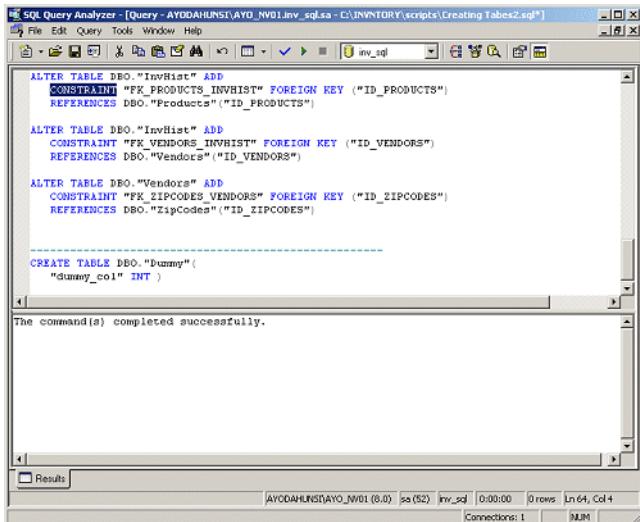
Data conversion and the role of constraints

If you take a look at the script created with the synchronizer, you will see some CONSTRAINT statements towards the end of the script. This book contains a number of references to constraints and referential integrity, which you can find in the index. Nevertheless, as a recap, a primary key constraint will ensure that all rows (records) in a

MS SQL

table will have a unique key, and a foreign key constraint will ensure that a relationship exists between two tables.

Figure 12: Constraint statements in SQL script



The screenshot shows the SQL Server Management Studio (Query Analyzer) interface. The query window displays three SQL statements:

```
ALTER TABLE DBO."InvHist" ADD  
CONSTRAINT "FK_PRODUCTS_INVHIST" FOREIGN KEY ("ID_PRODUCTS")  
REFERENCES DBO."Products"("ID_PRODUCTS")  
  
ALTER TABLE DBO."InvHist" ADD  
CONSTRAINT "FK_VENDORS_INVHIST" FOREIGN KEY ("ID_VENDORS")  
REFERENCES DBO."Vendors"("ID_VENDORS")  
  
ALTER TABLE DBO."Vendors" ADD  
CONSTRAINT "FK_ZIPCODES_VENDORS" FOREIGN KEY ("ID_ZIPCODES")  
REFERENCES DBO."ZipCodes"("ID_ZIPCODES")  
  
CREATE TABLE DBO."Dummy"  
"dummy_col" INT  
-----  
The command(s) completed successfully.
```

The status bar at the bottom indicates the session is 'AYODAHUNSI\AYO_NV01 (8.0)' with 'sa (52)' as the user, and the command took '0:00:00' to execute. It also shows there are 0 rows and 64 columns.

For example, you cannot have a record inserted into the VENDORS table without an ID from the Zip Code table. Now this presents a problem considering the way Clarion inserts a blank record when it tries to auto-increment; when you try to insert a record into the VENDORS table, Clarion will actually try to add a blank record first. Unfortunately, when foreign key constraints are used, SQL Server will not allow this since the column - ID_ZIPCODES in the VENDORS table has not yet been filled with any value.

According to the Clarion help on Relationship Properties in the Dictionary, setting the Referential Integrity Constraints to **Restrict**, **Restrict (Server)**, **Clear (Server)** are supposed to generate different kinds of code within the SQL script. However, from what I have seen so far, whatever setting you use will still generate these Foreign Key constraints.

For now, you can remove these foreign constraints. You can do this through the Enterprise Manager or by running the code script show below:

```
ALTER TABLE DBO."InvHist"  
DROP CONSTRAINT "FK_PRODUCTS_INVHIST"  
ALTER TABLE DBO."InvHist"  
DROP CONSTRAINT "FK_VENDORS_INVHIST"  
ALTER TABLE DBO."Vendors"  
DROP CONSTRAINT "FK_ZIPCODES_VENDORS"
```

The application

Remember that you deleted all the procedures in the application; you now have to import the procedures again from the original example application - INVNTORY.APP. When you do this Clarion will bring up an error about the dictionaries being different. This is no cause for concern since you didn't change any fields, rather, you only added some new ones. Make sure you import all the procedures except the MainFrame because you have already added the assignment to the connection string in a MainFrame embed point.

The application should compile fine without any errors. However, If you attempt to run the converted application, you are likely to get this error message:

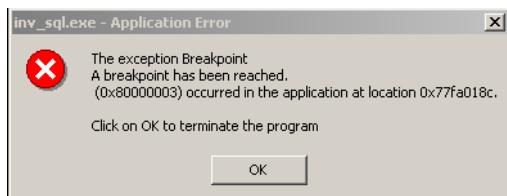
File (DBO.Vendors) could not be opened. Error: ODBC.DLL Could Not Be Loaded (1). Press OK to end this application.

This happens when you compile your EXE as a 16 bit application, which is the default mode for the shipped Inventory Application in versions prior to 5.5. Change your application to 32 bit and this is resolved.

When working with SQL-based systems, you need to start thinking of the files (tables), stored procedures, and views as one single unit called the database. So, it is better to open all the files (tables) at the start of the application, and close them when exiting. Of course, in applications that have a lot of tables, this can take a long time, but once the tables are opened, whatever procedure needs to use these tables will not have to reopen them again. Note also that an attempt to open the first table establishes a connection (using your connection string) with the database.

After you have made the changes to the example application, running the application and exiting will give you the GPF shown in Figure 13:

Figure 13: A breakpoint exception exiting the SQL application



You can resolve this GPF by adding the INVHIST, PRODUCTS, VENDORS, and ZIPCODES tables to the **Other Tables** section of the File **Schematic** in the MainFrame procedure. By doing this, the tables are opened as the MainFrame is opened and closed properly when the frame closes.

Which way to go

Having gone through the basic conversion of a Topspeed file system to a SQL Server system, you need to ask yourself if you want to stop here and deploy the application to your customers in the present state. Or do you want to move out the underlying business logic to the server as stored procedures and database constraints? For instance, it would be nice to have a trigger that generates an email message when the stock quantity of any product falls below the re-order level.

As I said before, how much code to move to the server is a business decision. One thing is certain: SQL systems are rapidly replacing flat-file systems, and programming languages/interfaces are increasingly transferring business logic to the back end.

Source code

See “Appendix A: Getting Support,” p. 601, for information on how to get the source accompanying this book.

- v3n8sql.zip

USING SQL SERVER'S DATA TRANSFORMATION SERVICES

by Ayo Ogundahunsi

In “Migrating The Inventory Application To SQL Server,” p. 317, I demonstrated how to convert the Inventory application from TopSpeed file format to a more powerful database engine - Microsoft SQL Server. In converting an application, changing database drivers is not the only requirement. You also have to provide a way to convert existing data. In most cases, existing clients are probably running applications using the TopSpeed database driver, and transition to the upgrade should be easy and accurate.

One way to convert data is to use Microsoft’s Data Transformation Services (DTS). As the name indicates, DTS transforms data from one form to the other. DTS is a very powerful and effective tool, and comes bundled with SQL Server versions 7 and 2000. In this chapter I will demonstrate how to use DTS to move data from the Inventory example to the converted SQL Server application.

Using DTS

You can use DTS to move data from Data/ODBC Sources to SQL Server and vice versa. For example, you can move data from an Oracle Database to a Sybase Database without going through SQL Server at all.

One of the strengths of DTS is its ability to assemble tasks or functions and connections to heterogeneous systems, and synchronize all these together into what is called a package. The tasks can be importing or exporting data, or sending an email as soon as a particular task is completed. The actions, processes, and settings created for transforming data can be saved in SQL Server as a package, or externally as a Visual Basic Script (VB Script) that can be run from a Visual Basic Application.

Improving the database

It's quite simple to set up DTS to transform the data contained in the four files of the Inventory application (INVHIST.TPS, PRODUCTS.TPS, VENDORS.TPS, and ZIPCODES.TPS). As usual, I will be using the terms "files" and "tables" interchangeably, however I will specifically use "files" when I am talking about TopSpeed data, and "tables" when I am referring to SQL Server data. I will also introduce some additional tables that will make the Inventory Application's database design more practical.

The VENDORS file contains City, State and ZipCode fields. It also contains a field called ID_ZIPCODES that is to be the linking field with the ZIPCODES files. To follow normalization rules, and achieve better performance, I will remove the City, State, and ZipCode fields from the VENDORS files. I'll also create separate City and State tables to replace the fields in Vendors, but this data will also be linked to Vendors via ZipCode. While this is a good idea, understand that if you have to enter data into another table, say a Customer table you must know the zip code for the customer or you will not be able to add the record.

The interrelation will now be:

```
STATE <->> CITY <->> ZIPCODES <->> VENDORS
```

Every zip code belongs to some city, and each city is in a state, so as long as you have the zip code stored in the Vendor table you can easily get the rest of the information. The idea is to try to eliminate redundant or repeated data as much as possible. This is called Data Normalization. For more information please read Tom Ruby's five chapters on the subject, beginning with "Managing Complexity, Rule 1: Eliminate Repeating Fields," p. 21.

Updating the ZipCodes file

The ZIPCODES file in the original Inventory application contains about 3,924 zip codes. This is not even half of the zip codes in the United States, but it is possible to add data from an external table with enough zip codes that gives a realistic figure. You can download a more accurate zip code file (<http://www.par2.com/getit/zipcodes.zip>) from Steve Parker's web site. This file contains about 48,254 records.

A little later I'll show how to import a text file containing the 50 US states as well as their abbreviations into a SQL Server database, with the CITY table filled accordingly, and the linking IDs appropriately inserted. For now, I am going to do a straight import of the four files directly into the SQL Server back end.

Connecting to data sources

When connecting to a Data source, what readily comes to my mind is Open Data Base Connectivity (ODBC). I can use ODBC with DTS, but I can also connect to a data source by using OLE DB Drivers.

Microsoft made some improvements to ODBC by creating what is known as OLE DB which is a way of providing data access at a component level with an interface that is not restricted to relational data only (as implemented in ODBC), but any other kind of non-structured data that can include spreadsheets, email, etc. The driver for OLE DB is known as an OLE DB Provider.

Microsoft ships an OLE DB Provider (Microsoft OLE DB Provider for ODBC) with SQL Server.

See the following sites for more information on ODBC and OLE DB:

- <http://www.microsoft.com/data/oledb/default.htm>
- <http://www.oledb.com/ole-db/guide.html>

Data conversion options

There are two popular ways to convert data from the TopSpeed files to SQL Server tables. One is by using Data Transformation Services (DTS), and the other is by adding TopSpeed Data Source as a Linked Server to the SQL Server Environment using OLE DB. With a linked server, you can send SQL to the database on the linked server and it will behave as if it were an SQL Server database. However, the response to SQL requests is not as fast as if

an SQL Server database is being accessed directly. In this chapter, I will limit my explanation to DTS.

TopSpeed ODBC Driver

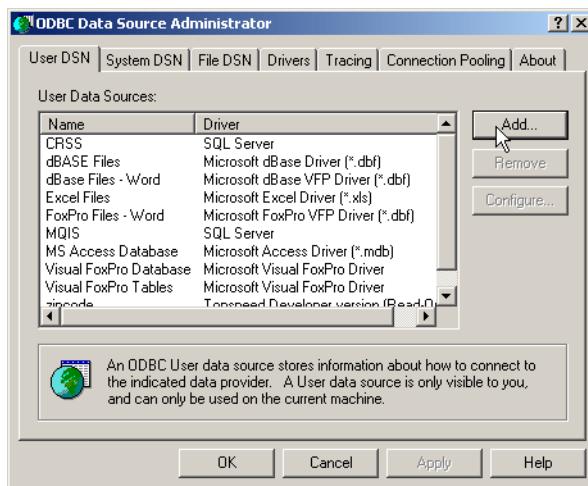
When you install the Enterprise Edition of Clarion, you automatically install the TopSpeed ODBC Driver (developer version) as well. The developer version of the driver *cannot* be deployed with any application; if you want your customers to use it you need to contact Soft Velocity Sales for ODBC license requirements.

Nevertheless, the developer version is appropriate for the purposes of this chapter. Since this is an ODBC driver, the first step is to create a Data Source Name (DSN).

WARNING: It is important to mention here that any attempt to do anything too complicated with this TopSpeed ODBC Driver freezes my system. For example when I tried to create a linked server to TopSpeed, the process froze my SQL Server, and nothing worked until I rebooted the PC. Probably this is a security feature by SoftVelocity, but it is strange that I was unable to run my Enterprise Manager after this. Shutting down the SQL Server service and restarting it did not make any difference either. I had to restart my PC.

You can add a DSN by using the ODBC Data Source Administrator as shown in Figure 1.

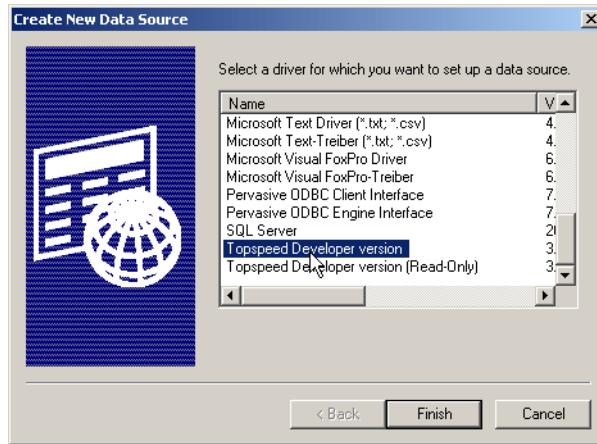
Figure 1: Adding a DSN, step 1



Using SQL Server's Data Transformation Services

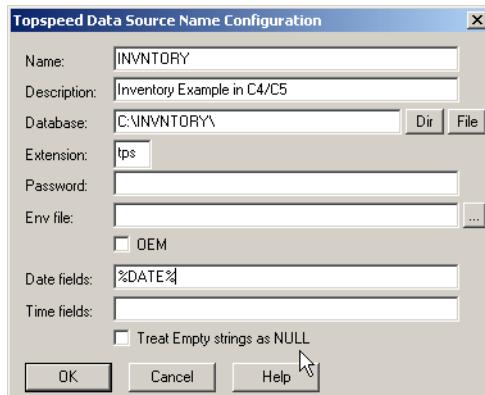
Two kinds of the TopSpeed ODBC Driver are usually installed. One is Read-Only. Either will work for this purpose (See Figure 2).

Figure 2: Adding a DSN, step 2



You then specify the physical location of the TPS data file, as shown in Figure 3.

Figure 3: Adding a DSN, step 3



In Figure 3, the %DATE% shown in the Date Fields allows you to automatically convert a Clarion Date field which is defined as a LONG data type to an ODBC date data type. The same applies to TIME fields.

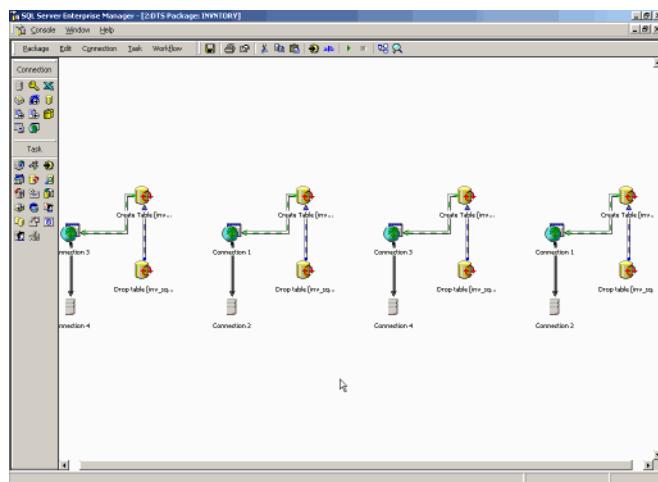
Understanding DTS

A DTS package contains all the rules required to automate the process of transforming data from one storage format to another. Note that I didn't say "from one SQL back end to another." You can use DTS to transform data from an Excel spreadsheet to SQL Server or another back end (e.g. Oracle, Sybase, DB2, etc.), or, to a flat file system like TopSpeed, dBase, or an ASCII file, and vice versa.

In order to transform data, you need a data connection to the source where data is coming from as well as the destination where the data is to be converted. Default connections are available for Microsoft Access, Excel, Paradox, Oracle (using an ODBC driver for Oracle installed with SQL Server), Text Files, and HTML. Also available is connection to any ODBC Driver as well as the OLE DB Provider for SQL Server.

DTS works in the form of a process-flow, and you create this visually. To build this process-flow diagram, you drop tasks into the designer and connect them together, as shown in Figure 4.

Figure 4: The DTS Designer



Different tasks can be linked together as a chain of events to be executed one after the other. Some of these tasks are:

- Connecting to an FTP site and transferring files
- Automatically sending an email once a task has been completed
- Running a Microsoft Message Queue Task
(<http://www.microsoft.com/msmq/default.htm>)

- Calling a stored procedure,
- Executing an SQL task,
- Copying a database

A very useful white paper on DTS is available at the MDSN web site:
http://msdn.microsoft.com/library/default.asp?URL=/library/techart/dts_overview.htm.

Another site dedicated solely to DTS is <http://www.sqldts.com>

Creating a package

Within the Microsoft SQL Server Program menu, there is a sub-menu - “Import and Export Data”. This brings up an easy-to-understand wizard that takes you through the process of setting up the package.

In setting up the package, for the Data Source select “TopSpeed Developer Version,” and the DSN you just created (see Figure 5).

Figure 5: Choosing a Data Source



After selecting a Data Source Name (DSN), proceed by clicking the **Next** button. You do not need to fill the other fields like **Username** and **Password** since you didn't fill these while creating the data source.

MS SQL

In configuring the destination, select the “Microsoft OLE DB Provider for SQL Server”. Remember not to choose an ODBC connection as this requires creating a DSN entry for SQL Server, which is unnecessary since the OLE DB driver is adequate, and more efficient.

Figure 6: Choosing a Destination



In choosing a destination, you have to fill in the **Username** and **Password** fields. The database to be selected is the one I described earlier.

The next form on the wizard (see Figure 7) shows how to do a straight copy by selecting the default radio button – **Copy table(s) and view(s) from the source database** – or by

Using SQL Server's Data Transformation Services

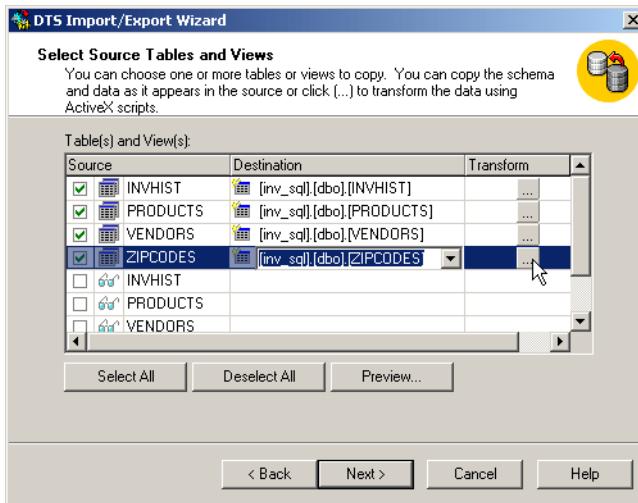
writing a SELECT statement to retrieve a specific record set to be transformed – Use a query to specify the data to transfer.

Figure 7: Specifying Data selection method



The next step is to map fields in the TopSpeed files to tables in SQL Server, as in Figure 8.

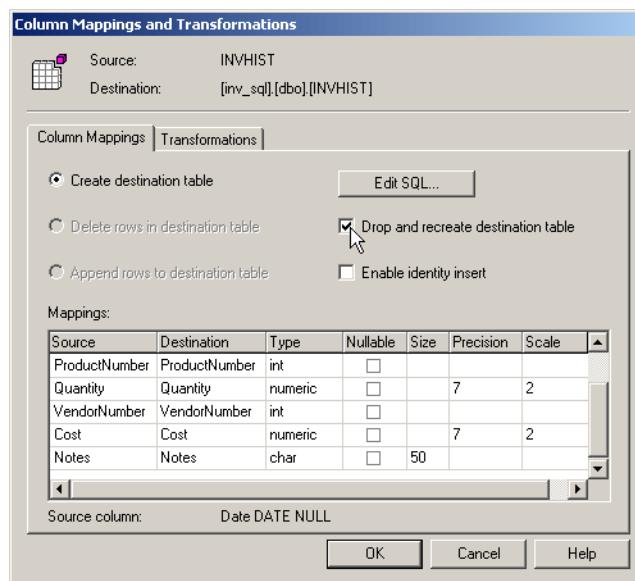
Figure 8: Mapping tables



MS SQL

When you click on the ellipsis button in the **Transform** column as shown in Figure 8, you can specify how the destination table is used. This means the destination table(s) can be deleted (DROP in SQL terms) and recreated before fields (called *columns* in SQL) and the records (called *rows* in SQL) are updated sequentially with data from the TopSpeed files (see Figure 9).

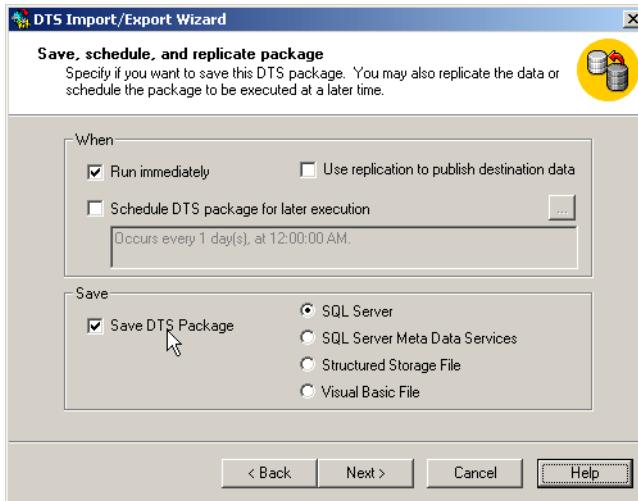
Figure 9: Pre-update Actions



Using SQL Server's Data Transformation Services

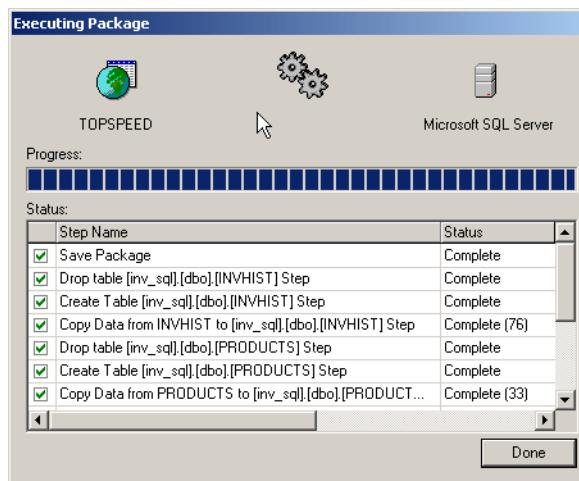
There are different ways a DTS package can be saved. The most ideal way is to save it in SQL Server, so you can run it at any time via a stored procedure, as in Figure 10.

Figure 10: Save and Run



Click on **Next** to start the transformation process. If all goes smoothly, then a screen similar to Figure 11 appears.

Figure 11: DTS Completed



MS SQL

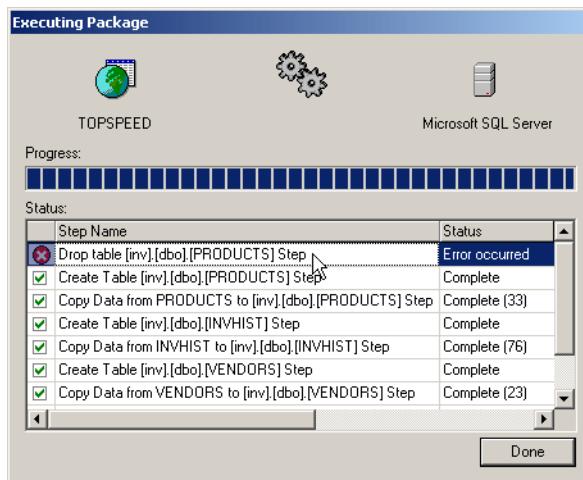
We know all does not go smoothly most of the time. When an error happens you can modify the transformation script to suit your environment. In the example the tasks executed are in this sequence:

- 1) Save Package (The package is saved in your SQL database) - This is always executed except when disk space is insufficient, or if you do not have the required database permission to save DTS packages.
- 2) Drop Table [Table Name]
- 3) Create Table [Table Name] - This task might not execute if you do not have the permission to CREATE tables, or if you have run out of disk space.
- 4) Copy Data

Note that tasks in 2, 3, and 4 are repetitive for all tables.

Whenever the execution of a task is unsuccessful, you will see a red “x” instead of the green check mark indicated in the first column. The severity of the failure is dependent on the kind of task being performed. For example, if (2) is unsuccessful but (3) and (4) succeed, this could be due to the fact that you checked **Drop and recreate destination table** as indicated in Figure 9 when the table is non-existent in the database. It could also be that you do not have the database permission to DROP tables, in which case you could end up with duplicated data if you really wanted to start with a blank table before your data is transformed. This applies to (3) as well.

Figure 12: DROP Error in DTS Package



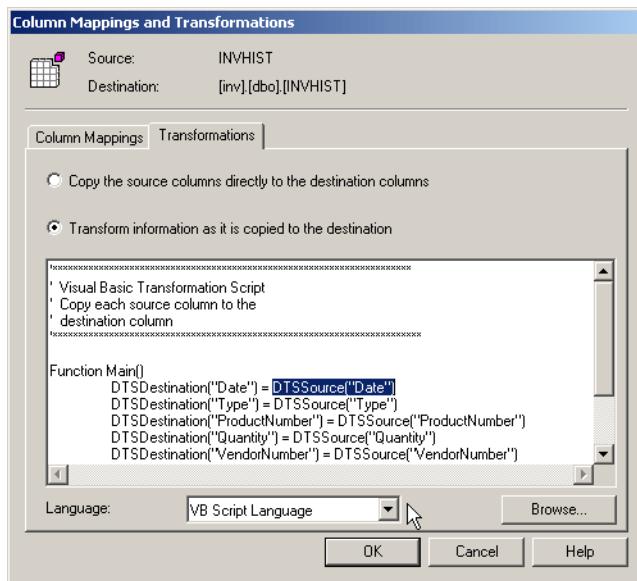
On the other hand, if (4) does not succeed, then no transformation has been done; this can happen with data type conversions. See the next section for more on how to resolve this. Another notable cause of failure can come from a column with a unique index being populated with a duplicate values.

Whenever the status of a task indicates an error, double-clicking on the task will display the reason for the error.

A VB Script example

When you are transforming data from, for example, a Btrieve file to a SQL Server table, you are likely to run into Date field conversion problems, in which case DTS will not transform the data. When this happens, you have to modify the Transformation script as shown in Figure 13. (Note that this figure is similar to Figure 9; you get to it by clicking the **Transformations** tab.). Your language of choice for editing can either be VB Script, or JavaScript.

Figure 13: Modifying column mappings, step 1



If you are converting a file with a structure where date and time information is stored in two separate fields, it makes sense to merge the fields together and update the corresponding SQL Server DateTime column.

MS SQL

For example, assume there is another time field called TIME in the INVHIST file; the line containing the selected text as shown in Figure 13 is:

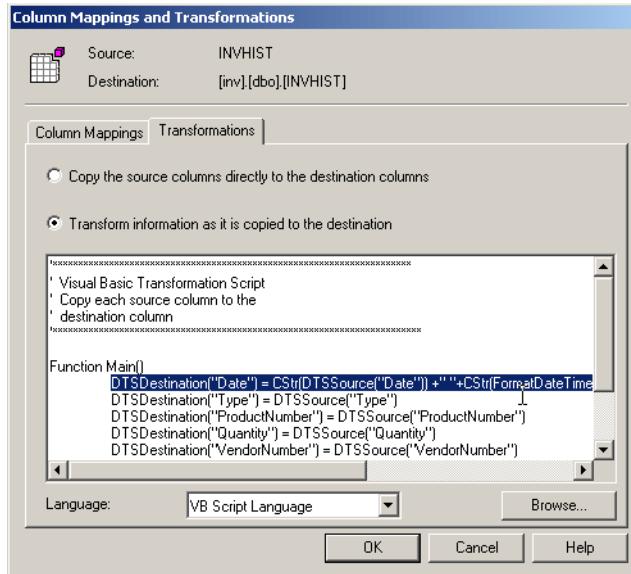
```
DTSDestination("Date") = DTSSource("Date")
```

Modifying the script, you will now have:

```
DTSDestination("Date") = CStr(DTSSource("Date")) +"  
"CStr(FormatDateTime(DTSSource("Time"), vbShortTime))
```

I've used some VB Script functions (in bold) in order to achieve a SQL DateTime field format picture. VbShortTime is a Visual Basic constant which allows you to display time using the 24-hour format (HH:MM).

Figure 14: Modifying column mappings step 2



If you are thinking of doing a lot of your work in DTS, it is a good idea to start getting familiar with VB Script or JavaScript. You can download the VB Script HTML Help file from the Microsoft Script Technologies web site:
<http://www.microsoft.com/msdownload/vbscript/scripting.asp>

Summary

Converting the Inventory application's TPS data to a SQL database is quite simple. Nevertheless, for practical purposes, say deploying an upgrade to an existing site currently

Using SQL Server's Data Transformation Services

running on TopSpeed files might require some level of automation. For Visual Basic (VB) applications, a DTS can be saved as a Visual Basic Script (VB Script). This can be compiled as part of Visual Basic. As usual, the Clarion Language does not enjoy this luxury, so there is the need to provide a Clarion Application with an automation feature that can also be integrated into the upgrade.

MS SQL

CONVERTING DATA WITH LINKED SERVERS

by Ayo Ogundahunsi

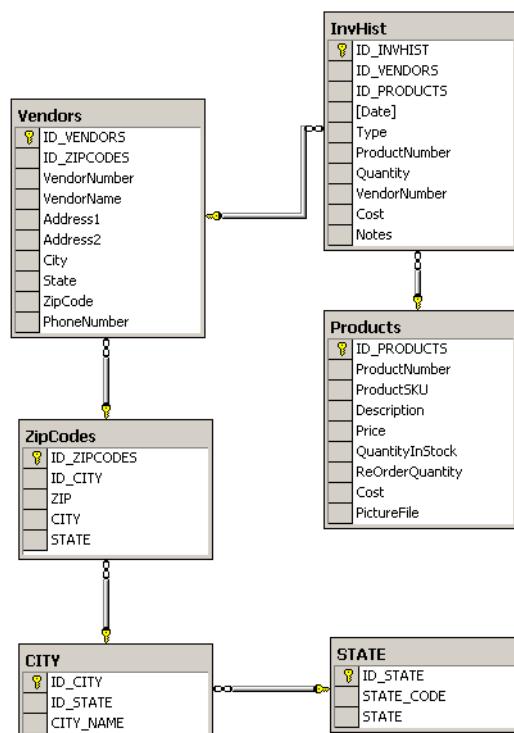
In all the examples I have presented in previous chapters, I didn't alter any of the data. However, when migrating from a flat-file system to an SQL environment, you can never have an efficient SQL application if you are only comfortable with a straight conversion, where data access and updates still follow the logic of a flat-file system. On the other hand, if you intend to make the application more SQL-like, you might have to do some data manipulation, column filling and table splitting to guarantee a more efficient system.

Now I will show you a different method of moving data into SQL Server. I will also demonstrate how the data can be manipulated and converted to a more efficient relational database model.

A new database schema

Figure 1 shows how these changes will be reflected in the database structure:

Figure 1: New Schema for Normalization (1st Stage)



Notice `dbo.CITY` and `dbo.STATE` have been added, but for now the columns `dbo.ZIPCODES.CITY` and `dbo.ZIPCODES.STATE` still exist in `dbo.ZIPCODES`. The `dbo` prefix specifies the database owner. I will be qualifying tables in SQL Server occasionally this way in order to differentiate them from Topspeed files.

When converting a flat file system to SQL, most developers start the design of the database from the Clarion Dictionary and then export to SQL. For database maintenance, most developers will make changes in Clarion and then try to use the synchronizer to push the updates to SQL Server. I prefer to make the changes in SQL Server and then use the synchronizer to push the updates to Clarion. (Actually, I make the changes in the Clarion Dictionary manually most of the time.)

You can get the script `State_City.sql` that creates the `STATE` and `CITY` tables as part of the downloadable source at the end of this chapter. It is important you run this script before you start any conversion.

Zipcodes

If you take a look at the Topspeed `ZIPCODES` file, `C:\INVNTORY\ZIPCODES.TPS`, you will notice that the `State` field data is abbreviated, for example 'FL' for Florida, as shown in Figure 2. However, I need to be able to get the full name of a state.

Figure 2: Original Zip Code file

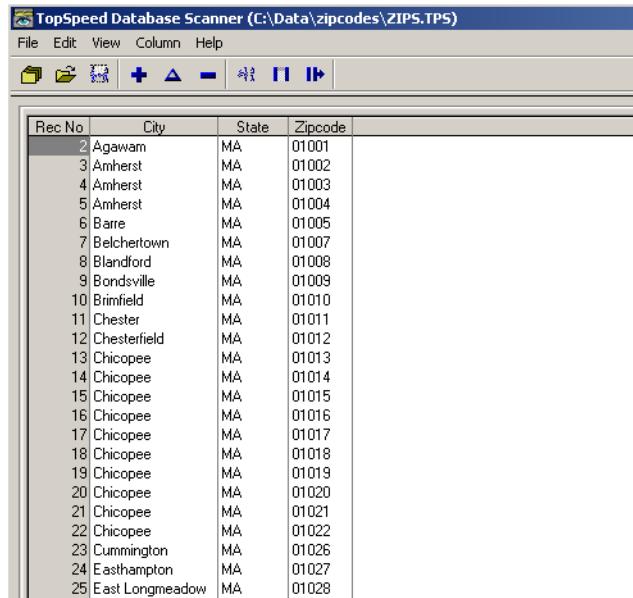
Rec No	Zip	City	State
2	32002	Astor	FL
3	32005	Barberville	FL
4	32007	Bostwick	FL
5	32008	Branford	FL
6	32009	Bryceville	FL
7	32010	Burnell	FL
8	32011	Callahan	FL
9	32012	Crescent City	FL
10	32013	Day	FL
11	32014	Daytona Beach	FL
12	32015	Daytona Beach	FL
13	32016	Daytona Beach	FL
14	32017	Daytona Beach	FL
15	32018	Daytona Beach	FL
16	32019	Daytona Beach	FL
17	32020	Daytona Beach	FL
18	32021	Daytona Beach	FL
19	32022	Daytona Beach	FL
20	32023	Daytona Beach	FL
21	32027	Daytona Beach	FL
22	32028	De Leon Springs	FL
23	32029	Daytona Beach	FL
24	32030	Middleburg	FL
25	32031	East Palatka	FL
26	32032	Edgewater	FL
27	32033	Elkton	FL

Also, this zipcodes file contains about 3,924 records, whereas you can get a file containing about 48,254 records from Steve Parker's site (<http://www.par2.com/getit/zipcodes.zip>) . In order to avoid confusion with the original zip code file shipped with Clarion, I will rename the downloaded file to `ZIPS.TPS`.

MS SQL

A close look at the contents of this downloaded file shows that it also contains only the abbreviation of states. While this is okay for something like a mailing label e.g. MA 01001, it is inadequate for proper report generation.

Figure 3: Downloaded Zip Code file



The screenshot shows a software interface titled "TopSpeed Database Scanner (C:\Data\zipcodes\ZIPS.TPS)". The menu bar includes File, Edit, View, Column, Help, and a toolbar with icons for file operations. The main window displays a table with four columns: Rec No, City, State, and Zipcode. The data consists of 25 records, all from the state of Massachusetts (MA), ranging from zip code 01001 to 01028. The table has a header row and 25 data rows.

Rec No	City	State	Zipcode
2	Agawam	MA	01001
3	Amherst	MA	01002
4	Amherst	MA	01003
5	Amherst	MA	01004
6	Barre	MA	01005
7	Belchertown	MA	01007
8	Blandford	MA	01008
9	Bondsville	MA	01009
10	Brimfield	MA	01010
11	Chester	MA	01011
12	Chesterfield	MA	01012
13	Chicopee	MA	01013
14	Chicopee	MA	01014
15	Chicopee	MA	01015
16	Chicopee	MA	01016
17	Chicopee	MA	01017
18	Chicopee	MA	01018
19	Chicopee	MA	01019
20	Chicopee	MA	01020
21	Chicopee	MA	01021
22	Chicopee	MA	01022
23	Cummington	MA	01026
24	Easthampton	MA	01027
25	East Longmeadow	MA	01028

Note that ZIPCODES.TPS also contains some non-US Postal/Zip codes, and I have to retain these when I attempt to merge ZIPCODES.TPS and ZIPS.TPS.

I have itemized a list of operations needed to reconstruct this schema:

- Merge ZIPS.TPS (48,254 records) into ZIPCODES.TPS (3,924 records) while retaining non-US zip codes.
- Where zip codes match, use City field from ZIPS.TPS.
- Auto-increment `ID_ZIPCODES` in `dbo.ZIPCODES`. This will now be the primary key of this table and SQL Server will set the value automatically on inserts, since this column was declared with an `IDENTITY` property. Whatever value is updated will have a relationship with the `VENDOR` table as explained in the next point.
- The column `ID_ZIPCODES` in `dbo.VENDORS` will be updated with the corresponding `ID_ZIPCODES` from `dbo.ZIPCODES`.

The last item does not have to be done at this part of the conversion.

Linked servers

Earlier, I briefly mentioned *linked servers*. A linked sever is virtual server created by a link from an SQL Server environment to an OLE DB data source. This link can be a relational database or columnar data in a flat file. It can even be an Excel spread sheet as you will soon see. See MSDN (http://msdn.microsoft.com/library/default.asp?url=/library/en-us/acdata/ac_8_qd_12_5vv.asp) for more details on linked servers.

Included in the source download at the end of this chapter is the Excel spreadsheet State.xls containing all the states in the US. Note that there are other ways of importing the data contained in Excel file into SQL Server. You can export the spreadsheet to a text file and subsequently import it to SQL Server by using the BULK INSERT command in Transact SQL (Transact SQL is the dialect of SQL specific to SQL Server), or, you can use the BCP command. BCP is a command line utility for importing data into a SQL Server Table, or exporting the rows from a SQL Server table to a text file. You can also use Data Transformation Services (DTS) to import the Excel file as I demonstrated in “Using SQL Server’s Data Transformation Services,” p. 341.

But the most exciting way is to make the Excel file a linked server. This allows you to manipulate the Excel file like relational data, and even create a join (a JOIN helps you to return a result set whose columns can be a combination of two or more tables) or a View that includes the Excel data. After downloading STATE.XLS, copy it to C:\INVNTORY, or whichever directory you have been using for data conversion.

The Excel spreadsheet

STATE.XLS contains the names of the 50 United States together with their two-character codes. In order for me to be able to access the cells in the Excel spread sheet, it is mandatory that I define a range of the data to be used in whatever query is required. This I have labeled US_STATES, and is from A1:B52. The first row contains the field names,

MS SQL

since any query sent from SQL Server needs to understand what column is to be returned. Note that the first row is *always* returned as the column name.

Figure 4: Excel Spreadsheet

	US STATES	= STATE	C	D	E	F
1	STATE	CODE				
2	Alabama	AL				
3	Alaska	AK				
4	Arizona	AZ				
5	Arkansas	AR				
6	California	CA				
7	Colorado	CO				
8	Connecticut	CT				
9	Delaware	DE				
10	District of Columbia	DC				
11	Florida	FL				
12	Georgia	GA				
13	Hawaii	HI				
14	Idaho	ID				
15	Illinois	IL				
16	Indiana	IN				
17	Iowa	IA				
18	Kansas	KS				
19	Kentucky	KY				
20	Louisiana	LA				

Creating the Linked Server

Creating a linked server is quite simple. You can do this from SQL Server's Enterprise Manager, or you can execute SQL to create it. I will use SQL rather than the easy way of using the Enterprise Manager. Below is the SQL code to create a linked server, add a login to the linked server, and then use SELECT to list the rows in the STATE.XLS. Copy this code to the Query Analyzer, and execute it.

```
sp_addlinkedserver N'ExcelZ', N'Jet 4.0',
    N'Microsoft.Jet.OLEDB.4.0',
    N'C:\INVNTORY\state.xls', NULL, N'Excel 5.0'
GO
sp_addlinkedsrvlogin N'ExcelZ', false, sa, N'ADMIN', NULL
GO
select * from EXCELZ...US_STATES
GO
```

In this SQL code, you will notice I preceded the string parameters with the N character. This means I am passing Unicode constants, and the parameters are interpreted within the SQL engine as Unicode data. On the other hand, I can remove the Ns and the code will still

Converting Data With Linked Servers

work fine, but I will be passing string constants which are going to be evaluated with the engine using a code page. Unicode is necessary for internalization of applications

Figure 5 shows what the output will look like. Because there are state codes, you can create a JOIN/VIEW on the codes to ZIPCODE.TPS.

Figure 5: Creating a Linked Server

The screenshot shows the SQL Query Analyzer interface. The title bar reads "SQL Query Analyzer". The menu bar includes File, Edit, Query, Tools, Window, and Help. The toolbar contains various icons for file operations. The connection dropdown shows "master". The query window title is "Query - AYODAHUNST\AYO_NW01.master.AYODAHUNST\Ayo - Untitled1*". The query itself is:

```
sp_addlinkedserver N'ExcelZ', N'Jet 4.0',
                    N'Microsoft.Jet.OLEDB.4.0',
                    N'C:\INVNTORY\state.xls', NULL, N'Excel 5.0'
GO

sp_addlinkedsrvlogin N'ExcelZ', false, sa, N'ADMIN', NULL
GO

select * from EXCELZ..US_STATES
GO
```

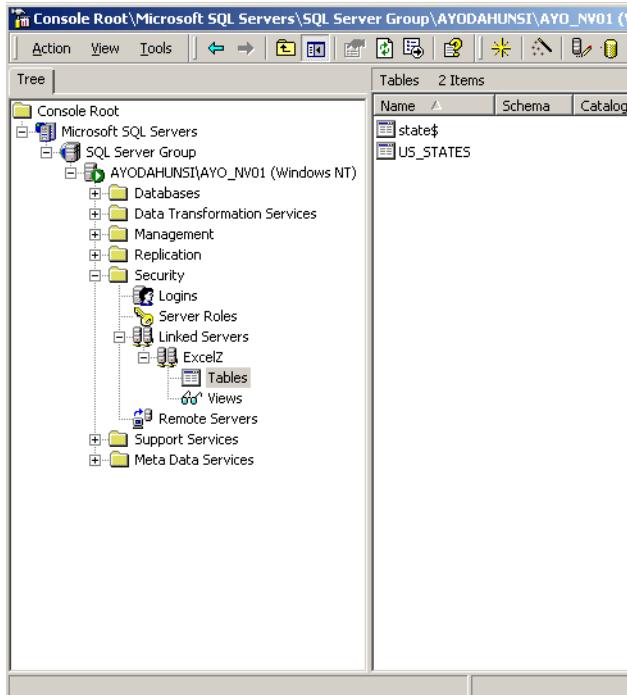
Below the query window, a results grid displays the data from the query:

	CODE	STATE
1	AL	Alabama
2	AK	Alaska
3	AZ	Arizona
4	AR	Arkansas
5	CA	California
6	CO	Colorado
7	CT	Connecticut
8	DE	Delaware
9	DC	District of Columbia
10	FL	Florida

MS SQL

After you have created the Excel Linked server, you will see it appear under the **Security** section of the SQL Server Group (See Figure 6).

Figure 6: Created Linked Server in Enterprise Manager



I have used two stored procedures, `sp_addlinkedserver`, and `sp_addlinkedserverlogin`. The first one creates the linked server, and the second one helps to ensure that your local instance of SQL Server can have access to the linked server.

Updating the STATE table

The first step in table updates for database restructuring has to do with `dbo.STATE`. Since I already have direct access to the Excel file, I can use an `INSERT` command to fill this table. What I want to do is to retrieve data from the Excel file and insert into `dbo.STATE` all in one go; this is what SQL is good at, working in a set-oriented way. The conventional (flat file) method is to loop through the data one row at a time. This supports my earlier comment about the need to restructure updates in a legacy system if you want to reap the benefits of migrating to SQL.

The script below will fill `dbo.STATE` with data from `STATE.XLS`. Remember to run the script `state_city.sql` first so that `dbo.STATE` exists.

```
INSERT dbo.STATE (STATE, STATE_CODE)
SELECT STATE, CODE FROM EXCEL...US_STATES
```

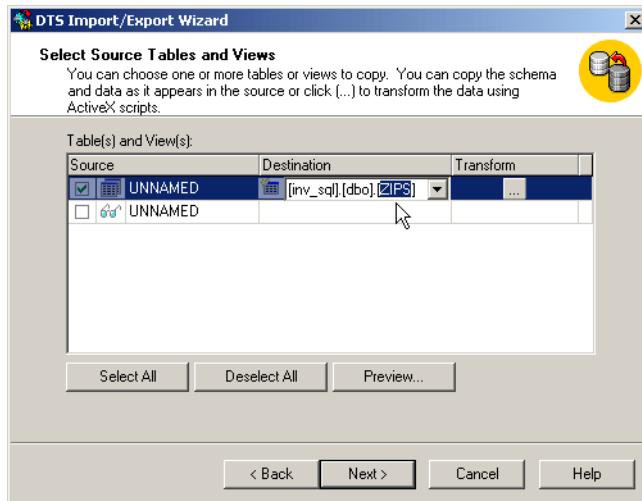
Note that the second line returns a result set in the order of the columns being inserted into the first line.

Moving zipcode data to SQL Server

The list of cities and the zip codes is in `ZIPCODES.TPS` and `ZIPS.TPS` (this is the downloaded file). It might have been better to create another linked server to `ZIPS.TPS`, but as I mentioned earlier, the Topspeed ODBC driver always hung my system whenever I tried doing anything complicated. For all I know, this might be a restriction on the read-only version I am using. So, I will use DTS and move the data in `ZIPS.TPS` into a staging area where I can then manipulate it.

Following the steps outlined in “Using SQL Server’s Data Transformation Services,” p. 341, I will move the contents of `ZIPS.TPS` into `dbo.ZIPS`. As `dbo.ZIPS` is non-existent in the database, I have to ensure that DTS is able to create it prior to data import.

Figure 7: Naming destination table



As shown in Figure 7, I named the destination table `ZIPS`. After I execute the DTS package, `ZIPS` contains a total of 48,253 records.

Updating the CITY table

I need to perform a series of SQL tasks before I can populate `dbo.CITY`. The first thing I need to do is to logically combine the two zipcode tables, `dbo.ZIPCODES` and `dbo.ZIPS`, in a VIEW and then extract city information from here. At this stage, I need to create a logical, rather than a physical combination due to the fact that I have to populate the foreign key `dbo.ZIPCODES.ID_CITY` with the matching IDs expected to be in `dbo.CITY`. Unfortunately, I cannot do this now because I intend to populate the `CITY` table from the zipcodes tables. My planned stages of conversion are:

- 1) Create a VIEW that combines `dbo.ZIPCODES` and `dbo.ZIPS` into one result set. This view will contain the same rows expected to be in the final version of `dbo.ZIPCODES`. Below is the script that creates the view. The `UNION` keyword combines the output of two result sets into one single result set.

```
CREATE VIEW ALLZIP AS
/*
    Rows in dbo.ZIPCODES that do not have duplicates in
    dbo.ZIPS retrieved
*/
SELECT ZIP, CITY, STATE
FROM ZIPCODES
WHERE ZIP NOT IN
    (SELECT ZIPCODE FROM ZIPS)
UNION
/*
    Rows in dbo.ZIPS that do not have duplicates in
    dbo.ZIPCODES retrieved
*/
SELECT ZIPCODE, CITY, STATE
FROM ZIPS
WHERE ZIPCODE NOT IN
(
-- Query to return all rows dbo.ZIPCODES.ZIP
    SELECT ZIP FROM ZIPCODES
)
```

- 2) Create a JOIN combining the `ALLZIP` view with `dbo.STATE`. This will be used for populating `dbo.CITY`. Cut and paste the code below into Query Analyzer, and run to update the `CITY` table.

```
INSERT dbo.CITY (ID_STATE,CITY_NAME)
SELECT dbo.STATE.ID_STATE, dbo.ALLZIP.CITY
FROM   dbo.ALLZIP LEFT OUTER JOIN
       dbo.STATE ON dbo.ALLZIP.STATE = dbo.STATE.STATE_CODE
```

- 3) Export the contents of `ALLZIP` to a temporary file. This is necessary because the `ALLZIP` view is built by combining two tables in a non-efficient way, and moreover, the `ZIPS` table does not have a primary key or any indexes, so performance is poor. At the same time, the data returned by this

view is not static and is dependent on what is in `dbo.ZIPCODES` and `dbo.ZIPS`, so any updates to `dbo.ZIPCODES` will affect what is in the view. Here is the code to copy the contents of the view to a temporary file

```
SELECT * INTO #temp_zip FROM allzip
```

The `SELECT INTO` command will create a new table having the same structure as the `FROM` table, and fill it up with each row of the `FROM` table. Starting a table name with a single hash “#” indicates the table is created as a local temporary table. This table exists during the life of a connection. Once a connection is terminated, SQL Server automatically deletes the table. Also the table cannot be seen from another connection. This is different from a global temporary table, which starts with two hash signs (e.g. `##temp_zip`) and is visible to other connections, but is deleted once all connections referencing the table have been disconnected.

- 4) The next thing to do is to delete all records from `dbo.ZIPCODES`, and fill it with rows from `#temp_zip`. I also need to correct the length of one of the columns in `dbo.ZIPCODES`.

The column `dbo.ZIPCODES.CITY` is 25 characters in length whereas, the corresponding field `dbo.ZIPS.CITY` is 28 characters. This I need to take care of before I start transferring any data from `#temp_zip` since this temporary file is created from the `ALLZIP` view, which is created from a combination of `dbo.ZIPS` and `dbo.ZIPCODES`.

The script below is in four parts (separated by the `GO` keyword). The first part checks for and removes the Foreign key constraint `dbo.ZIPCODES` has with `dbo.VENDORS` (this is added back in the last part of the script). The second section `DROPS` the table; this effectively removes all the rows, as intended. The third section recreates the table only, but this time the size of the `STATE` column is increased to 30.

```
if exists
(
select * from dbo.sysobjects
where id = object_id(N'[dbo].[FK_ZIPCODES_VENDORS]')
and OBJECTPROPERTY(id, N'IsForeignKey') = 1
)
ALTER TABLE [dbo].[Vendors] DROP CONSTRAINT
FK_ZIPCODES_VENDORS
GO
if exists
(select * from dbo.sysobjects
where id = object_id(N'[dbo].[ZipCodes]') and
OBJECTPROPERTY(id, N'IsUserTable') = 1
)
DROP TABLE [dbo].[ZipCodes]
GO
CREATE TABLE [dbo].[ZipCodes]
(
```

MS SQL

```
[ID_ZIPCODES] [int] IDENTITY (1, 1) NOT NULL,
[ID_CITY] [int] NULL,
[ZIP] [varchar] (10) NULL,
[CITY] [varchar] (30) NULL,
[STATE] [varchar] (2) NULL
)
ON [PRIMARY]
GO
ALTER TABLE [dbo].[ZipCodes] WITH NOCHECK ADD
CONSTRAINT [PK_ZIPCODES] PRIMARY KEY CLUSTERED
(
[ID_ZIPCODES]
) ON [PRIMARY]
GO
```

- 5) After dbo.ZIPCODES has been modified, the next thing to do is import rows from #temp_zip .

```
INSERT dbo.ZIPCODES (ZIP,CITY,STATE) -- Fill from #temp_zip
SELECT ZIP, CITY, STATE
FROM dbo.#temp_zip
```

- 6) After this I need to map the IDs in dbo.CITY to dbo.ZIPCODES and update dbo.ZIPCODES accordingly. If you are coming from a procedural programming background, what readily comes to mind as a solution is LOOPing through all the rows (records), of which there are about 44,000 in the table. Processing many records this way is rather inefficient, which is why the Clarion Process template is not suitable for record processing when using a SQL Driver - every single record is processed one at a time. The script below is a faster way of updating all the rows in the table.

```
UPDATE ZIPCODES
SET ID_CITY = c.ID_CITY FROM dbo.CITY c
WHERE c.CITY_NAME = CITY
GO
```

You can get the script containing all these steps from the downloadable zip.

Clarion dictionary changes

It is necessary to make a corresponding change to the CITY field in the Clarion dictionary (INV_SQL.DCT). You have to change the size from 26 characters to 31 characters; if you don't do this, you will get an error when you attempt to open the ZIPCODES table. Later, I will eliminate fields like dbo.ZIPCODES.CITY, dbo.ZIPCODES.STATE and correct references to them in the application.

Summary

In this chapter, you will notice that all conversions and data manipulation have been done in SQL, because SQL is more efficient in manipulating large data sets than standard Clarion code. You may want to stick to your traditional way of data conversion, that is, using the Clarion language. However, getting comfortable with SQL will greatly improve your efficiency and give you a greater edge. I remember a poll taken a while ago by ClarionMag on developers who intend to convert their applications to SQL. I believe figures from those results would have doubled by now. It is nice having Clarion do the work of interfacing and data manipulation, but to get the most out of SQL databases you need to go beyond Clarion code and improve your SQL skills.

Source code

See “Appendix A: Getting Support,” p. 601, for information on how to get the source accompanying this book.

- v4n02listbox.zip

MS SQL

CONVERTING THE INVENTORY EXAMPLE - CALLING STORED PROCEDURES

by Ayo Ogundahunsi

In the previous chapters I showed how to connect to SQL Server, import data, and generally run a Clarion application with a SQL Server back end. In this chapter, I will show you how to call a stored procedure, pass a parameter, and (hopefully) get results from the stored procedure. I will also provide a template to make this easy to do.

The first step is to split the inventory application to an EXE and DLL(s). While this is not a specific requirement, it is better to organize your applications this way (see “Splitting Your App Into DLLs” on page 581 of the *Clarion Tips & Techniques* book, available from www.clarionmag.com). Depending on how big an application is, you may wish put all your reporting procedures in one DLL, all your browses and forms in another DLL, etc. A common practice is to have a DLL just for data elements, i.e. file declarations and other global data. The EXE and other DLLs still have those declarations, but everywhere except in the data DLL they are declared as EXTERNAL - the memory is only allocated in the data DLL.

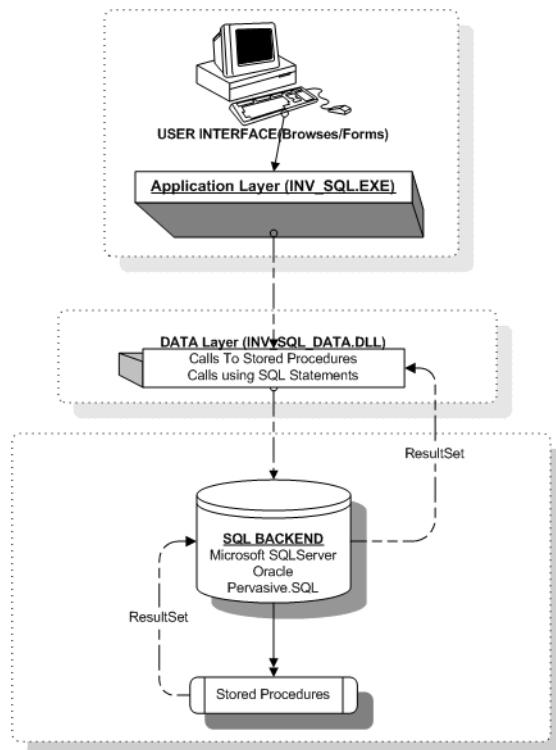
MS SQL

For the inventory application, I will be calling the data DLL INV_SQL_DATA.DLL. For the most part, this chapter will focus on the process of splitting the inventory app into DLLs.

Data access and update layers

One good reason to have a data DLL is that I want to have only one access point for all calls I make to SQL Server. In other words, I don't want to have embedded SQL all over my applications; I only want one point of maintenance, and the less embedded SQL code I have in my application, the easier it is to have the business logic external and available for non-Clarion use. Figure 1 demonstrates the data access logic.

Figure 1: Data Access Logic



Application Layer - This layer will contain the call to the procedure (located in INV_SQL_DATA.DLL) that interacts with SQL Server. For example, the procedure `ProcessPrices` uses the Process template to change the prices for all the products in

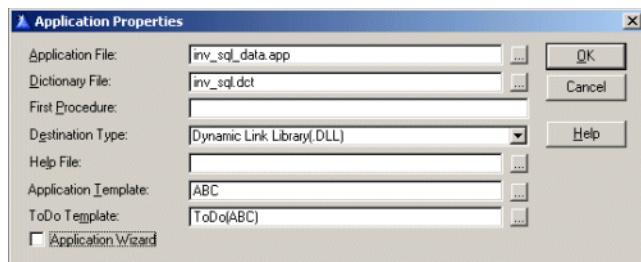
Converting The Inventory Example - Calling Stored Procedures

the Inventory. To make the application more SQL-like, I will replace this process with a stored procedure. Therefore, instead of making the call to the stored procedure inside INV_SQL.APP, I will make the call inside INV_SQL_DATA.APP.

Data Layer - This layer will contain the table definitions, all calls to stored procedures, all SQL statements that will be executed, and a connector procedure which replaces the connection string initialization (e.g. GLO:ConnectionString = '(LOCAL)', inv_sql, sa, sa') which you would otherwise put in an embed point.

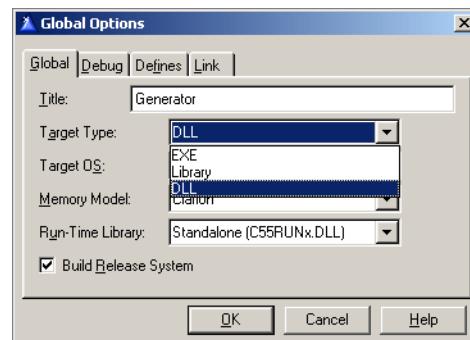
Step 1: Create a new application as a DLL, as shown in Figure 2.

Figure 2: Creating a DLL - Application Properties



Step 2: Project Settings. Make sure the project settings are correct, as shown in Figure 3.

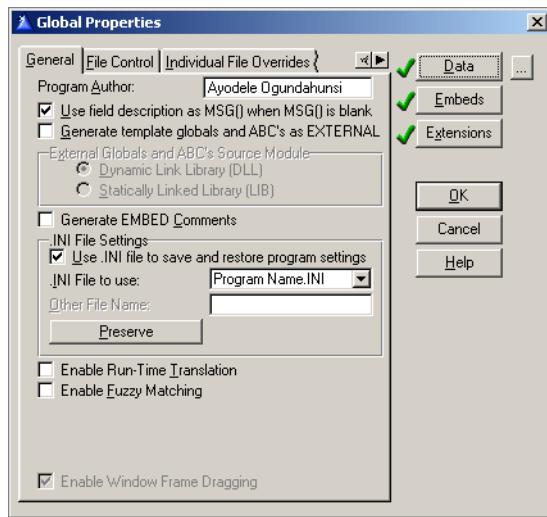
Figure 3: Creating a DLL - Global Options



MS SQL

Step 3: Set Global Properties. On the **General** tab, uncheck the **Generate template globals and ABC's as EXTERNAL** option. See the Clarion Help for more explanation on this.

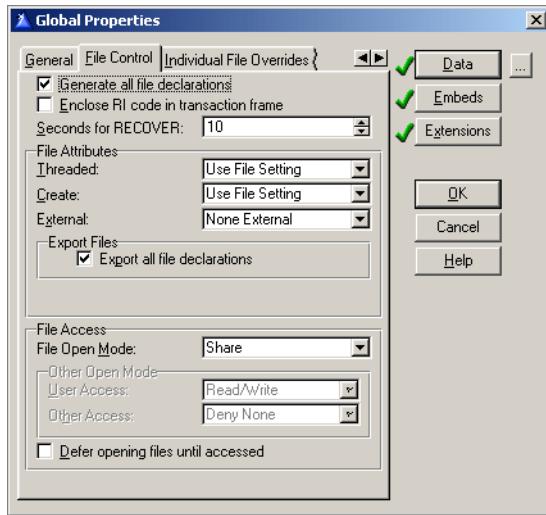
Figure 4: Setting Global Properties - General



Converting The Inventory Example - Calling Stored Procedures

Step 4: Set Global Properties - **File Control**. Check **Generate all file declarations**, change the **File Attributes (External)** to **None External**, and check **Export all file declarations**, and uncheck **Defer opening files until accessed**.

Figure 5: Setting Global Properties - File Control



Once you have done this, compile your application and you will have `INV_SQL_DATA.DLL`.

Modifying your EXE

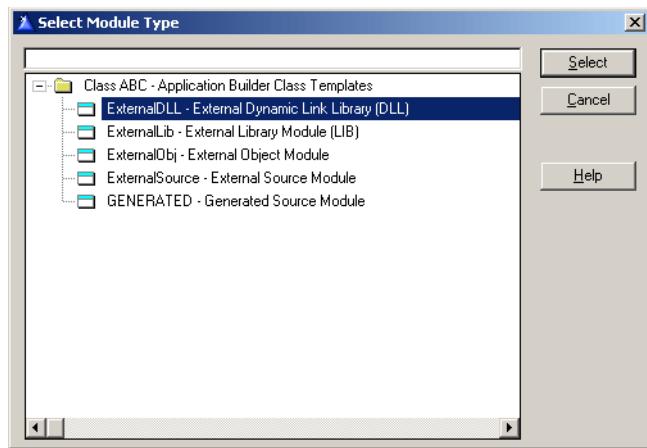
Now you need to connect `INV_SQL_DATA.DLL` to `INV_SQL.EXE`, and in doing so you must remove the data component in `INV_SQL.APP`. Close the `INV_SQL_DATA.APP` application and open `INV_SQL.APP`.

Step 1: Choose **Application|Insert Module**.

MS SQL

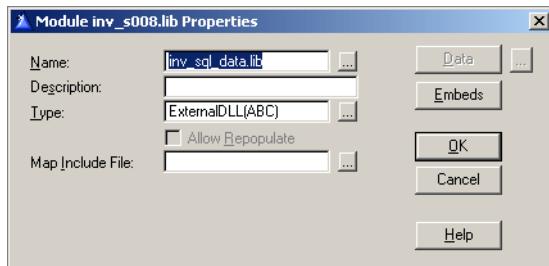
Step 2: Select a Module Type of **ExternalDLL**.

Figure 6: Selecting Module Type



Step 3: Name the module library, in this case **INV_SQL_DATA.LIB**.

Figure 7: Module Name



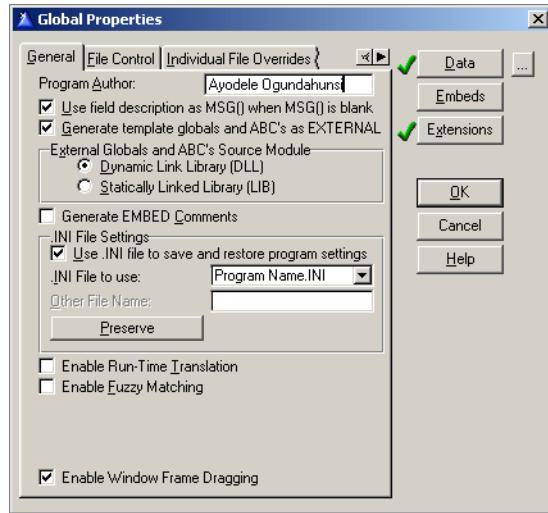
Once you are done, you will have to modify the Global Properties.

Step 4: Global Properties - General. **Check Generate template globals and ABC's as EXTERNAL**. This doesn't remove the data declarations; instead, it declares them as

Converting The Inventory Example - Calling Stored Procedures

EXTERNAL, which means they won't be allocated memory. The app must reference a LIB for a DLL that exports these declarations, which is what the data DLL does.

Figure 8: Global Properties - General (EXE)

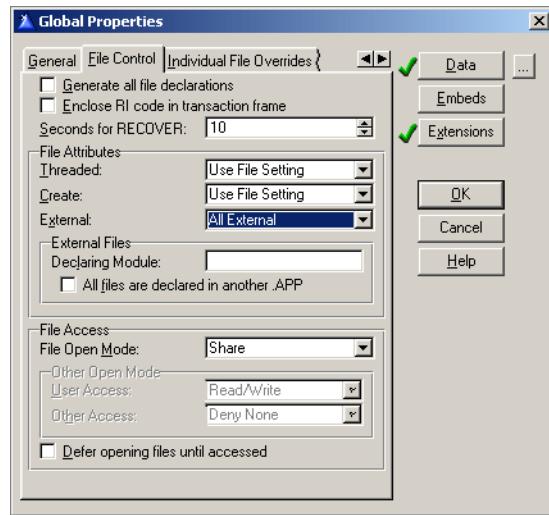


Step 5: Global Properties - File Control. Uncheck **Generate all file declarations**, which means that this app will only declare the file declarations it actually uses. Also uncheck

MS SQL

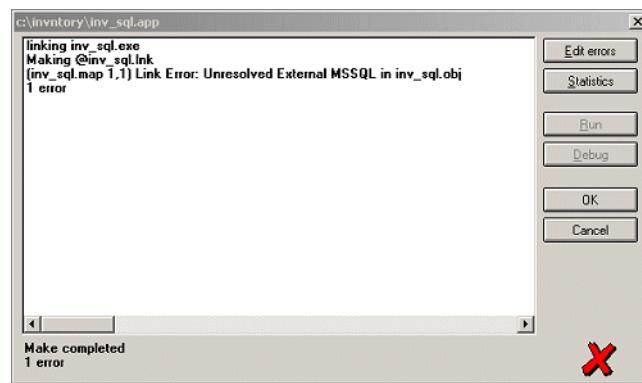
Enclose RI code in transaction frame - I always uncheck this since I enforce RI at the server

Figure 9: Global Properties - File Control (EXE)



While compiling, it is possible you will get an **"Unresolved External MS SQL in inv_sql.obj"** error, as shown in Figure 10.

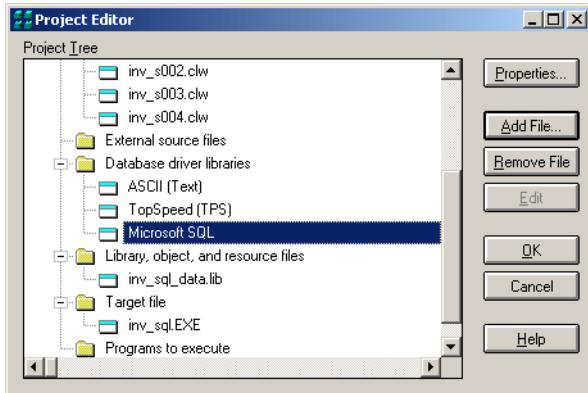
Figure 10: Unresolved external symbol compile error



Converting The Inventory Example - Calling Stored Procedures

It seems whenever you uncheck **Generate all file declarations**, the IDE removes the Database Driver Library for MS SQL Server. If this happens, just go to your Project Editor and add the driver, as shown in Figure 11.

Figure 11: Adding MS SQL Driver



Now compile again - the error should go away.

Now it's time to begin implementing stored procedures. As part of the download, I have included a template to make integration to SQL Server a lot easier, and that's the subject of the next section.

The AyoMSSQL Template

Before you can use the AyoMSSQL template you need to register `AyoMSSQL.tpl` (see the link at the end of this chapter). I will not go into full details about the inner working of this template, rather, I will describe how to use it.

The key features of this template are as follows:

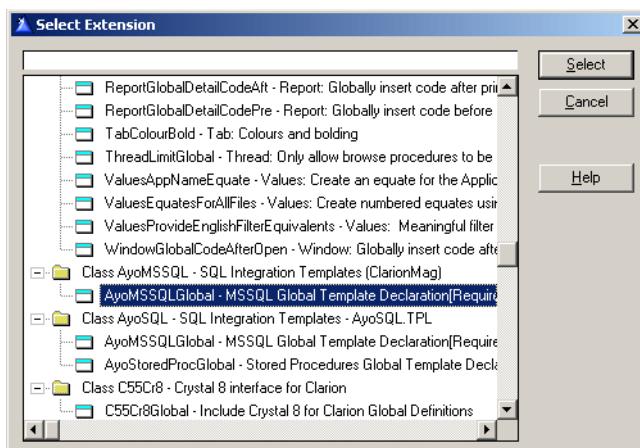
- 1) It allows you to choose the method of data access from Clarion to SQL Server. (See Step 3).
- 2) It generates the connection string. You can define the variables without embedding any code.
- 3) It generates SQL code for creating a “Dummy” table.
- 4) It generates code to save your connection string settings in an INI file.

The first place to integrate the template is at the data layer.

MS SQL

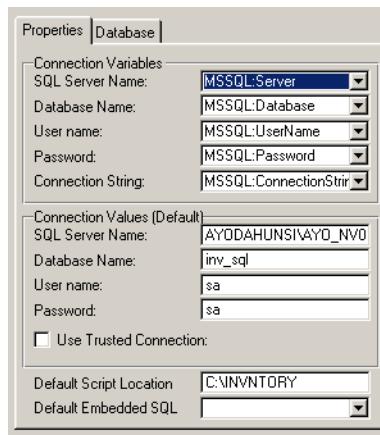
Step 1: Add the global extension to `INV_SQL_DATA.APP`.

Figure 12: Global Extension



Step 2: Set the global properties. The first tab under the **Global Extension** template allows you to set **Connection Properties**, **Connection Values**, etc.

Figure 13: Global Extension - Properties

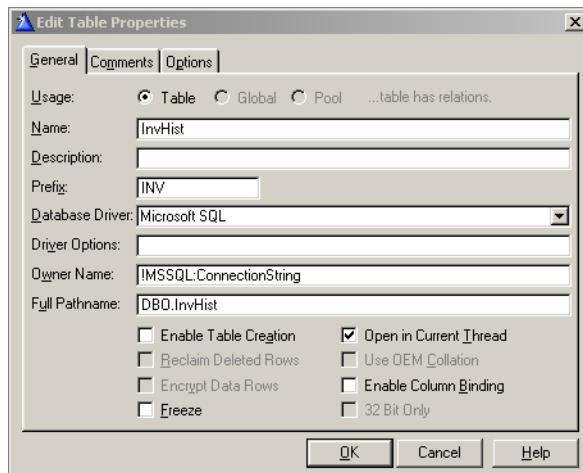


Connection Variables: These contain template created/generated variables that are used to form the connection string. Though you have the ability to change the name of the variables, it isn't necessary to do so. Note that all global variables, including those defined in the dictionary, are available in the drop list. However, since my connection string variable is now going to change from `GLO:ConnectionString` (as I defined it when I started this

Converting The Inventory Example - Calling Stored Procedures

series) to `MSSQL:ConnectionString`, I need to reflect this change in the Owner Name field for table properties of every MS SQL table defined in the dictionary. Figure 14 shows the required change for the `IN VHIST` table.

Figure 14: Table Properties



Connection Values: The values entered here (see Figure 13) will probably serve well during testing. However, for deployment, you would need to make these values updateable. That is why the connection variable described above becomes useful - you can write your own code to prime the variables.

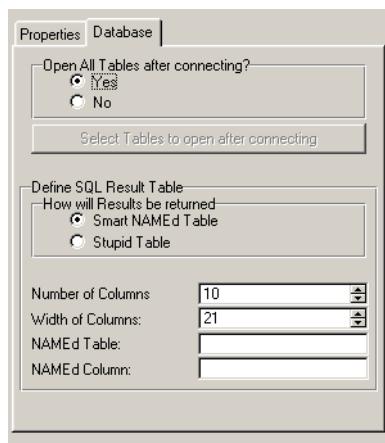
Trusted Connection: As a recap, a trusted connection simply means that SQL Server will trust Windows NT/2000 to verify the user's password. Therefore on a Windows 98 machine this will not work. When **Trusted Connection** is checked, the **Username** and **Password** fields become disabled. This could be useful if you use Windows Authentication Mode in setting up your SQL Server (see "Migrating The Inventory Application To SQL Server," p. 317 for information on authentication modes).

Default Script Location: Some scripts will be generated in the course of using this template. They will be output in the location defined here.

Default Embedded SQL: The procedure selected here contains code that calls the `PROP:SQL` statement. `PROP:SQL` is never called anywhere else in this application. The main advantage is that you can isolate all your calls to SQL Server. There are two extension templates that generate this procedure, and which I'll explain a little later.

Step 3: Global Extension - Database. The second tab contains database related settings. These settings affect how data is returned (when stored procedures are called), and how tables are accessed during startup.

Figure 15: Global Extension - Database



Open all Tables after connecting: In Relational Database Management Systems (RDBMS), all tables are treated as one single unit - the database. What this means is that you don't really have CLOSEing and OPENing of tables during the life of an application like you see in flat file systems. Clarion gives you the option of opening your tables when they are first accessed. If you decided to set up your system this way, a browse that displays information from a very big table (that is, many fields/columns) will take some time to load. This setting forces all the tables to be opened during startup. Psychologically, a user can bear with an application taking some time to load, rather than a browse loading very slowly when the application has been in use for some hours. This setting will display the table name in the title bar as every table is being opened. You can also decide to not force all tables to be opened during startup.

How will Results be returned: When returning results from SQL, you need an intermediate structure to catch the result. This can be a VIEW, or a TABLE. I will not get into details of what the code looks like, however, I will mention the logic behind this.

Stupid Temp Table Theory

Several years ago Troy Sorzano publicized what he called the Stupid Temp Table Theory (Editor's note: you can still read Troy's description via the Wayback Machine at <http://web.archive.org/web/19980527022855/www.cwsuperpage.com/Articles/SQLFAQ/S>

Converting The Inventory Example - Calling Stored Procedures

TTT2.htm). This theory simply means you use a physical table to capture the results from a SQL back end. The Stupid Temp Table approach in this article has been revised. Though my approach still requires a table on the server, I use a VIEW structure to display the results. Also, the column's data type is CSTRING. Clarion's automatic type conversion makes it easy to retrieve numeric data into a string data type, and the NAME attribute is not used in any column. In order to make this work, you will have to create a table (for the returned result set) on SQL Server. In my template, this VIEW structure is passed as a parameter to a common procedure that is created by a procedure template.

During compilation, a script file is generated and saved in the directory you specified under Default Script Location (Step 2 above). It is the "Stupid" or Dummy table structure located in SQL Server. You will have to run this script via the Enterprise Manager or OSQL.EXE before you run your application in order to create the table in SQL Server.

The generated script looks like this:

```
if exists (select * from dbo.sysobjects
where id = object_id(N'[dbo].[SQLRESULT]')
and OBJECTPROPERTY(id, N'IsUserTable') = 1)
drop table [dbo].[SQLRESULT]
GO
CREATE TABLE [dbo].[SQLRESULT] (
    [Column1] [varchar] (21) NULL,
    [Column2] [varchar] (21) NULL,
    [Column3] [varchar] (21) NULL,
    [Column4] [varchar] (21) NULL,
    [Column5] [varchar] (21) NULL,
    [Column6] [varchar] (21) NULL,
    [Column7] [varchar] (21) NULL,
    [Column8] [varchar] (21) NULL,
    [Column9] [varchar] (21) NULL,
    [Column10] [varchar] (21) NULL,
    [ID_SQLRESULT] [int] IDENTITY (1, 1) NOT NULL
) ON [PRIMARY]
GO
ALTER TABLE [dbo].[SQLRESULT] WITH NOCHECK ADD
CONSTRAINT [PK_RESULT] PRIMARY KEY CLUSTERED
(
    [ID_SQLRESULT]
) ON [PRIMARY]
GO
```

Smart NAMEd Table Theory

The Smart NAMEd Table Theory borrows some concepts from Troy's method in the use of the NAME attribute. However, no physical file has to exist in order for this approach to work. The basic concept is to fool the Clarion SQL driver into thinking that a table exists. The driver always compares table definitions in the dictionary with what exists on SQL Server, so if you give tables and fields external names that match names existing in your

MS SQL

SQL Server, the driver will think the table actually exists. (See Clarion help for explanation on using the NAME attribute). In the template, a table (*FILE) structure is passed as a parameter to a common procedure created by a procedure template.

The remainder of the settings from Figure 15 are as follows:

Number of Columns: This controls the number of columns being returned by your SQL query. If you want to execute a stored procedure that will return 15 rows of data, then you have to increase this to 15.

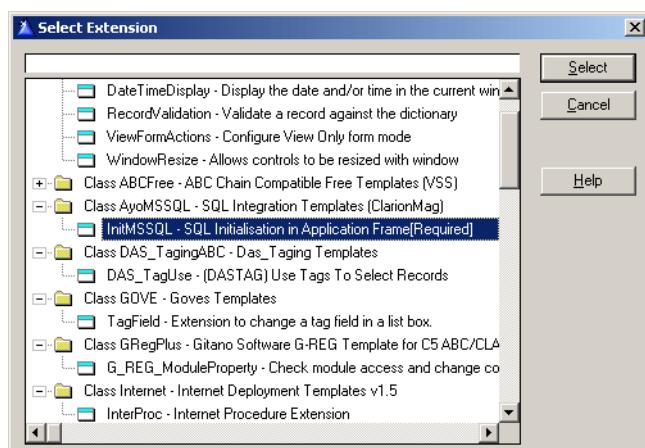
Width Of Columns: If you are going to be returning text that will be more than 21 characters in length, then you have to increase this size.

NAMED Table: This is the name of any existing table. Note that a table structure will be created that will be NAMED based on what you type here. However, the table will not be physically created on the SQL Server.

NAMED Column: This is any column that belongs to the table defined in the previous field.

Step 4: Extension Template - initialization code. The code that establishes a connection to SQL Server is automatically generated by this template. The variables used to create the connection string come from steps 2 and 3.

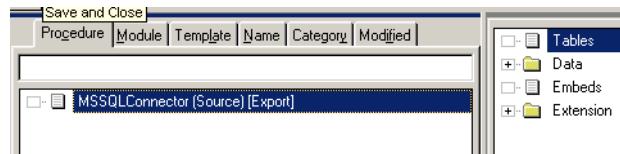
Figure 16: Extension Template - Initialization Code



Converting The Inventory Example - Calling Stored Procedures

The best approach in using this template is to create a source procedure, and then add this template as an extension to the source template. In this example, I will call this source procedure MSSQLConnector.

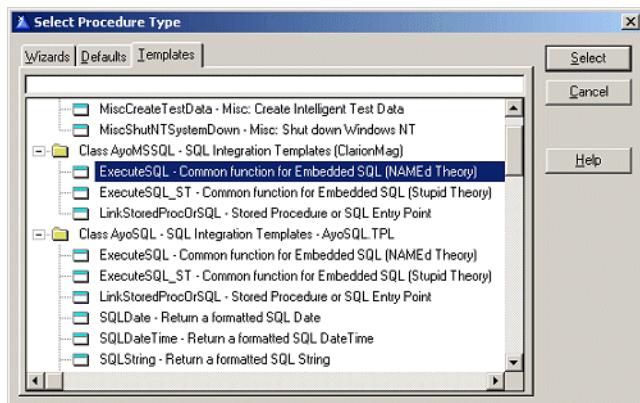
Figure 17: Extension Template - Connector



This procedure will be called in the Main frame of INV_SQL.APP.

Step 5: Procedure Template - ExecuteSQL. As I mentioned in Step 2, the code that calls PROP:SQL has been isolated to one single procedure. This procedure can be created from a procedure template supplied with this SQL integration template.

Figure 18: ExecuteSQL - PROP:SQL encoding



These two procedure templates basically do the same thing. The only difference is that one makes a call to the server using the Stupid Temp Table Theory, while the other uses the NAMED Table theory. I have named the procedures I created using these templates `SendSQLString1` and `SendSQLString2`.

So far, everything that has do with a connection or any call to SQL Server has been restricted to INV_SQL_DATA.APP, which is the initial idea of grouping such functionality under the Data Access Layer.

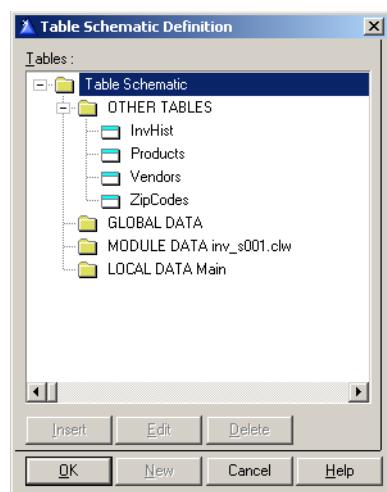
I can now go ahead and compile INV_SQL_DATA.APP.

MS SQL

Step 1: Adding External Procedure(s). The first thing I need to do here is add the MSSQLConnector procedure as an **External Procedure**.

Step 2: Removing Tables from File Schematic. In “Migrating The Inventory Application To SQL Server,” p. 317, I added the tables INVHIST, PRODUCTS, VENDORS, and ZIPCODE to the table schematic of the main frame. These have to be removed since the external procedure MSSQLConnector handles this functionality.

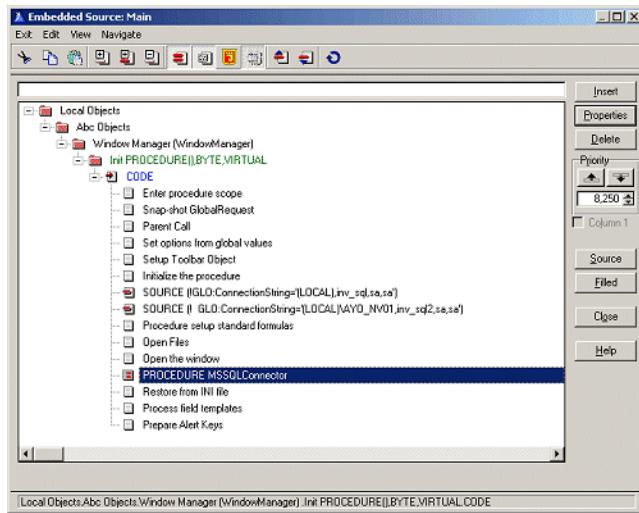
Figure 19: Removing Tables from File Schematic



Converting The Inventory Example - Calling Stored Procedures

Step 3: Adding the procedure. Comment out or delete the embedded code where a value is assigned to GLO:ConnectionString, and then add MSSQLConnector to an embed point immediately after Open the Window.

Figure 20: Adding MSSQLConnector to embed



Now that you know how to use the supplied template, it's time to learn how to call a stored procedure. But first, it is important that I mention some tools that will help you troubleshoot activity between your Client application and SQL Server. For someone new to SQL, it can be a daunting task when your application fails and you have no idea about what is going on. Two useful programs that can help you to debug SQL activity are Clarion's TRACE.EXE and the MS SQL Profiler.

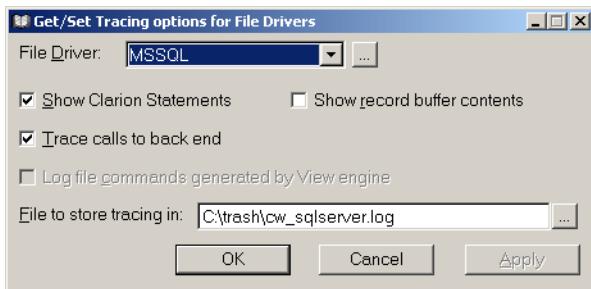
TRACE.EXE

TRACE.EXE is a Clarion program that allows you to trace input/output from any Clarion file driver. It is located in your Clarion installation's `bin` subdirectory. The shortcut to this

MS SQL

program is usually added to your Clarion menu under the Tools submenu. When you run TRACE.EXE you'll see the window shown in Figure 21.

Figure 21: Setting Trace Properties



Remember to stop the trace once you are done using it, because this file can really grow big and will slow down your application considerably.

The trace is in text form and can look cryptic sometimes, but with a little effort you shouldn't find it too difficult to read.

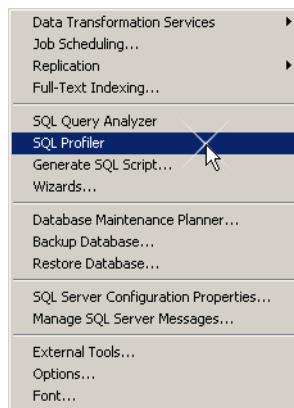
MS SQL Profiler

Profiler is a utility in SQL Server. Using this program you can log the actual SQL commands that are executed by your client application. If you use the Enterprise manager a lot, you might want to capture SQL code. Also, any SQL code executed after the profiler starts (that could be as a result of the PROP:SQL command, or any stored procedure) is captured by the profiler. The profiler is very useful in this regard.

Converting The Inventory Example - Calling Stored Procedures

You can run the profiler from the Enterprise manager, as shown in Figure 22.

Figure 22: Calling Profiler from Enterprise Manager



Or, from the SQL Server menu, as in Figure 23.

Figure 23: Calling Profiler from Program Menu



MS SQL

When you execute the stored procedure example in this chapter, the Profiler will capture an output as shown in Figure 24.

Figure 24: Viewing Trace from Profiler

The screenshot shows the Microsoft SQL Server Profiler interface. The title bar says "SQL Profiler". The main window displays a trace titled "Untitled - 1 (AYODAHUNST\AYO_NW01)". The "Event Class" column lists "TraceStart", "SQL:BatchCompleted", "RPC:Completed", and "RPC:Completed". The "TextData" column shows T-SQL code. The ">LoginName" column shows "sa" for all events. The "CPU" and "Reads" columns show resource usage. Below the table, the T-SQL code is displayed in a large text area:

```
declare @P1 int
set @P1=0
declare @P2 int
set @P2=16388
declare @P3 int
set @P3=8196
declare @P4 int
set @P4=0
exec sp_cursoropen @P1 output, N' EXEC INV_ProcessPrices 1.05 ', @P2 output, @P3
select @P1, @P2, @P3, @P4
```

The status bar at the bottom says "Ready".

The profiler is not available with Microsoft Desktop Engine (MSDE), a trimmed down version of SQL Server.

Why stored procedures?

One of the strengths of Clarion lies in the fact that you can roll out applications easily and quickly. From standard report/browse/form handling to automatic data type conversion to threading (this will probably change with the upcoming version 5.6), a lot is done behind the scenes. As a result, there is always the tendency to want to write as much as you can in Clarion. Unfortunately, this might not be the best approach when you are using Clarion with a Relational Database Management System (RDBMS).

The first step in migrating an application to SQL Server is to get the application to run. After conversion, you need to examine parts of your code that result in frequent trips to the server, and then try to optimize by moving some processing to the server.

A stored procedure in simple terms is your code written in SQL syntax and stored at the server, where they become part of the database just as tables are part of the database.

There are many books on SQL that explain more about stored procedures so I won't go into a lot of detail here.

The ProcessPrices procedure

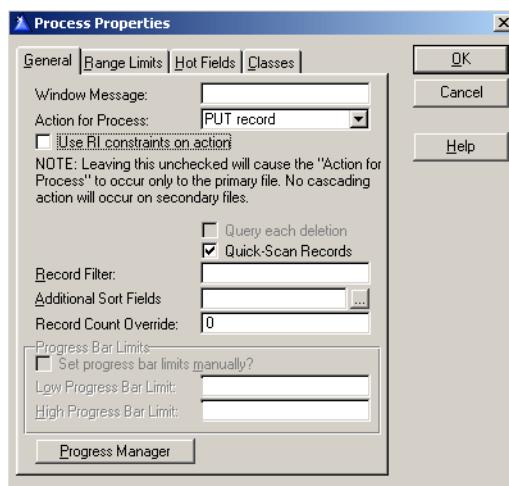
In INV_SQL.APP, there is the ProcessPrices procedure that is used to do a batch update on all product prices based on a specified percentage. If, after compiling the application, you try to run this procedure you get a **Record Not Available** error as shown in Figure 25.

Figure 25: Error after conversion



One lesson to learn here is that straight conversion will not always guarantee your application will work out of the box. A lot of tasks handled by the template when you are using a flat file system might not work when you convert to SQL. I don't want to get into too many details about this particular error, but, if you uncheck **Use RI constraints** on Action and recompile, the error disappears and the update is done.

Figure 26: Process Template - Properties



Take a look at the cost in bandwidth of doing the update the traditional (flat file) way. If you look at the Clarion Trace program, you will see that the SQL UPDATE command was called repeatedly, once for each record in the PRODUCTS table.

```
NEXT(VIEW:416AA4:DBO.Products 2) Time Taken: 0.00 secs
```

MS SQL

```
POSITIONfile(VIEW:416AA4:DBO.Products 2) Time Taken: 0.00 secs
Closing Statement 1104f08 Time Taken:0.00 secs
Resetting Parameters Statement 1104f08 Time Taken:0.00 secs
Preparing Statement 1104f08 : UPDATE DBO.Products SET "PRICE" = ?
    WHERE "ID_PRODUCTS" = ? Time Taken:0.00 secs
Binding ? 1 for input with C type CHAR as 3 for Statement
    1104f08 Time Taken:0.00 secs
Binding ? 2 for input with C type SLONG as 4 for Statement
    1104f08 Time Taken:0.00 secs
Executing prepared Statement 1104f08 Time Taken:0.00 secs
PUT(VIEW:416AA4:DBO.Products 2) Time Taken: 0.01 secs
```

This code (extracted from the Trace file), is executed 32 times (there are 32 records in the PRODUCTS table), which means the client machine was the one initiating and executing the update logic. Even though this might not seem quite a problem, imagine what happens when you try to update about 200,000 records. This means this command will be called 200,000 times!

This kind of repetitive client/server interaction negates the purpose of developing an application to run in a SQL environment. Assuming I want to increase all prices by 5%, the best thing would have been to send a SQL command like this:

```
UPDATE PRODUCTS
SET PRICE = (1.05 * PRICE)
```

However, if I embed this code in my Clarion application, and call it via PROP:SQL, no other application will be able to use it.

A better approach is to create a stored procedure that receives a parameter for the price factor (i.e. 1.05). In that way any other program, such as one written in Visual Basic, or perhaps an ASP web application, can also call this update.

In short, one very big advantage of using stored procedures is that you can remove most of your business logic from your Clarion application and make it part of your database.

That's the theory; now it's time to replace the ProcessPrices procedure with a stored procedure.

The ProcessPrices stored procedure

The code to create the stored procedure to update the product prices is as follows:

```
CREATE PROCEDURE dbo.INV_ProcessPrices
(
    @PercentageIncrease decimal(7, 2)
)
```

Converting The Inventory Example - Calling Stored Procedures

```
/*
Object:      INV_ProcessPrices
Description: Allows you to increase the prices of all
              items in inventory
Usage:       INV_ProcessPrices @PercentageIncrease=1.5
              -- 5% increase
Returns:     (None)
Author:      Ayodele Ogundahunsi Email: ayodele@dolasoft.com
Revision:    1
Example:    INV_ProcessPrices @PercentageIncrease=1.5
Created:    2002-09-30.
*/
AS
SET NOCOUNT ON
BEGIN
    UPDATE PRODUCTS
        SET COST = (COST*@PercentageIncrease)
END
RETURN 0
```

Note that it is not a very good idea to name your stored procedure starting with `sp_`. While `sp_` is not a reserved word, it means “special” within the context of SQL Server. There are a lot of special system stored procedures shipped with SQL Server, and you don’t want to confuse them with your stored procedures.

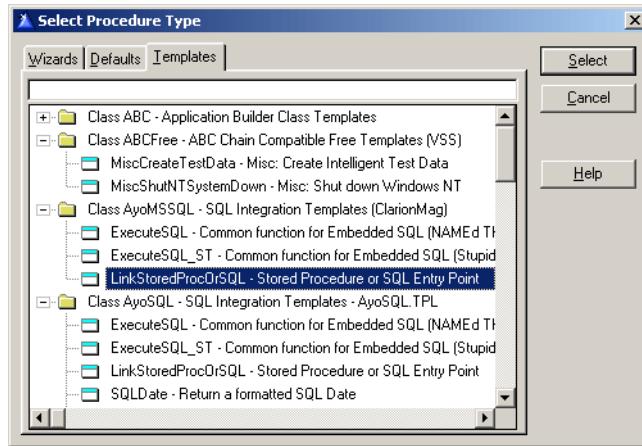
Linking INV_ProcessPrices to Clarion

Now that I have created the stored procedure `INV_ProcessPrices`, I will need to link it with Clarion. I have included an extension template in the downloadable source that makes this easy. This extension template has been designed to create a procedure that wraps all calls and interfacing with SQL Server in one single place. This approach makes it possible to create an external procedure in another EXE or DLL that calls a procedure

MS SQL

created here. As a standard, I will call my procedure the name of the stored procedure I am calling, in this case, `INV_ProcessPrices`.

Figure 27: Adding Stored Procedure Template



This extension allows you to call and execute a stored procedure. In the full template version (which is not supplied here - email me at ayodele@dolasoft.com for information on availability), you can also pass a SQL statement to SQL Server or even run a script file.

LinkStoredProcOrSQL Extension Template

The template prompts, as shown in Figure 28, are quite easy to follow.

Prototype: The prototype prompt follows the same standard as we are all used to except that you can enter the data type only. What this means is that a prototype like this is illegal:

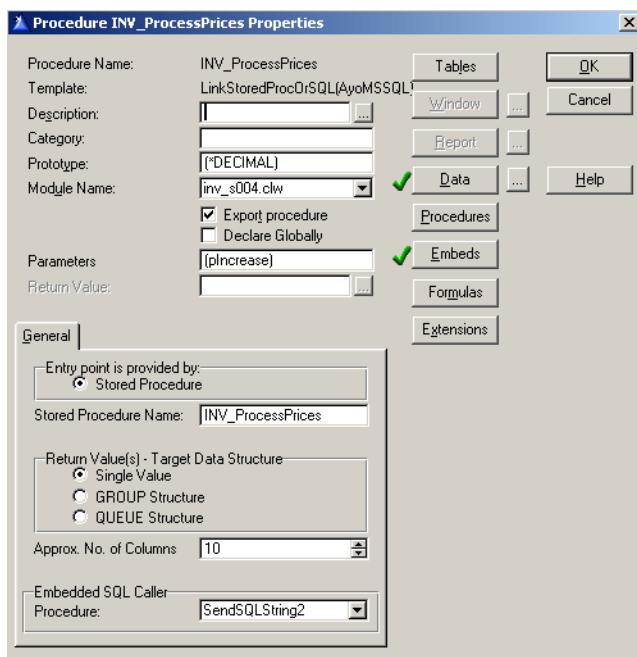
```
(*DECIMAL pIncrease)
```

Rather, you have to enter:

```
(*DECIMAL)
```

Converting The Inventory Example - Calling Stored Procedures

Figure 28: Stored procedure extension template



The complete template set (not included as part of this chapter) allows you to return results from a stored procedure or SQL query into a Queue. This is how it works; if you want to return a result set from the called stored procedure or query, you have to pass a TYPED Queue or Group as the last parameter in the prototype. For example, within an include file, or in your data section, you can define a TYPED Queue this way:

```
myQueue_TYPE    QUEUE, TYPE
Field1           CSTRING(10)
Field2           CSTRING(20)
Field3           LONG
Field4           LONG
Field5           LONG
Field6           LONG
Field7           LONG
Field8           LONG
Field9           LONG
Field10          LONG
END
```

In the case of the inventory example, I have to define this in `INV_APP_DATA.APP`.

MS SQL

My prototype then becomes:

```
(*DECIMAL, myQueue_TYPE)
```

I now define the actual queue inside INV_SQL.APP:

```
myQueue LIKE (myQueue_TYPE)
```

When this procedure is called from INV_SQL.APP, it will be called this way:

```
INV_ProcessPrices( PriceIncrease, myQueue )
```

After execution, myQueue will automatically get filled with the returned result set.

Parameters: The name of the parameter passed is entered here.

Entry Point: Normally, this contains two radio buttons. You can contact me if you want to purchase the template. The other one (not shown) allows you to type SQL query directly into the template, or even execute queries from query stored in an external file.

Stored Procedure Name: The default value generated here is the name of the procedure. If your procedure name matches the name of your stored procedure (as I suggested earlier), then you don't need to make any changes.

Return Value(s): Within the context of this chapter, you can leave the **Single Value** radio button selected as it is by default. The other selections **Queue**, **Group** are only useful if you are passing either a Queue or Group as the last parameter because you want values returned into them.

Approx. No. of Columns: This is an approximate number of columns to be returned by the stored procedure (just make sure you don't specify fewer columns than the stored procedure returns). Note that the number of fields described in your TYPED definition for a Queue or Group should match the number indicated here.

Embedded SQL Caller: By default, the selection you made in the Global Template (see Step 5) will appear. To recap, the procedure(s) that appear here determine how the data access you are using is implemented. You can use the StupidTempTable Theory, or Smart Named Table.

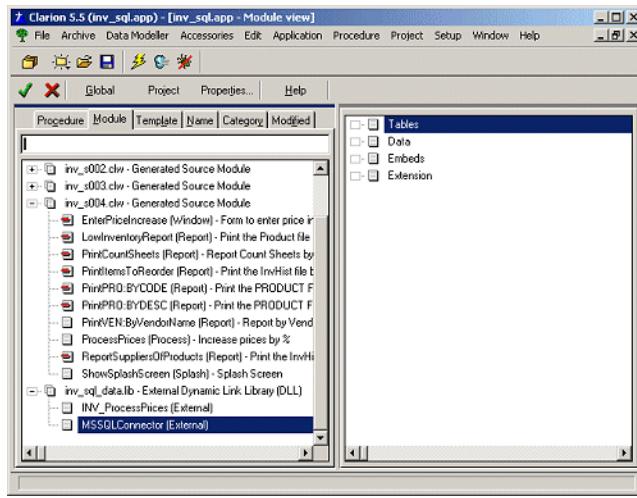
After you have filled all the prompts, you can compile. The stored procedure is executed by code generated from the template.

Converting The Inventory Example - Calling Stored Procedures

Adding INV_ProcessPrices to (INV_SQL.APP)

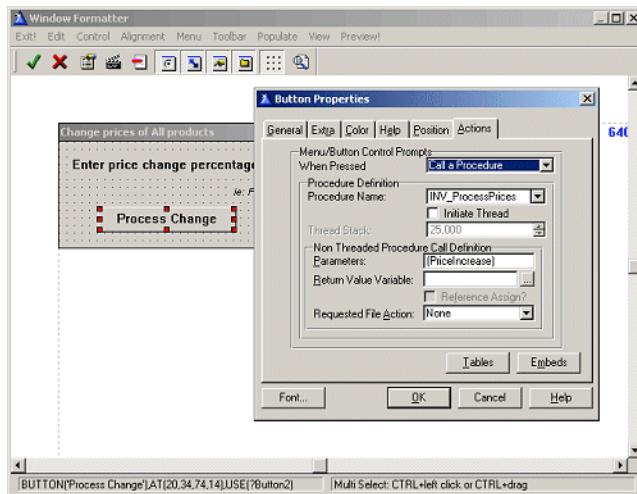
Now that `INV_SQL_DATA.DLL` contains the call to the stored procedure, I need to create an external procedure that will allow me to call `INV_ProcessPrices` from `INV_SQL.APP`.

Figure 29: Adding INV_ProcessPrices



Now that I have added `INV_ProcessPrices` procedure to the application, I need to replace the `ProcessPrices` process procedure.

Figure 30: Calling New Price Increase Procedure



I'm now ready to save and compile the application. When I execute the **Price Increase** function from the menu, it will call the stored procedure.

Summary

I have shown you how to call a stored procedure using the supplied template. While this template has been crippled (contact me at ayo@dolasoft.com if you will like to have the full version) it still allows you to execute a stored procedure. I also mentioned two powerful tools that can aid you in debugging your SQL queries.

From this chapter, you can see that I have been able to isolate all my SQL coding to `INV_SQL_DATA.APP`. In fact, there isn't any embedded SQL code. I don't dispute the fact that sometimes this is not possible, especially when you want to be able to manipulate browses. However, coding your applications with external access and reuse of business logic in mind will save you a lot of maintenance headaches in the future.

Source code

See "Appendix A: Getting Support," p. 601, for information on how to get the source accompanying this book.

- `v4n10conv2.zip`

SQL IDENTITY: ANOTHER APPROACH

by John Griffiths

Having researched Identity field management from the Client-Side (in Clarion) and the Server-Side (in MS-SQL 2000) I determined that neither worked adequately in a project I was working on. The specifications called for inserting a series of child records at the same time as the parent record was inserted, i.e. multiple task records for each job record (JOB <—> TASKS). There was also another need to set each sysID into a certain range depending on something called the branchID. In that project each sysID was set as a DECIMAL (12, 0) and held the three digit branch number in the billions area. Thus for a branch numbered 333, IDs would range 333,000,000,001 through 333,999,999,999. The long term plan was to move data from all branches into one database and continue to be able to do inserts which retained the numbering sequences.

From that project came the method I describe here for managing the sysIDs of each table within the SQL database. The method described here is simplified by ignoring the BranchID feature and uses integer fields for the sysIDs. The sysID fields in the database tables do not have the IDENTITY property.

Creating the sysIDs

Basically, the SQL database manages the supply of the sysIDs. The program simply says “Give me the next number for this table”.

The system uses a table, called `nextNumTbl`, with two fields, `nextnum` and `tblname`.

When the Clarion program needs to do an insert, it calls a stored procedure in the database that supplies the next ID number for the table specified in the call. Each call to the stored procedure will return a unique unused ID for the table specified. A return of zero will signify an error.

The stored procedure works by selecting the appropriate `nextnum` field from the `NextNumTbl`, increments the ID and tries to update the `NextNumTbl`. If it succeeds, it returns the ID and the `NextNumTbl` will now hold the next available number.

This process makes it easy to manage inserts of child records without having to rely on the SQL `IDENTITY` processes; `@@IDENTITY`, `IDENT_CURRENT()` or `SCOPE_IDENTITY()`. Your program then has the parent ID and any child IDs needed for data integrity.

In my SQL tables, each table that needs managing this way has a field named `sysID` of type `int`.

The `NextNumTbl` has a row for each table. Here is the `CREATE` statement for `NextNumTbl`:

```
create table dbo.nextnumtbl (
    nextnum int null,
    tbl      varchar(20) not null) --adjust:widens to suit
```

Data in the table may look like this:

2091	ACNOTE
7123	JOB
59003	TASK
1873	CLIENT

The stored procedure call

I need to call the stored procedure anytime I insert a record. Here is a call from within Clarion to fetch and prime the next sysID for the `acnote` table:

```

tblName  = 'ACNOTE'
tblName = UPPER(tblName)
SQLcallString = 'CALL jgNextID ( ' & TblName & ')'
    TempF1{PROP:sql} = SQLcallString
NEXT(TempF1)
GotSysid = TempF1.F1
IF GotSysid > 0
    ACN:Sysid = GotSysid
ELSE
    ACN:Sysid = 0
    ! Handle error here
END

```

Note: TempF1 is a dummy table with one column of type CSTRING, and TblName, SQLcallString and GotSysid are local variables.

The jgNextID Stored Procedure

The jgNextID stored procedure is called with the name of the table as a parameter. It looks up the next number and returns the number via the dummy table buffer.

Here is the CREATE statement for this stored procedure, with notes below describing significant lines:

```

1 CREATE PROCEDURE jgNextid ( @tblName varchar(20) )
2 AS
3 DECLARE @nextid int ,
4          @maxtimes int
5 set nocount on
6 set @maxtimes =10
7 while (@maxtimes > 0)
8 begin
9     SET TRANSACTION ISOLATION LEVEL REPEATABLE READ10      begin
transaction11      -- get the next number to use

```

Line 6: Set the number of tries the procedure will use to get a valid number. With a site having twenty users inserting data and a value here of 5, I have not seen any collisions

Line 9: Set isolation level to protect the transaction

Lines 12,13: The select statement to fetch the next available number

Lines 15-18: Attempt to update the table where the nextnum matches the value just fetched. This ensures a good number held by no one else.

Lines 19-23: If the update was successful, break from the while loop.

Lines 24-29: The update failed, so continue loop.

Lines 31-32: If the loop counter hit zero, set @nextID to 0

MS SQL

Line 33: Select the value in @nextID for collection by the Clarion program.

Management stored procedure

What if you want to reset the starting number for the next sysID? To manage the NextNumTbl another stored procedure was developed. This one is called for each table and sets/resets the nextnum field as required. Again, see the notes below for what is happening.

```
1 CREATE PROC jgSetNextNumFor( @tblName  varchar(20))
2 AS
3 DECLARE      @nextInt integer ,
4             @gotMax integer ,
5             @selStr  varchar(201) ,
6             @currentNextInt  integer
7
8 SET @selStr='SELECT n1 = MAX(sysid) INTO ##JJ8 FROM '
9     + @tblName
10 BEGIN
11     EXEC (@selStr )
12     SELECT @gotMax = n1 FROM ##JJ8
13     DROP TABLE ##JJ8
14 END
15 IF @gotMax = 0 OR @gotMax is NULL
16     SET @nextInt = 1
17 ELSE
18     SET @nextInt = @gotMax + 1
19
20 SELECT @currentNextInt = nextnum FROM NextNumTbl
21     WHERE tbl = @tblName
22 IF @currentNextInt is NULL OR @currentNextInt=0
23     BEGIN
24         INSERT INTO NextNumTbl VALUES (
25             @nextInt ,
26             @tblName )
27         RETURN
28     END
29 IF @currentNextInt < @nextInt
30     UPDATE NextNumTbl
31         SET nextnum = @nextInt WHERE tbl = @tblName
```

Lines 8-9: Build a string needed for the select statement at line 11. As that EXEC call will be in a separate scope, the local variables are not visible so their values need to be put into the select string. Table ##JJ8 is created as a global temporary table.

Line 11: This will execute the select string and places the result into a field named N1 in the global temporary table named ##JJ8 (These temporary table names are not significant – use whatever you like)

Line 12: Collect the table's maximum value here and place it into the local variable.

Line 13: Drop the temporary table.

Lines 15-16: If the result is a 0 or null value, it shows that there were no records for that table, so set the next number to 1

Line 18: Bump the max value by 1

Lines 20-21: See what value exists in the NextNumTbl for the table specified.

Lines 22-28: If there is no valid number there, go ahead and insert one.

Lines 29-31: If the number available in the NextNumTbl is less than the MAX (+1) value in the table, then update NextNumTbl with the next valid number.

Extension Template

Here is an extension template for adding to standard forms. It will do the necessary field priming for the table inserts. This is my first foray into template programming, but it works!

```
#TEMPLATE (JGsql14, 'SQL Table Identity Aid by JLG')
#Extension(JGAutoIncSysid, 'For insert priming from StoredProc
jgNextNum')
#boxed('SQL JG SP_AutoInc')
#DISPLAY('Enter the table Clarion prefix:-')
#PROMPT(' Table Prefix Code:',@S3),%TablePre
#DISPLAY('Next name the SQL table:-')
#Prompt(' Table for Insert ',@s20),%JGFile
#DISPLAY('')
#endboxed
#at(%DataSectionBeforeWindow)
tblName      CSTRING(21)
GotSysid     LONG,auto
SQLCallString cstring(401),auto
#endat
#AT(%PrimeFields,'Prime record fields on
Insert'),WHERE(%InsertAllowed)

#DECLARE (%JLGTblSysid)
tblName = '%JGFile'
tblName = upper(tblName)
SQLcallString = 'call jgNextID ( ' & TblName & ')'
Open(TempF1)
TempF1{PROP:sql} = SQLcallString
next(TempF1)
GotSysid = TempF1.F1
#set(%JLGTblSysid,%TablePre & ':' & 'Sysid')
if GotSysid > 1
    %JLGTblSysid = GotSysid
else
    %JLGTblSysid = 0 !and insert should fail
end
```

MS SQL

```
    Close(TempF1)
#EndAT
```

Summary

This method of handling identity values for child inserts from within a parent record has worked well in a large multi-DLL app which was moved several years ago from Clarion DAT files to MS-SQL2000. While it takes a little extra setup work, it simplifies sysID management for child inserts. Thanks go to George Hale for insight on how this could be achieved.

CREATING UTILITIES FOR MS SQL 2000

by Bernard Gosperrin

Quite often it happens that Clarion programmers see only limitations when using SQL from within Clarion, as compared to other tools. In fact, this limit is mostly in our heads. Clarion's PROP:SQL makes it easy to execute arbitrary SQL statements in Clarion, and a few tricks simplify the task of getting information back from these SQL statements.

In this chapter I will show how you can create utilities for an SQL database with PROP:SQL, using the example of MS SQL 2000. If you are familiar with the Clarion community's most popular Internet sites, you may have noticed an excellent set of articles by Dan Pressnell on SQL, published on Icetips (<http://www.icetips.com/showarticle.php?articleid=1>) . I will build on Dan's ideas to create a little utility for creating a Microsoft SQL 2000 database, including several tables, entirely from scratch.

Note: I have not tested the code with SQL server 7.0, but I can see no reason why it shouldn't work there, or with MSDE. In fact with some modifications you should be able to use this approach with just about any SQL database, as long as there are system tables.

How it works

The basic idea behind this utility is that all that Clarion needs to send commands/requests to the server is a “dumb” file in the dictionary with a number of fields large enough to contain the result of the biggest request you think you may have to execute. There is absolutely no need for this file to match anything actually existing in the database, except for the name of an existing table. As long as you have this file declared, (some developers called it this the “stupid temp table trick”) you can execute SQL statements on the server and get data back.

In order to achieve this goal as simply as possible, I will use two tools extensively. One, as I mentioned before, is Clarion’s PROP : SQL, which allows me to send commands directly to the server; the other is what I would name the “Queue from Queue”, or maybe more precisely the “Query from Queue” mechanism, which is a great idea from Dan Pressnell.

I will explain what this tool does in detail later on, but basically the Query from Queue mechanism does this:

- 1) It frees me from having to manually type (sometimes very lengthy) SELECT statements.
- 2) It returns the results of a query directly into a Clarion Queue, so that I can check this result set more than once, sort it different ways, check directly for a given value with a get, in other words all things not possible with the classic Clarion result set, which I can only read sequentially, and will lose on my next query.
- 3) It lets me use a column name in my code, as if I was using PROP : SQL on my dumb file. Ordinarily I would have to write something like
`name=dumbfile.f1` somewhere, then later `phone=dumbfile.f1`, which does not help much making my code readable. With the Query from Queue technique, I can write `name = QueryQueue.name`, which makes a little more sense.

But first, I need to back up just a little. I said I am going to write a utility to initialize a database from scratch. But how do I connect to a nonexistent database?

Fortunately, even without any operational database on the MS SQL server, a model/template for all other databases already exists, containing a certain number of system tables, plus many other things needed for the database engine to work properly, such as views, stored procedures, etc. This MS SQL database is named `master`.

So I want to connect to an MS SQL 2000 server, to the `master` database. I obviously need to know a user name and password with an administrator level to be able to create a database, whether I’m using Clarion or not. This being done, I want to create a database

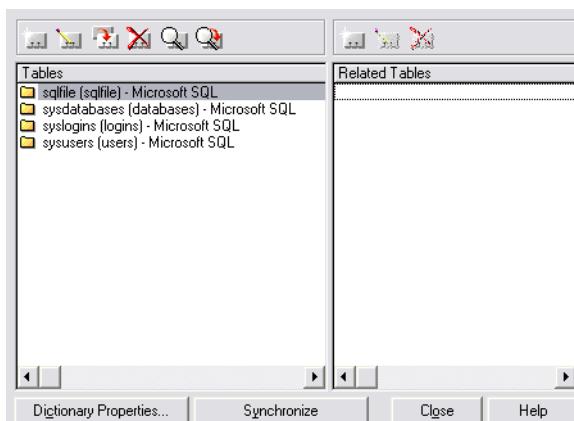
(without, if possible, overwriting an existing one with the same name), create a user who will be the owner of this base, create the tables I need, and check that I have done what I believed I was doing.

I must also say that I want this to be a simple process. I do not want to have to distribute an SQL script to my users which they need to run through some other utility; my users may have only MS Client installed on their computers, and nothing else.

The dictionary

To simplify my life a little, in this first version, I created a dictionary by importing a few tables from the master database (although it's possible to do without these tables). Figure 1 shows the dictionary.

Figure 1: The necessary tables



You can see that there are not so many tables needed:

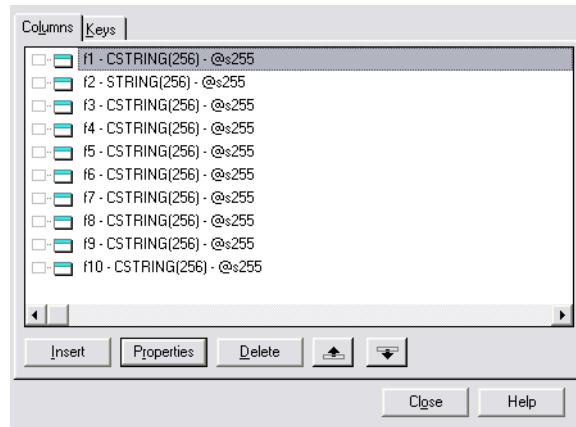
```
Sysdatabases  
Syslogins  
Sysusers
```

Sysdatabases contains the list of databases, Syslogins the list of roles (a role is a set of access rights grouped under a name), and Sysusers is the list of users, who can be associated with multiple roles.

MS SQL

The more interesting table is the one called `SqlFile`. Figure 2 shows the layout.

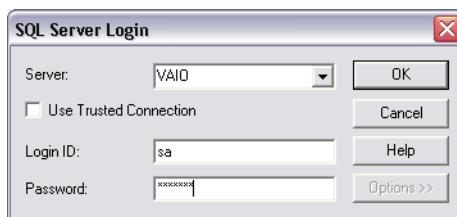
Figure 2: The `SqlFile` structure



`SQLFile` may look a little weird to you, even more so because the full path name for this table is `sysdatabases`, again! Clarion's "magic" data type conversion will allow me to use this table in a generic way, as a recipient for a *result set* (i.e. a set of records, using the specified columns) of an SQL request. There are only ten fields here, meaning any SQL statement cannot return more than ten columns of data, but nothing restrains you from creating more fields to accept result sets with as many fields as you like.

As this utility targets end-users, I have no idea what their server name could be, as well as their user name or password (if they had the good idea to change the default user name of `sa` to something a little less known and a bit more sophisticated), so there is no owner name specified anywhere in the dictionary. As a result, the application automatically opens the connection dialog box while you try to open any of the SQL tables.

Figure 3: The login dialog box



And that is a ready-made dialog I don't have to do anything about – it's all handled by the driver.

Send_Query

One of the requirements of this application is the ability to send an SQL command to the server. So that I don't have to write the same code over and over again, here's a function that sends the query and reports any error:

```
Send_Query      PROCEDURE  (string pQuery)
! Start of "Data Section"
! [Priority 5000]
! End of "Data Section"
CODE
! Start of "Processed Code"
! [Priority 5000]
open(sqlfile)
sqlfile{PROP:sql} = pQuery
if errorCODE()
    Message('Error:' & ErrorCode() & '-' & error() |
        & '<13,10,13,10>' & fileerror() & '<13,10,13,10>' & pQuery)
    Return 1
end
return 0
! End of "Processed Code"
```

I simply pass as parameters the SQL request I want to execute, do a `SqlFile{PROP:Sql}`, and test if I get an error. As a result, I now have a single line in the code to test for error, instead of six, and I am going to execute quite a few requests.

Query_from_Queue

Okay, it's time for a bit more serious stuff. Here, my goal is to have a request *generated* from a queue, and to have its result returned into the same queue. No no, don't start running away, it's not black magic and sorcery, even if it *is* very cool. All the credit goes to Dan Pressnell; I am just using the code and trying to explain. Here's the code for the `Query_From_Queue` procedure:

```
Query_From_Queue  PROCEDURE  (*Queue pQueue, string pExtra, |
    byte pDebug=0, long pLimit=0) ! Declare Procedure
! Start of "Data Section"
! [Priority 1000]
i    long
r1   any
r2   any
dstr &idynstr
! End of "Data Section"
```

MS SQL

```
CODE
! Start of "Processed Code"
! [Priority 5000]
dstr &= newdynstr()
dstr.cat('select ')
if pLimit <> 0
    dstr.cat('top ' & pLimit & ' ')
end
loop i=1 to 100000
    r1 &= what(pqueue,i)
    if r1 &= null
        break
    end
    if i > 1
        dstr.cat(',', '')
    end
    dstr.cat(who(pqueue, i))
end
dstr.cat(' ' & pextra)
X#=send_query(dstr.str())
IF pDebug
    Message ('QUERY='& dstr.str())
END
dstr.release()
free(pqueue)
loop
    next(sqlfile)
    if error() then break.
loop i=1 to 100000
    r1 &= what(pqueue, i)
    r2 &= what(sqlfile.record, i)
    if r2 &= null
        IF pDebug
            stop('Not enough fields in sqlfile to hold intended
number of columns!')
            halt
        ELSE
            break
        End
    end
    if r1 &= null or r2 &= null
        break
    end
    r1 = r2
end
add(pqueue)
end
```

Query_from_Queue receives a Queue as parameter, pQueue, a String, pextra, a flag to debug (which is an omittable parameter), and a limit indicator, omittable also. Here's the declaration for the queue:

```
DB_Q      Queue
DbID      like(databases:dbid), name('sysdatabases.dbid')
FileName   Like(databases:filename), name('sysdatabases.filename')
DB_NAME   Like(databases:name), name('DB_NAME(sysdatabases.dbid)
AS 'DB_NAME')
End
```

I'll explain the name('DB_NAME(sysdatabases.dbid) AS DB_NAME') syntax in a bit.

In MS SQL, the sysdatabases table is structured as follows:

```
CREATE TABLE [dbo].[sysdatabases] (
    [name] [sysname] NOT NULL ,
    [dbid] [smallint] NOT NULL ,
    [sid] [varbinary] (85) NULL ,
    [mode] [smallint] NOT NULL ,
    [status] [int] NOT NULL ,
    [status2] [int] NOT NULL ,
    [crdate] [datetime] NOT NULL ,
    [reserved] [datetime] NOT NULL ,
    [category] [int] NOT NULL ,
    [complevel] [tinyint] NOT NULL ,
    [filename] [nvarchar] (260) COLLATE SQL_Latin1_General_CI_AS
NOT NULL ,
    [version] AS
    (convert(smallint,databaseproperty([name],'version')))
) ON [PRIMARY]
```

And here is the SELECT statement I want Query_from_Queue to build:

```
SELECT sysdatabases.dbid, sysdatabases.filename,
DB_NAME(sysdatabases.dbid) AS DB_NAME
```

As you can see, the SELECT statement fields are the same as the fields declared in DB_Q. Let's see how Query_From_Queue builds the SELECT statement directly from this queue structure.

First, the code begins by initializing a dynamic string (the Dynstr interface) which will hold the SELECT statement. This is a string you can add to incrementally using its cat() (for concatenate) method. The first string added is 'select'. If a limit on the number of records to return is specified, the next string added is 'top ' & pLimit & ' '. Then comes the loop to construct the list of fields to return, using WHAT:

```
R1 &= what(pqueue,i)
```

The WHAT function returns a reference to the specified field number of a group, a queue or a record. Note r1 is declared as ANY, so that it can receive a reference to any data simple type. To say it differently, at the first loop iteration within the queue, R1 will receive the value DB_Q.DbID. If "what" returns nothing (null), the loop immediately breaks, as I reached the queue end. Using WHAT, here, has no other goal than letting the loop know when to break.

Note this code: If i > 1 ; dstr.cat(, ,); End. In other words, the comma is only added after the second iteration, since SELECT ,field1,field2 would result in a syntax error.

MS SQL

Next is the code `dstr.cat(who(pqueue, i))`. WHO returns a group's field name, from the position indicated by the second parameter. In the first iteration, `i` value is 1, and field's name is `sysdatabases.dbid`, so now the string contains the value `SELECT sysdatabases.dbid`.

In the second iteration `i > 1`, so the code adds a comma and second field's name; the value is now `SELECT sysdatabases.dbid, sysdatabases.filename`, and so on.

The loop exits after the third iteration (for this table), and pextra is appended. From the `Load_DBQ` routine in the main procedure you can see that pextra contains `FROM dbo.sysdatabases (result of Name(sysdatabase)) ORDER BY dbid`. Now the string's value is `SELECT sysdatabases.dbid, sysdatabases.filename, DB_NAME(sysdatabases.dbid) AS DB_NAME FROM dbo.sysdatabases ORDER BY dbid`.

Run this request in SQL Query Analyzer; you'll see something like the following:

1	C:\Program Files\Microsoft SQL Server\MSSQL\data\master.mdf	master
2	C:\Program Files\Microsoft SQL Server\MSSQL\data\tempdb.mdf	tempdb
3	C:\Program Files\Microsoft SQL Server\MSSQL\data\model.mdf	model
4	C:\Program Files\Microsoft SQL Server\MSSQL\data\msdbdata.mdf	msdb
5	C:\Program Files\Microsoft SQL Server\MSSQL\data\pubs.mdf	pubs
6	C:\Program Files\Microsoft SQL Server\MSSQL\data\northwnd.mdf	Northwind

By the way, `DB_NAME` is an MS SQL function which returns the database name from the database ID passed as parameter. In this case I really don't need to do that, as a field in `sysdatabases` contain this data, but it makes the data more readable.

Next, the code reads the “dumb” `SqlFile` table in a loop, and it starts again in a new loop very similar to the first one. Only this time, it copies the label from the `Queue` to `r1`, and the label from `sqlfile` to `r2`. It then tests for an error to exit the loop, and if there is no error it ends up with `r1 = r2`, or `DB_Q.DbID = Sqlfile:f1`, for the first queue field. After the break, the code does an `add(pqueue)`. Et voila!. This code is actually filling in the queue with the results matching the request generated by this very queue.

This explanation was a bit lengthy, but it's not very often that you see a queue used to generate an SQL request *and* containing the result of the request.

Now I have the tools for the job, and a pretty good idea on how to use them, it's time to do some real work.

Main procedure

To be certain that the database I am going to create does not exist already on the server, the first thing to do is to load a queue with all existing databases on this particular server. MS SQL has a table, `Sysdatabase`, with this information `Load_DbQ` routine is doing that job.

Next, the code finds the default path to create the database “physical files”. In order to achieve this, it will simply look where the master database is installed, and filter the result to keep only the directory name.

The code then checks that the current user has the proper rights required to create a database on the server. The simple SQL SELECT `IS_SRVROLEMEMBER('sysadmin')` AS Value does all the work! `IS_SRVROLEMEMBER` is one of the very numerous stored procedures already in MS SQL (to get ideas about how to use these stored procedures, read the online books). The result of this request is 1 or 0, `true` or `false`, and is stored in a local variable. If the current user does not have the required rights, she is politely told so when the window opens.

The next test is to verify that the database does not exist already. This is very simple to implement, as the queue of the existing databases is already there. A simple GET on the queue determines if the database exists. If no error, there's a problem!

Now it's time to create the new database, using an SQL command built up from variables. In this example there are a certain number of parameters hard coded, but everything could be a variable as well. Read Microsoft documentation on the possible variants for `CREATE DATABASE`. Just remember that in my specifications, I wanted something simple for the end user.

It's always good to check that the database has actually been created. The code again loads the database list with, but this time, there should *not* be an error on the GET!

Now it's time to connect to the newly created database (`USE database`). The code creates a special user who owns the tables, and gives that user a certain number of privileges. Here again, the best thing is to dig into Microsoft documentation, as your needs might be very different than the one taken into account for this example. Since this user will have `CREATE` privileges, the code can now create every table, eventually with indexes

and keys. As in this example, it might be a good idea to use an ASCII File to log what happens, and display this log to the user.

Summary

With Clarion's PROP:SQL it's easy to execute SQL statements on MS SQL Server, or just about any other SQL server. And Dan Pressnell's *Query_from_Queue* procedure is a great tool for building SQL statements and retrieving the results of those statements into a queue.

This paper's goal is really not to explain Microsoft's own version of SQL syntax, but simply to show that it is totally possible to use all these commands from Clarion. And remember that with PROP:SQL you are not limited to existing tables, it's even possible to use dynamics result sets (not existing statically in tables, but dynamically calculated at request's time).

Source code

See "Appendix A: Getting Support," p. 601, for information on how to get the source accompanying this book.

- v5n12sqlutilities.zip

GENERATING MS SQL SERVER SIDE TRIGGERS

by Ayo Ogundahunsi

Clarion 6 comes with a lot of new features, amongst which is the capability to integrate client-side triggers as part of the design of the database (see “Using Client-Side Triggers In Clarion 6,” p. 543).

I must say this new feature is quite ingenious, and it is a very bold step. Nevertheless, without downplaying its usefulness, I have always preferred the traditional way of using server-side triggers, because of their inherent advantages. These include:

- 1) Ability to monitor and audit updates no matter how the database is modified.**
This means even if a table is updated outside of your Clarion application, you can still audit who made this change. In today’s world, scalability and interoperability among diverse components is usually preferred, hence, if you wrap your audit logic within Clarion, you will have to maintain at least a second code base if an external system is going to be updating your database.
- 2) Centralized management of data integrity, and in some cases referential integrity.**

What are Triggers?

The purpose of this article is not to give a detailed exposition on triggers, but rather to present a useful template that can help automate auditing of user updates to your database. Auditing means creating a log of who made what change to a table, and when the change happened. This template automates the process of creating both the audit table(s), and the triggers which create the audit data.

Since there is extensive documentation in books, or on the internet on Triggers, I will not explain in great detail what triggers are or the kind of triggers available for use.

If you're unfamiliar with triggers, here is a definition from SQL Server's Books Online:

Microsoft SQL Server 2000 triggers are a special class of stored procedures defined to execute automatically when an UPDATE, INSERT, or DELETE statement is issued against a table or view.

Triggers are powerful tools that sites can use to enforce their business rules automatically when data is modified. Triggers can extend the integrity checking logic of SQL Server constraints, defaults, and rules, although constraints and defaults should be used instead whenever they provide all the needed functionality.

Tables can have multiple triggers. The CREATE TRIGGER statement can be defined with the FOR UPDATE, FOR INSERT, or FOR DELETE clauses to target a trigger to a specific class of data modification actions. When FOR UPDATE is specified, the IF UPDATE (column_name) clause can be used to target a trigger to updates affecting a particular column.

For further reading on Triggers, you can check the following resources:

- MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/tsqlref/ts_create2_7eq.asp
- Microsoft Support site:
<http://support.microsoft.com/default.aspx?scid=%2Fservicedesk%2Fwebcasts%2Fen%2Fwc011601%2Fwct011601.asp>
- DevBuilder:
http://www.devbuilder.org/asp/dev_article.asp?aspid=16

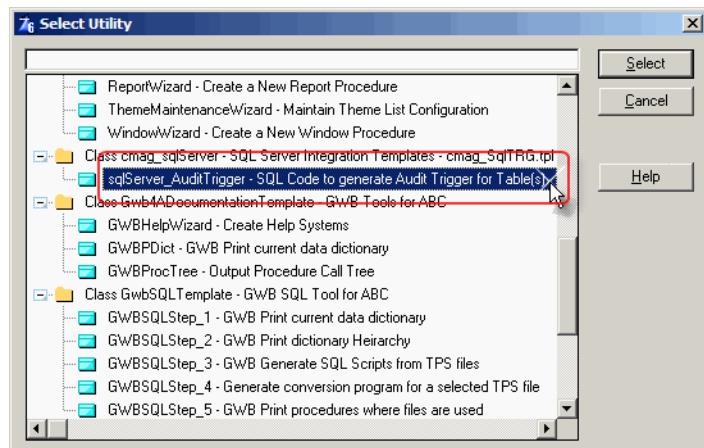
The Template

This template, which was inspired by ideas on auditing with triggers in one of Ken Henderson's books, will take tables defined in a Clarion Dictionary and generate matching trigger scripts.

The template file is cmag_SQLTRG.tpl. You have to register it the same way you register all other templates. Since the template code is contained in a single file, you do not need to update paths in your .RED file if you choose to put the template somewhere other than the templates directory.

The template is a Utility template and to use it, you need to have an application loaded. Then choose **Application|Template Utility** from the Clarion main menu, or press **Ctrl-U**. Choose the `sqlServer_AuditTrigger` template, as shown in Figure 1.

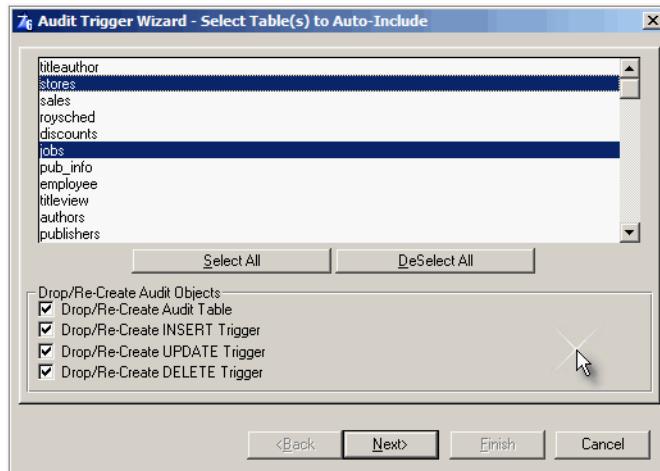
Figure 1: Selecting the template



MS SQL

Once you select the template, you will be presented with a wizard, as shown in Figure 2. The tables contained in the dictionary for the application are automatically made available for selection.

Figure 2: Selecting Tables to Audit



Tables to Audit

This form on the wizard allows you to select the table(s) to create audit triggers for. You can generate all the trigger code into a single file, or individual files that reflect the name of the tables.

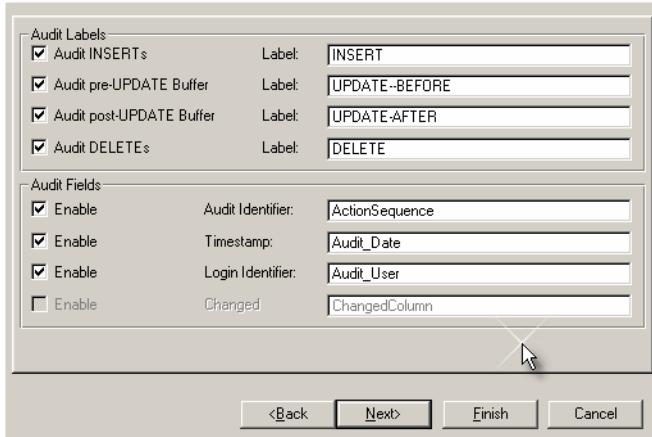
Drop/Re-CREATE Audit Objects

The **Drop/Re-create** checkboxes simply allows you generate SQL code that will always DROP and CREATE the trigger whenever you run the generated script.

Audit Labels

The Audit label fields allow you to define what entry is recorded in the audit table:

Figure 3: Audit Labels & Fields



As shown in Figure 4, whatever text is typed for the label will be stored accordingly under the `ActionSequence` column.

Figure 4: Representation of Audit Labels

Data in Table 'JOBS_AUDIT' in 'pubs' on 'AYO-LAPTOP\AYO_NV01'						
<pre>SELECT * FROM JOBS_AUDIT</pre>						
job_id	job_desc	min_lv	max_lv	ActionSequence	Audit_Date	Audit_User
3	Publisher	150	250	UPDATE-BEFORE	4/17/2004 6:22:13	AYO-LAPTOP\Ayod
5	Publisher - ClarionMag	150	250	UPDATE-AFTER	4/17/2004 6:22:13	AYO-LAPTOP\Ayod
15	this is a test	22	50	INSERT	4/17/2004 9:05:04	AYO-LAPTOP\Ayod
15	this is a test	22	50	DELETE	4/17/2004 9:06:28	AYO-LAPTOP\Ayod
*						

Audit Fields

The concept of the template is based on the fact that you should be able to take a snapshot of a record buffer (or row) before an update, or save a record buffer (or row) that was used to add a new record (row) to your table.

As a result of this you need to have an audit table with a structure similar to the table you are auditing, however, with three extra fields. These fields are:

MS SQL

- 1) **Audit Identifier** – Here you store a text that describes the update action you just performed which you are auditing.
- 2) **Timestamp** – Records the date and time the update action was performed. This is updated with the Transact-SQL GETDATE () function.
- 3) **Login Identifier** – Records the user that performed the update action. This is updated by the Transact-SQL SUSER_SNAME () function.

Depending on whether you enable any of these fields (using the **Enable** checkbox), the script generated may or may not provide code to add them to the audit table.

For example, if you want the **Timestamp** field to be added to your audit table, this is the kind of code the template generates:

```
-- Add 'Audit_Date' column to structure if it doesn't exist already
IF NOT EXISTS
    (SELECT COLUMN_NAME FROM #TEMP_AUDIT WHERE COLUMN_NAME =
'Audit Date'
    AND TABLE_NAME = 'JOBS')
BEGIN
    ALTER TABLE dbo.JOBS_AUDIT ADD Audit_Date DATETIME
END
GO
```

#TEMP_AUDIT table is a table containing a list of fields that belong to an audit table.

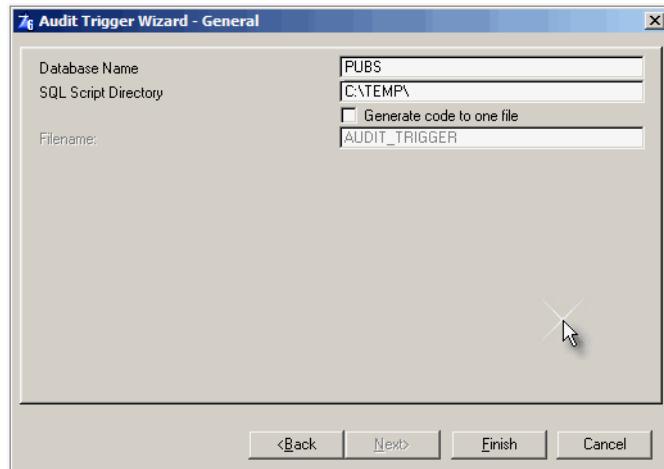
You can also change the column names to whatever you desire. For example, you can choose to change 'AuditSequence' to 'UpdateAction'.

Generating MS SQL Server Side Triggers

Script Destination

You can define where the generated script is output to. If you want you can also generate code for multiple tables into a single file, this way it is easier to apply the script once.

Figure 5: Script Destination



Running the script

It is easy to run the created script. From your Query Analyzer, select the database that contains your tables, and run the script from there.

Figure 6: Executing the generated script

The screenshot shows a Microsoft SQL Server Management Studio (Query Analyzer) window. The title bar reads "Query - AYO-LAPTOP\AYO_NV01.pubs.AYO-LAPTOP\Ayodele Ogundahusi - C:\Temp\...". The main pane displays the following T-SQL script:

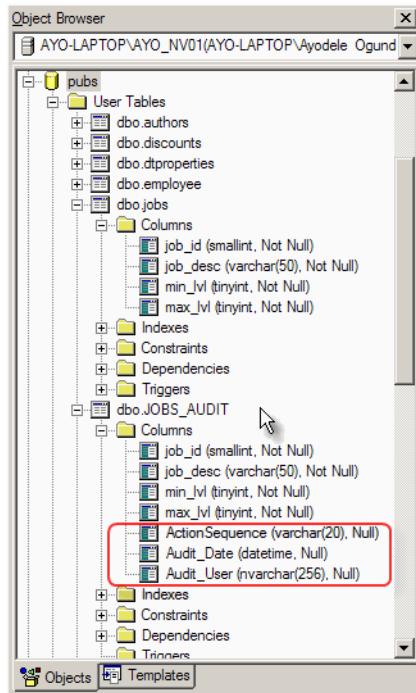
```
*****  
dbo.JOBSS  
*****  
-- Create a list of existing columns in JOBS/dbo.JOBSS  
SELECT COLUMN_NAME, DATA_TYPE, CHARACTER_MAXIMUM_LENGTH, NUMERIC_PRECISION  
FROM INFORMATION_SCHEMA.COLUMNS WHERE TABLE_NAME = 'JOBS'  
GO  
  
-- Check for JOBS_AUDIT, if it exists, DROP it.  
IF (OBJECT_ID('dbo.JOBSS_AUDIT') IS NOT NULL) AND (OBJECTPROPERTY(OBJECT_ID  
DROP TABLE dbo.JOBSS_AUDIT  
  
(4 row(s) affected)  
  
(0 row(s) affected)
```

The status bar at the bottom shows the following information: "Qu AYO-LAPTOP\AYO_NV01 (8.0)" "AYO-LAPTOP\Ayodele Ogundahusi" "pubs" "0:00:03" "0 rows" "Ln 1, Col 1".

Generating MS SQL Server Side Triggers

Once you do this you will see the Audit table created as shown in Figure 7. ActionSequence, Audit_Date, and Audit_User are the three extra fields added to the audit table for JOBS (i.e. JOBS_AUDIT which has a similar structure).

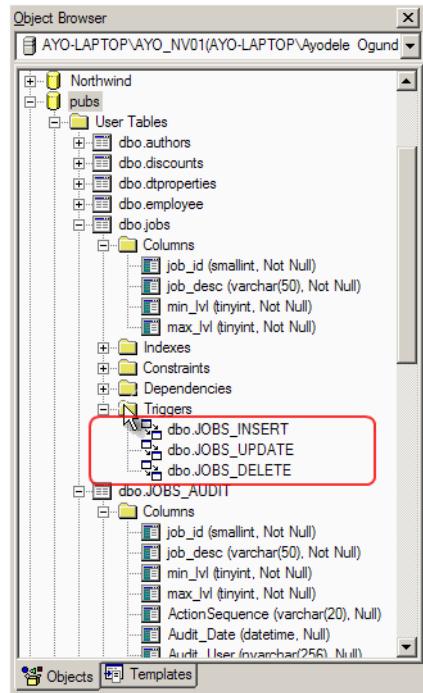
Figure 7: Audit fields in audit table



MS SQL

Figure 8 shows the created triggers. JOBS_INSERT, JOBS_UPDATE, JOBS_DELETE are the three triggers added to the JOBS table.

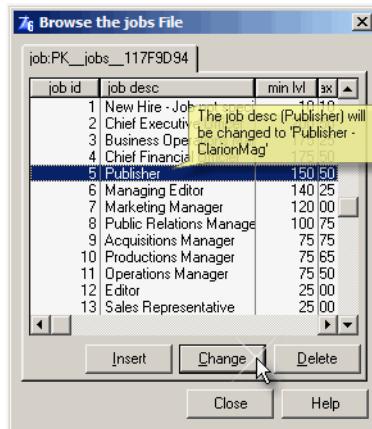
Figure 8: Generated Triggers



How it works

To demonstrate how the applied script works, I've imported the PUBS database, that comes with SQL Server, into Clarion and I've generated an application. I will be using the JOBS table.

Figure 9: Jobs browse (before update)



Next I load the jobs browse and change the item in the list box with `job_id = 5` from 'Publisher' to 'Publisher - ClarionMag'.

Figure 10: Jobs browse (after update)



MS SQL

I open the table JOBS_AUDIT, and I can see the audit records that have been created as shown in Figure 11.

Figure 11: Audit table results (update from Clarion app)

The screenshot shows the SQL Server Enterprise Manager interface with a query window titled "SQL Server Enterprise Manager - [Data in Table 'JOBS_AUDIT' in 'pubs' on 'AYO-LAPTOP\AYo...'". The query is "SELECT * FROM JOBS_AUDIT". The results grid displays two rows of data:

job_id	job_desc	min_lvl	max_lvl	ActionSequence	Audit_Date	Audit_User
5	Publisher	150	250	UPDATE-BEFORE	4/17/2004 6:22:13	AYO-LAPTOP\Ayo
5	Publisher - ClarionMag	150	250	UPDATE-AFTER	4/17/2004 6:22:13	AYO-LAPTOP\Ayo

The beauty about maintaining server side triggers is the fact that no matter the application that updates the JOBS table, even if you make a change from the enterprise manager, it logs the action.

In Figure 12, I change the value in the min_lvl column of the JOBS table for the Marketing Manager from 120 to 125 from within the Enterprise Manager.

Figure 12: Updating jobs table from Enterprise Manager

The screenshot shows the SQL Server Enterprise Manager interface with a query window titled "Data in Table 'jobs' in 'pubs' on 'AYO-LAPTOP\AYO_NV01'". The query is "SELECT * FROM jobs". The results grid displays 13 rows of data. In the 7th row, the "min_lvl" column for the "Marketing Manager" job is currently set to 120, which is highlighted with a red border. This indicates that the value has been modified from its original value of 120.

job_id	job_desc	min_lvl	max_lvl
1	New Hire - Job not specified	10	10
2	Chief Executive Officer	200	250
3	Business Operations Manager	175	225
4	Chief Financial Officer	175	250
5	Publisher - ClarionMag	150	250
6	Managing Editor	140	225
7	Marketing Manager	120	200
8	Public Relations Manager	100	175
9	Acquisitions Manager	75	175
10	Productions Manager	75	165
11	Operations Manager	75	150
12	Editor	25	100
13	Sales Representative	25	100

After doing this, inspection of the JOBS_AUDIT table shows the content of the row before the update (UPDATE--BEFORE), and the new contents of the row (UPDATE-AFTER) as shown in Figure 13.

Figure 13: Audit table results (update from Enterprise Manager)

job_id	job_desc	min_lvl	max_lvl	ActionSequence	Audit_Date	Audit_User
5	Publisher	150	250	UPDATE-BEFORE	4/17/2004 6:22:13	AYO-LAPTOP\Ayc
5	Publisher - ClarionMag	150	250	UPDATE-AFTER	4/17/2004 6:22:13	AYO-LAPTOP\Ayc
7	Marketing Manager	120	200	UPDATE-BEFORE	4/19/2004 4:38:16	AYO-LAPTOP\Ayc
7	Marketing Manager	125	200	UPDATE-AFTER	4/19/2004 4:38:16	AYO-LAPTOP\Ayc

Summary

Although this template makes it easy to use triggers for an audit trail, you need to be aware of potential performance issues. Triggers are actually stored procedures that execute whenever an INSERT, UPDATE, or DELETE happens. If you are doing batch operations on a file, the trigger will execute for each row that is changed, and that could slow down processing. For example, if you have a table you import a large amount of data into, this table should not have the audit trigger created for it.

Having said this, it is up to you to test and determine the effect of using the audit trigger code generated by this template in your peculiar environment.

Source code

See “Appendix A: Getting Support,” p. 601, for information on how to get the source accompanying this book.

- v6n04sstriggers.zip

MS SQL

DATE FILTERING WITH MS SQL

by John Griffiths

When writing business software I find I often need to mess with dates and date ranges in my MS-SQL queries. Sometimes I need to use a temptable to grab some data from SQL on a form or report, or I need to adjust the FILTER conditions for a browse.

Working within the Clarion source with Clarion language statements, and translating the FILTER to that needed by the SQL engine, can become more complex than is really necessary.

Here is an example of a WHERE clause I wrote about four years ago:

```
sqlWHERE = ' WHERE (h.dtact_date >= ''' & format(FromDate,@D02) &
00:00:00''' &|
' AND h.dtact_date < ''' & format((ToDate + 1),@D02) &|
00:00:00''' &|
' AND h.histype = ''V''' '
```

I know I could have used the SQL BETWEEN verb. I also know by looking at the execution plan that MS-SQL simply converts the BETWEEN verb it to a \geq AND \leq construct. I am trying to reduce the load at the server, and so I code using the expanded construct.

MS SQL

The above works great, but I figured MS-SQL is storing the DATETIME values as pure numerics, and formats the output to a STRING date representation of that numeric or formats an input STRING representation of a date to a numeric, depending on the request you are sending.

I am expending programming effort and my Clarion program is using up clock cycles converting numbers to strings, strings to numbers, and my SQL server is doing likewise.

About a year ago, I changed my approach and started adding an EQUATE into my Clarion modules:

```
DATE1900 EQUATE(36163) ! the MS-SQL date num for Jan 1 1900 = 0
```

This equate allows me to easily communicate with MS-SQL simply using numerics. I now build query strings equivalent to the SQL WHERE string above as follows:

```
sqlWHERE = ' WHERE (h.dtact_date >= ' & (FromDate - DATE1900) & |
    ' AND h.dtact_date < ' & (ToDate - DATE1900 + 1) & |
    ' AND h.histype = ''V'''
```

This works very well. Why? Look at your “SQL Server Books Online.” In the CAST CONVERT page you will see a datatype-to-datatype conversion matrix. Note there the implicit conversions that can take place between DATETIME and NUMERIC types.

In testing I found that I could scrip a field in a new table as:

```
DtBaseDate DateTime default 37000.9999999 ,
```

or:

```
DtBaseDate DateTime default CONVERT(INT,GetDate()) , -- 00:00 time
today
```

Here is a small example where I am running a query with a TEMPFILE.I am using a TEMPFILE named tfile, with the first field named t1 and seeking the MAXimum date in the dtAct_Date column:

```
1   OrigDate = 0      ! a Clarion LONG
2   open(tfile)
3   SQLstr2 = 'SELECT convert(int,MAX(h.dtdate)) from History AS h '
4   tfile{prop:sql}=SQLstr2
5   next(tfile)
6   if errorcode()
7       message(FileError())
8   else
9   OrigDate = tfile.f1 + Date1900      ! adjust to a ClarionDate
LONG
10 end
11 close(tfile)
```

At line 3) I need to tell SQL to send the data back as a numeric, or it will return a date string formatted according to default settings.

Line (9) adjusts the integer field obtained from SQL by adding my EQUATE and then assigns the result to my Clarion variable (a LONG).

Had I been getting back a DATETIME field as a string using a line (3) like this:

```
SQLstr2 = 'SELECT CONVERT(DATETIME,MAX(h.dtact_date),112 )  
          from History AS h '
```

I would have received a STRING formatted as YYYYMMDD and then would need to convert this to a numeric and store in my Clarion variable, something like:

```
OrigDate = DEFORMAT( tfile.f1, @d12)
```

I hope this has given you the incentive to experiment with how you work between Clarion and MS-SQL. As MS-SQL is the only SQL engine I have worked with, I cannot vouch that this method will work with other SQL servers. You may have some testing to do.

MS SQL

Book Reviews

BOOK REVIEW: POSTGRESQL DEVELOPER'S HANDBOOK

By Ewald Geschwinde and Hans-Jergen Schenig

Published by Sams, December 2001

ISBN: 0672322609

768 pages,

\$44.99 US

\$67.95 CA

£32.99 UK

Reviewed by David Harms

PostgreSQL Developer's Handbook, at some 750 pages, is one of the longer and more detailed books on PostgreSQL currently available. Not that there are a lot to choose from - a search on Amazon for PostgreSQL books yielded 11 hits to arch rival MySQL's 49.

The lesser part of this book is taken up with the nuts and bolts of common SQL operations as defined by PostgreSQL. Chapter one covers basic concepts, and chapter two installation on Unix/Linux and on Windows (the latter using CygWin - ugh! - happily a native

Book Reviews

Windows version is expected – see “Getting Started With PostgreSQL,” p. 273). Chapter three is a longish one dealing with the basic SQL commands for creating databases, tables, and other components, adding and modifying data, and retrieving data. There’s a lot of fairly detailed information here, including topics such as self joins, casting data types, and working with arrays, BLOBS, and other special data types. There’s even a section on modeling databases.

After a very brief treatment of transactions, Handbook goes on to a topic that I’ve found of considerable personal interest lately: the PL/PGSQL programming language. Unfortunately this is also a short chapter - if you get into writing stored procedures in a big way, you’ll probably spend more time with the standard docs. Database administration, backup and recover, and performance tuning all get their own chapters, none extensive, but generally sufficient. The performance tuning chapter deals mainly with database design issues (when and how to create indexes, optimizing queries, using EXPLAIN and VACUUM), and touches on system issues such as file systems and buffers.

There are some long, long chapters in this book. The granddaddy of them all is Programming Interfaces, which spans almost 200 pages (a book in itself!) and provides an overview of using PostgreSQL from a half-dozen or so languages, and with ODBC. Only the latter is likely to be of interest to most Clarion developers, so there’s a good chunk of this book you probably won’t care about. The last chapter in Part 1 summarizes software add-ons for PostgreSQL, such as a cube datatype (you can, for instance, select the union of two cubes), full text indexes, ISBN/ISSN numbers, a utility for dumping large objects, a soundex module, and more.

If you’ve read this far through the book you’re about two-thirds done; the last third is devoted to real-world examples. I won’t go into all the details, but the topics include: working with EBCDIC data; multidimensional data structures; classifying and aggregating data; generating Flash content with PHP and PostgreSQL; running the PostgreSQL regression tests (mainly useful to the PostgreSQL developers, but you may want to do this to ensure that the latest beta works on your system); extending PostgreSQL with C functions; creating custom datatypes; creating operators (for instance, overloading the + operator to allow for adding RGB color values to get a new color); writing new rules (PostgreSQL is a highly configurable database); handling date and time calculations; persistent database connections (using PHP); using ODBC; and finally, graphing PostgreSQL data using gnuplot.

The PostgreSQL Developers Handbook is a heavy volume with a lot of information on an incredible variety of subjects. It may not tell you everything you ever want to know about the SQL in PostgreSQL (although it will come reasonably close), but it will open your eyes to the great breadth of functionality available in this popular open source database.

BOOK REVIEW: SQL TUNING

By Dan Tow O'Reilly, November 2003

<http://www.oreilly.com/catalog/sqltuning/index.html>

ISBN: 0-596-00573-3

336 pages

\$39.95 US

\$61.95 CA

£28.50 UK

Reviewed by David Harms

The last time I polled ClarionMag readers about how much of their time was spent developing for SQL databases, over half said 25% or more, and 18% said they developed only for SQL. Now, developing applications that use SQL databases is one thing; making sure those applications make efficient use of the SQL server is another.

Like many developers who use SQL, I started off simply by creating a Clarion application that used an SQL database (MySQL, in my case). I let the ABC templates/classes create and manage the underlying views, which through the ODBC driver created the SQL statements. But soon enough I wanted to do some things with the data that ABC didn't

know how to do. So I started writing SQL statements, and executing them with PROP:SQL, or via the MySQL client program. I soon had a library of SQL statements, some of which worked well, and others which took horrendously long to complete. I'm not a DBA, but clearly I'd learned enough about SQL to become dangerous. Eventually, by trial and error (and by upgrading to a more recent version of MySQL), I solved most of those problems, or at least made the execution of those queries tolerable. But I can't say I really learned how to optimize my SQL statements.

Optimizing SQL is really what this book is about. It is *not* about how to optimize an SQL server. That is, it won't tell you what settings the server needs to run any given SQL statements as quickly as possible; instead it provides a methodology for optimizing those statements so they load the server as little as possible. This is an important distinction — server tuning is important, but it's no substitute for SQL statement tuning. And although the book focuses on DB2, Oracle, and SQL Server, the basic concepts, as well as the methodology, described here can be applied to SQL servers in general.

The author, Dan Tow, describes three key aspects of SQL tuning. The first, is to find and interpret the *execution plan* of the SQL statement. This is information, which the server will provide on request, explaining how it processes a particular query. The execution plan includes information such as which tables are processed in which order, and which keys (if any) are used. The second is to modify the SQL statement so that it uses the best possible execution plan. The third, and most important, aspect is knowing how to decide which is the best execution plan. Many developers make this decision intuitively, or by trial and error, using best guesses; Tow, however, provides a methodology, complete with diagramming technique, for finding the best execution plan.

This is by no means a beginner SQL book, but Tow does spend a chapter on data access basics, from a behind-the-scenes perspective. When he explains tables, for instance, he points out the importance of knowing how the physical layout of the tables affects read performance. When he discusses indexes, he explains how B-trees work, and looks at more exotic solutions such as index-organized tables and bitmapped indexes. And of particular interest, he describes how SQL servers go about processing JOIN statements.

This chapter is followed by the requisite two chapters covering the topics of obtaining and understanding execution plans (with full examples for each of the three subject databases), and of modifying SQL statements to achieve a different execution plan. These modifications can force the server to use or ignore specific indexes, use or prevent join orders, and control the order in which outer queries and subqueries are executed. Again, there are specific examples for DB2, Oracle, and SQL Server.

Most of the rest of the book deals with Tow's tuning methodology, which uses a diagramming notation that focuses specifically on those parts of a query which really do affect performance, including the proportion of records returned from any table in a join (the *filter ratio*), and average number of records found on each side of the join (the join

ratio). The diagram ignores those things which have little or no affect, or which cannot effectively be optimized anyway (columns selected, ORDER BY and GROUP BY clauses, table names, details of join conditions, absolute table size, etc.).

The diagramming notation itself is remarkably simple – the hardest part (which is not that hard) is getting the data on join ratios and filter ratios. Once you have that you can lay out the diagram, and once you've laid out the diagram you follow a few simple rules to deduce the best execution plan. Tow's examples range from simple two-table joins to monsters of 17 tables or more. And he does have experience with massive SQL statements – he writes that he routinely tunes joins with more than 40 tables, and his personal record is 115 tables.

The chapter on tuning complex statements (and no, a 17 table join isn't necessarily complex, at least according to Tow) is also a lesson in database design. When a SQL statement becomes too complex to diagram quickly, often the reason is a flaw in the database itself, and usually in the design of the application that uses the database. Possible flaws include cyclic joins, two disconnected queries combined in a single query, query diagrams with multiple roots, joins with no primary key, and problematic one-to-one joins, among others. There is also a brief chapter on why the diagramming method works, and several final chapters on special cases and solutions to “unsolvable” problems.

As evidenced by other reviews I've read of this book, Dan Tow is clearly an individual with a keen understanding of how SQL servers work, and how they can be made to work most efficiently. His diagramming notation is simple and effective, and his methodology highly effective. If you've been using SQL and you're starting to bump up against performance issues, or if you'd like to make your SQL applications more robust, or if you're just curious about how SQL servers handle multi-table queries, then you need to read this book.

Book Reviews

BOOK REVIEW: SQL IN A NUTSHELL

By Kevin Kline, with Daniel Kline, Ph.D.

Published by O'Reilly, December 2000

<http://www.oreilly.com/catalog/sqlnut/>

ISBN 1-56592-744-3

224 pages,

\$29.95 US

\$43.95 CA

£20.95 UK

Reviewed by David Harms

O'Reilly's SQL In A Nutshell is a desktop reference for SQL as implemented in SQL Server, MySQL, Oracle and PostgreSQL. This is a slightly older work, first published in December 2000.

This comparison of four of the most popular SQL databases (notable exceptions being Sybase, DB2, and Interbase/Firebird) is instructive in two ways. First, it points out the startling lack of real standards in the SQL world, and second, it offers practical information for anyone wishing to develop applications which can be easily ported to a variety of

Book Reviews

databases. This is not, however, a book for someone who doesn't already know a bit about SQL. It's a reference, not a tutorial.

The touchstone for this book's comparisons is the SQL99 standard. As the authors point out, SQL database vendors typically have data types that correspond to most of the SQL99 data types, but often use different naming conventions. These data types are listed in tables in the book; a cross reference showing each vendor's version of a given SQL99 type would be a useful addition.

The bulk of the book is taken up with an SQL command reference. Each of the SQL99 commands is discussed in detail, and here this is also a cross-reference table listing each vendor's implementation (or lack thereof). Most of this information is also available online, but not in such an organized fashion. If your concern is database portability, then you will want to compare statement syntax yourself, and not rely entirely on the descriptive text.

For instance, all four databases support the syntax `ALTER TABLE tablename ADD [COLUMN] ...`. Oracle's syntax for changing a column, however, is `ALTER TABLE tablename MODIFY [COLUMN] ...` while for the other three it's `ALTER TABLE tablename ALTER [COLUMN] ...`. PostgreSQL is the only one of the four databases that doesn't support `ALTER TABLE tablename DROP [COLUMN]` (as of the book's publication, that is - the ability to drop columns was added in PostgreSQL 7.3). None of this is mentioned in the text, although to be fair if every difference between databases were discussed in detail, this would be a book series, not a single volume. The point is that you'll still have to do some studying if you want to be able to seamlessly install and run your apps on different SQL databases. The authors do point out many differences between vendors in the text - just don't expect these discussions to be exhaustive. There are also useful descriptions of where the vendors depart from the SQL99 specification.

Some SQL commands get fairly length treatment, including `CREATE TABLE`, `GRANT`, and `SELECT`. And keep an eye out everywhere for those owl and turkey images in the text - these identify, respectively, helpful information and potential problems which deserve your special attention.

The last two chapters cover SQL functions (built-in, not user-defined) and unimplemented SQL99 commands. Again, what this book (perhaps unintentionally) points out is the absolutely pitiful state of SQL standardization.

Although the information in this book is slightly dated, that may not be a bad thing for anyone wishing to develop for multiple back ends, since it's unlikely that all potential clients will be running the latest version of their respective database of choice. Or it may help you decide which database will best serve your needs, and save you the grief of trying to satisfy the disparate requirements of each database vendor.

BOOK REVIEW: MANAGING & USING MYSQL

By George Reese, Randy Jay Yarger, Tim King With Hugh E. Williams

Published by O'Reilly, April 2002

<http://www.oreilly.com/catalog/msql2/index.html>

ISBN: 0-596-00211-4

424 pages

\$39.95 US

\$61.95 CA

£28.50 UK

Reviewed by David Harms

Managing & Using MySQL is a well-structured, well-written book that serves as a solid introduction to both MySQL and SQL databases in general. It also provides an introduction to some of the numerous development environments which work with MySQL, such as PHP, Perl, and Java. Curiously absent is any significant information about Windows development and ODBC, but Clarion developers, at least those new to SQL, will still find much here of interest, as will Clarion developers who dabble in some of the other languages the book discusses.

Managing & Using MySQL is divided into four parts. The first part covers MySQL and SQL basics, installation, and database administration. As with the rest of the book, this part discusses MySQL really only in reference to itself. Absent (aside from a few brief mentions in the opening chapter) are any comparisons between MySQL and, say, Oracle, MS SQL, or PostgreSQL. The authors are also relatively uncritical of MySQL's feature set as compared to more "mainstream" SQL databases. For instance, foreign keys are dismissed with the statement "Applications themselves should generally worry about foreign key integrity." Happily, that's something Clarion is quite able to do; still, if you're looking for a book to help you choose between databases, this isn't the one. What you will get (at least to begin with) is a very solid introduction to SQL basics and good instructions on MySQL installation and setup.

Part II has chapters on performance tuning, security, and database design. I particularly appreciated the discussion of the EXPLAIN SELECT command, which you can use to find out what steps the server is actually taking when it processes a SELECT. This information can be critical to improving performance. I also was very impressed with the chapter on relational database design; I found it a concise, readable introduction to the topic.

Part II ends at page 134 of some 400 pages. Part III, which is about as long as parts I and II combined, is primarily made up of chapters introducing MySQL development with Perl, Python, PHP, C, and Java. This is in keeping with the introductory nature of the book, and some chapters will be of interest to some Clarion developers, but I couldn't help but feel a bit let down; I really wanted this book to continue on to some advanced topics. On to Part IV.

The final part (a.k.a. the last third, by volume) of this book is reference material: SQL syntax; MySQL data types; operators and functions; PHP API reference, C reference, and the Python DB-API. There's some value added here, but I think this part of the book is a bit long for what is still mainly a regurgitation of information available online.

Note: This book covers MySQL 4.01; version 4.1, which adds some significant features such as subselects, is now in alpha release.

If I sound a bit critical of this book, it's only because it started so well, and I came to expect so much. Although half the book may not be of interest to most Clarion developers, I think it's still worth the price of admission, particularly for those just starting out with SQL and MySQL.

ABC Database Class Design Notes

INSIDE ABC: FIELDPAIRSCCLASS AND BUFFEREDPAIRSCCLASS

by David Bayliss

When I sent around the initial design proposal for what later became the ABC system one of the claimed benefits was a code reduction in user procedures of around half to two thirds. At the time this was viewed with some skepticism and so we set 30 percent as a reasonable goal. In the end we actually achieved around 92-94 percent, and the field class was chiefly responsible for the extra 30 percent.

To appreciate why, you need to consider how certain parts of the ABC system would be coded if the FieldPairsClass didn't exist. For the sake of concreteness I am going to use the example I used many moons ago when I was trying to persuade Tom Moseley that OOP could work in the templates.

Example 1 : Updating a link field

One constant bugbear in CW2 was overflow of the *appname_ru* (referential integrity, or RI) module (to >64K) on any sizeable or complex dictionary. One aim of OOP was to

reduce this problem. Although there were other procedures the heart of the referential integrity can be shown by pseudo-code for the RI update function.

Listing 1 assumes F1 and F2 are the related files (on the keys F1:K & F2:K with two and three components respectively, KC1, KC2, KC3 etc).

Listing 1. Code to cascade RI updates.

```
CLEAR(F2:KC3,-1) ! Clear minor-most components
F2:KC2 = F1:KC2
F2:KC1 = F1:KC1
SET(F2:K,F2:K)
LOOP
NEXT(F2)
IF F2:KC1 <> F1:KC1 OR F2:KC2 <> F1:KC2 THEN
    BREAK ! No longer meeting range
END
F2:KC2 = New:F1:KC2
F2:KC1 = New:F1:KC1
Cascade_Updates
END
```

I have left out several vital details but this is enough to show the nature of the problem. This code fragment (and every bit of file IO/error handling that goes with it) appears for every relation with RI restrictions on it. If you have 100 files, 250 relations means 250 copies of the code. It's not too surprising RI code frequently blows the segment limits in the legacy templates.

The challenge is to “proceduralize” the above code.

When trying to abstract out an algorithm I like to go through the code coloring the lines. I use three colors: blue for base classes; green for parameterized base classes; black for instance specific code. In my first pass over the code I just pick out the blue stuff: those lines of code which will always be the same no-matter which of the 250 copies of the code I am looking at.

Listing 2. Code to cascade RI updates with common code in blue.

```
CLEAR(F2:KC3,-1) ! Clear minor-most components
F2:KC2 = F1:KC2
F2:KC1 = F1:KC1
SET(F2:K,F2:K)
LOOP
NEXT(F2)
IF F2:KC1 <> F1:KC1 OR F2:KC2 <> F1:KC2 THEN
    BREAK ! No longer meeting range
END
F2:KC2 = New:F1:KC2
F2:KC1 = New:F1:KC1
Cascade_Updates
END
```

This ranks as grim. The vast bulk of the code is actually different between the 250 copies. You could put the loop in the base class and call out for the header and loop body, but the total lines of code (once you've allowed for two new procedures) actually goes up. Listing 3 shows five lines of code that replace the three blue lines.

Listing 3. Method to call RI virtuals.

```

RelationClass.UpdateSecondary PROCEDURE
CODE
    ! Virtual call- override for every relation
    SELF.UpdateSecondaryInit LOOP
        IF SELF.UpdateSecondaryIterate THEN BREAK .
    END

```

Perhaps the italic pen will yield better results. This time I can italicize lines with variables provided I then add a parameter to the procedure prototype to allow the value to be substituted.

Listing 4. Parameterized and base class code in green.

```

CLEAR(F2:KC3,-1) ! Clear minor-most components
F2:KC2 = F1:KC2
F2:KC1 = F1:KC1
SET(F2:K,F2:K)
LOOP
    NEXT(F2)
    IF F2:KC1 <> F1:KC1 OR F2:KC2 <> F1:KC2 THEN
        BREAK ! No longer meeting range
    END
    F2:KC2 = New:F1:KC2
    F2:KC1 = New:F1:KC1
    Cascade_Updates
END

```

There is a subtlety here. Why didn't I italicize the CLEAR? Because the number of components to be cleared is not a function of the relation, it is a function of the key used by the secondary file. Thus you cannot readily parameterize it. So now you get the code shown in Listing 5.

Listing 5. Parameterized method to handle RI update.

```

RelationClass.UpdateSecondary PROCEDURE(
    File F, Key K, *? F1Field1,*? F1Field2,
    *? F2Field1, *?F2Field2, ? New1, ?New2)
CODE
    SELF.UpdateSecondaryClear ! Virtual call- override for every
relation
    F2Field1 = F1Field1
    F2Field2 = F1Field2
LOOP
    NEXT(F)
    IF F2Field1 <> F1Field1 OR F2Field2 <> F1Field1 THEN
        BREAK

```

```
    END
    F2Field2 = New2
    F2Field1 = New1
    SELF.Cascade ! Virtual
END
```

Now each of the 250 code lumps becomes two small virtual procedures and one base class call (with eight parameters). This cuts down the line count although the actual amount of code generated is still quite high. Each *? parameter costs 30+ bytes, so 6 of them is 180 bytes. I have also snuck a little bug past you. I have been assuming throughout that there are two linking fields. There can (of course) be one, or three, or four etc. So you need copies of the UpdateSecondary procedure (and the other four) for each possible number of pairs of fields.

Now I have greatly shrunk the code (i.e. there will be no more 64K problems) and just about everything has been abstracted. Every now and then someone will call to complain that ABC doesn't support 9 linking fields in a relation and we can simply write a new UpdateSecondary9 (with 27 *? Parameters at 600+ bytes per call).

But code abstraction doesn't have to end here. This design has UpdateSecondary1, UpdateSecondary2 etc., and these procedures are really the same *except* in the number of parameters passed in. You can write a generalized UpdateSecondary procedure (except it won't compile!) as shown in Listing 6.

Listing 6. A general UpdateSecondary procedure.

```
RelationClass.UpdateSecondaryN PROCEDURE(
    File F, Key K, (*? F1Field1, *? F2Field1, ? New1) * N)
CODE
    SELF.UpdateSecondaryClear ! Virtual call- override for every
relation
    LOOP N Times
        F2FieldN = F1FieldN
    END
    LOOP
        NEXT(F)
        LOOP N Times
            IF F2FieldN <> F1FieldN THEN BREAK OuterLoop.
        END
        LOOP N Times
            F2FieldN = NewN
        END
        SELF.Cascade ! Virtual
    END
```

Listing 6 won't compile because that isn't a legal procedure prototype. But you can see the idea – I want to be able to pass in any number of fields without having to define the fields ahead of time.

Example 2 : Formatting a browse line

Another place that presented problems was the browse code. Most of the engine can disappear into a procedure (Bruce actually did this for CDD3.0). However there are three very large routines you cannot take down: filling a browse queue from data; filling the record buffer from the browse; and seeing if any data in the browse queue has changed. These three routines essentially look like this :

```
BrowseQ:Field1 = File:Field1  
BrowseQ:Field2 = File:Field2  
BrowseQ:Field3 = OtherFile:Field7
```

where “fill buffer” goes the opposite way to “reset buffer.”

Easy you say, that’s the same as parameterizing. But think about it! Restricting the number of linking fields to 9 is one thing, but the number of browse columns? We would have to go up to 100 just to avoid getting shot by the alpha testers! On the other hand if *only* I could get the LOOP N Times code from Listing 6 to compile then this really would be so easy.

(Some of you may think we could use the `:=:` syntax to move across the corresponding fields. In general that doesn’t work because it doesn’t allow for browse columns defined by local variables. It also suffers if you have two files in the browse with clashing field names).

In my opinion it was this problem that killed the CDD browse engine. Because the engine had to call back to the main code so frequently to do almost anything (and they didn’t have the virtual mechanism to clean things up) the code became almost impenetrable. So the engine died, the inline browse appeared, and the browse procedure became our main bugbear for over five years

What's the real requirement?

The job then becomes one of defining what it actually is about the LOOP N Times code that will solve the problem. I think it comes down to the following :

I need to be able to pass around a list of one or more field pairs which can then be manipulated as a *single entity*.

Think about those last two words; they are the key. If I can embody the LOOP N Times into a single line of code then I have the problem cracked.

My expression *field pairs* also betrays another consideration. In the browse case there are only ever two fields that are really interesting; for the RI code there are three interesting values (child fields, parent fields, new parent fields). The prototype for the

UpdateSecondary is also interesting. Note that the fields pertaining to the files are prototyped as *?, meaning they can be assigned to and from. The new fields are only ever used by value. It turns out that (in this example, at least) there are typically four different cases :

- 1) Single field. This is a list of fields with no partner. In fact the components of a key are stored this way which makes it possible to bring the CLEAR (keycomponent) into the base class as well!
- 2) Single field, buffered. These are fields which have to have a snapshot of their values taken *without changing and “real” program variables* so the variables can be later compared to those values.
- 3) Two fields. Two sets of fields, either of which can be assigned to and from the other.
- 4) Two fields, buffered. This is the most complex case of two sets of fields where either one may need snap-shutting.

Because the fourth case is much heavier than the others (although related) we decided to assign it to its own class which is derived from the field pairs class

The implementation - any ideas?

In order to understand how this class works you certainly need to understand queues but you also need to understand the ANY datatype. This is given an excellent coverage in the manuals which I shan't repeat. However, the key here is this: an ANY can act like a *? parameter OR a ? parameter dependant upon how you assign to it.

Specifically, if an ANY variable is NULL (has no value) then a straight value assignment to it produces a value ANY, while a reference assignment to it produces a variant any. Listing 7 shows an example.

Listing 7. Using ANYs to store values and references.

```
MyAny &= NULL
Field = 22
MyAny = Field          ! MyAny = 22
Field = 42              ! MyAny = 22
MyAny = 50              ! Field = 42, MyAny = 50
MyAny &= Field          ! MyAny = 42
Field = 62              ! MyAny = 62
MyAny = 72              ! Field = 72
```

Warning: CLEAR(MyAny) is equivalent to MyAny = 0. It is *not* the same as MyAny &= NULL.

The `Init` procedure is simple enough to use. It creates the queue that forms the basis of the class. A slight oddity is the call to `Kill` first. This is to allow a `FieldPairsClass` to be used and reused within a procedure. (Effectively `Init` acts as a `Reset`.)
`FieldPairsClass.AddItem PROCEDURE (*? Left)`

There are two notional `AddItem` methods (the second called `AddPair`). This one is used for cases one and two. Note the `ASSERT` to insure `Init` has been called. The `CLEAR` is dealing with some (rather nasty) memory management issues when dealing with `ANY` in queues (see the manual). The incoming variable is `&=` into the left hand queue element. It is then `=` into the right hand element. This distinction is crucial. It means that simply `AddIteming` a field is enough to snap-shot it so that it can be reset (or tested for difference) at a later stage. The parameter is called `Left` because you can think of it as something you can assign into (and which therefore appears on the left hand side of an assignment `(=)` operator).

`FieldPairsClass.AddPair PROCEDURE (*? Left,*? Right)`

This method is used for variant 3. Other comments are the same as `AddItem`. Note also that in this case `left & right` do not have any real significance. it is just a non-suggestive way of labeling the two entities.

`FieldPairsClass.AssignLeftToRight PROCEDURE`

This procedure is really meaningless in variant one (actually it converts a variant one into a variant two). In variant two this can be seen as a way of *snapshotting* the current values of all the variables. In variant three all the values from the variables passed in as ‘lefts’ will be copied into the variables passed in as ‘rights’.

Warning: Note the `PUT` after the assignment. This is because an assignment to an `ANY` variable can actually change the memory block allocated to the `ANY`. Hence you have to store the queue after an assignment *even if you know the is a variant*.

`FieldPairsClass.AssignRightToLeft PROCEDURE`

Again the use of this suggests you are not really in the variant 1 case. In variant two it has the effect of restoring all the variables passed in as `Lefts` to the values they had when an `AssignLeftToRight` was last done. (Which could be the implicit one at the `Additem` point). In variant three this is an assignment from the variables passed as `Rights` to the variables passed as `Lefts`.
`FieldPairsClass.ClearLeft PROCEDURE`

This has the same effect for all three variants, it `CLEARs` the variables passed in as `Lefts`. This is not the same as assigning to zero, because the left-hand side could be a string. It is also not *quite* the same as assigning to a blank string (consider `Cstrings` & `Pstrings`). Now you could argue that it *is* the same as assigning to a zero length string, which is true, but only by coincidence. This illustrates one of the big pitfalls of having a language “guru”

doing low-level classes. You can use your low-level knowledge to build assumptions into the system that are not required. The fact that *presently* all Clarion data-types can be CLEARED by assigning a zero length string is a very dangerous fact to build into a set of base classes (consider what would happen if you could pass a mixed-type group as a *?). The clearing mechanism is there to protect you from such assumptions, so the base classes use the full language facility where they can.

Note further that CLEAR(SELF.List.Left) is very different from SELF.List.Left &= NULL.

FieldPairsClass.ClearRight PROCEDURE

In variant two this clears the buffer values, in variant three it clears the variables passed in as Rights. This method is subject to the same considerations as ClearLeft.FieldPairsClass.

EqualLeftRight PROCEDURE

In variant two this compares the current values in each of the Lefts against the last snapshotted values. It returns a zero if any of the values differ. In variant 3 it compares each Left/Right passed in and returns a zero if there are any differences.

Note that this procedure effectively does a short-circuit evaluation which means the function returns as soon as a deviation is found. It demonstrates one of the reasons that I believe certain programming mantras can and should be violated in a controlled environment.

First the controlled environment. EqualLeftRight is 10 lines long, it fits on one screen and (I claim) should be understandable in one bite by a half-way competent programmer.

Now for the mantra. Good structured programming will teach you that any given procedure should have precisely one entry point and precisely one exit point. This procedure has two exit points. Why? Certainly efficiency, and also (I claim) clarity. Consider the obvious alternative in Listing 8.

Listing 8. A single exit point alternative to EqualLeftRight.

```
FieldPairsClass.EqualLeftRight PROCEDURE
I UNSIGNED,AUTO
B BYTE(1)
CODE
LOOP I = 1 TO RECORDS(SELF.List)
    GET(SELF.List,I)
    IF SELF.List.Left <> SELF.List.Right
        B = 0
    END
END
RETURN B
```

Now the method has the required one exit point. However there is an extra line of code and there are two extra assignments (BYTE (1) is an implicit assignment). But the real pain is more subtle. Imagine a big field list (100 fields) in which you are checking for a difference (say after an Edit-In-Place operation on a browse). This code will check all 100 fields even if the first one sets B to zero!

So you end up having to put a BREAK into the IF condition or code an UNTIL at the tail of the LOOP. The latter is less efficient still. The former is efficient *but* if you now draw a flow diagram of your algorithm you will find exactly the same logical structure as coding a RETURN but it took you 20% longer to say it!

This brings me to the Bayliss mantra: *keep it short and to the point, but then don't compromise!*

FieldPairsClass.Equal PROCEDURE

This is simply a logical short-hand for people using the FieldPairsClass as opposed to the BufferedPairsClass (where the explicit LeftRight is helpful).

FieldPairsClass.Kill PROCEDURE

Check over this code. The destruction sequence of queues with ANYs needs careful work. First you have to null out all of the any variables, then you can dispose of the list.

BufferedPairsClass

This class is really just an extension to the FieldPairsClass to handle case 4. Two fields are paired and there is a shadow third value. In some ways this makes it easier to understand than the FieldPairsClass. If ever Left or Right are assigned to/from then it is the values in the underlying fields that are being used. Buffer means the *shadow* which never effects any values in the “real” program.

Queue Derivation

The BufferedPairsClass is derived from the FieldPairsClass; that is to say whenever a buffered field class is being used without reference to the shadow value you can simply call the same functions as you would for a case three of the field class. The buffered field class is an extension for the case when buffering is needed. Now we could simply have implemented the BufferedPairsClass and used it for cases one through three. The main reason we didn’t is one of efficiency. ANY variables work extremely slowly compared to standard Clarion variables (about 30x slower, or similar to Visual Basic) and therefore maintaining an extra one or two for the very common cases (one through three)

was deemed unwise. The separation also enables the field class to have a relatively small, clean interface.

BufferedPairsClass.Init PROCEDURE

This procedure demonstrates a simple problem, with a simple enough fix, but to the unwary it can be very confusing. The FieldPairsClass contains a reference to a FieldPairs queue (with Left & Right ANYs). This is NEW/DISPOSED in the FieldPairsClass Init and Kill methods. The BufferedPairs class has a reference to a buffered queue with three fields. Now here is the problem: if the FieldPairsClass and BufferedPairsClass both have Init and Kill called then there will be two separate queues pointed to by two separate references. So the BufferedPairsClass.Init method does *not* call its parent. As a result there is only one copy of the queue.

But there is a subtler problem. Suppose the Equal method is called. This drills down to FieldPairsClass.EqualLeftRight which expects the SELF.List reference to be filled in, which it won't be. Bang!

Here is the fix. The BufferedPairsQueue is (very deliberately) just the FieldPairsQueue with extra fields added. The Init method &= the List in the FieldPairsClass to the RealList in the derived class. Now the methods in FieldPairsClass can access the same queue as those of the BufferedPairsClass but via a different reference.

Tech note: A particularly nice feature of queue and class references is that they contain type information. Thus CLEAR(MyQueueReference) will always clear the whole queue buffer. Similarly ADD(Queue) works on the *whole* queue.

BufferedPairsClass.AddPair PROCEDURE(*? Left,*? Right)

This method overrides the equivalent method in the base class. Later versions of it actually contain some rather intricate code to fix a subtle bug that I missed on the first lap. All the code is really trying to do is reference assign Left and Right (as per the parent function) and then CLEAR the buffer value (because I don't know whether to assign it to Left or Right). But the question becomes, what does it mean to clear an any variable? (See discussion on ClearLeft.) What I *really* want is to assign it to a value which will compare equal to the Left or Right variables if they have been cleared. The only general way I could think of doing this was to clear the Right variable and then assign it to the buffer. Of course people might object to me doing that so I temp-store it first.

I think the other methods are fairly self-explanatory given the FieldPairsClass explanations.

Finally

ANY variables (and type polymorphism) are key strengths of the Clarion language that make it possible to code complex database algorithms in a totally generic and safe way. The two field classes extend this paradigm up to lists of field pairs. If you scan the ABC sources you will find the field pairs classes are intrinsic to files, browses, drop combos and edit-in-place. If you scan generated source you will find AddPairs popping up very frequently. The combined effect of these facts is that most procedures can be generated without any need to derive the browse or file objects. This simplifies and reduces the amount of code required to use these classes and gives Clarion an implementation edge (from template or hand-code) over C++, VB and Object Pascal.

I hope this FieldPairsClass design overview has given you an insight to one of the fundamental building blocks of the ABC system.

ABC Database Class Design Notes

INSIDE ABC: THE FILEMANAGER

by David Bayliss

To understand the design strategy behind the FileManager you first need to understand the name. Many of the ABC classes have names which end with “Class.” That is to denote that they are logically complete entities in their own right. Clearly they are implemented using features of the Clarion language, but it is the class itself that logically provides the functionality.

When I came around to looking at the file part of the ABC structure I quickly realized that here was a very different problem. The largest bulk of file functionality already existed in the language in the form of the FILE structure support by the file drivers. In fact the file drivers *can* be viewed as objects (especially as Scott was keen to extend the file property syntax to expose some of the file driver data structures).

The file drivers have two major drawbacks when viewed from an ABC perspective:

- 1) They deal with files. At the template level Clarion files are uniquely gifted with all sorts of useful information supplied by the dictionary (including default values, validation, prompts, descriptions etc). The file drivers only (or mostly) support the information that is actually provided “down at the metal” of the file structure itself. Thus, if the whole of ABC were to use the

file drivers directly, all of the value of having a data rich dictionary would be lost to the class hierarchy.

- 2) They are a black box. A potential beauty of the Clarion language is the way that the grunge of file access is hidden, thereby providing a nice clean programmer interface. The problem is that the wrapper prevents the underlying functionality from being readily extended or overridden.

From these problems arose the idea of a `FileManager` class. Essentially each file would be owned by a `FileManager`. The manager would provide a wrapper around the file entity, and the wrapper would embellish the functionality of the file by utilizing (and acting as a repository for) the data about the file contained within the dictionary.

Future Proofing

The next aspect to consider is that the black box approach to file classes clearly cannot last for ever. If the ABC system really is to become a fully integrated development solution then, over time, it has to extend throughout the run-time system. The advantages to be gained by having a consistent object-oriented library that allows modification and expansion at all stages in the hierarchy are too compelling to be ignored. In fact, as Bruce Barrington announced at DevCon '98, the `FileClass` (note the name switch) project is under way and will be providing a major thrust (and raft of benefits) to our future products.

But this presents another problem. We have to be able to produce a `FileManager` interface *now* that will still be valid and supportable when the file drivers have changed completely. This is more challenging than the normal OOP problem. All classes need to encapsulate themselves, but the `FileManager` has to provide encapsulation for the file driver that is *not* in itself encapsulated. This is rather like the distinction between erecting a fence that will remind you not to walk on a flower bed and erecting a fence that will persuade an exuberant Labrador retriever not to walk on a flower bed. In particular it means that the `FileManager` has a higher proportion of PRIVATE and non-virtual methods than most of the other classes.

Static Usage

Another peculiarity of the `FileManager` (shared with the `RelationManager`) is that the instances of the class to be created will (at least by default) be static. Remember that the `FileManager` is really just the extension we would like to have made to the `FILE` itself (but couldn't). Thus it would have the same scope and persistence as a file buffer, i.e. global. But it's not quite global. Most files are actually threaded variables. This gives the `FileManager` a bit of a headache since some of the data it stores is global (the file name,

for example, is the same for all threads), but some of it is threaded (the open state of a file is thread specific). Therefore it was decided to build thread management into the `FileManager` so that it could control inter-thread resources directly.

Aims & Issues

In tackling a class of the size and complexity of this one, it is worth splitting out and enumerating some of the tasks and requirements it is to meet.

- 1) Structure Storage. To provide services to return the keys of a file, fields of a file, number of components of a key etc.
- 2) File Snapshotting. A problem of having global buffers is that when some other procedure wants to use a file buffer it tends to corrupt the buffer contents. In the classic template chain, for example, the RI code actually corrupts buffer contents. A requirement of the manager class was to allow file buffers (and file state) to be snap-shotted and restored at a later date.
- 3) Auto-Increment. The manager is to provide support for automatically incrementing key components.
- 4) Retrieval and Update of records. This is fairly obvious.
- 5) Initialization and validation. Whilst this is again fairly easy to specify it is one of the key capabilities of the Clarion data dictionary. `FileManager` must exploit these capabilities.

As well as providing some raw functionality there were some other issues and subtleties that significantly extended the work of implementation.

Firstly, the `FileManager` was to be smart and reliable. If you told it to open a file then it should go open the file, and do everything within its power to get the file open, including retrying, creating, building keys etc. The reason is to simplify the usage from within hand-code (or embed code). Because the methods will do all they possibly can to make the thing happen you can code as if it *has* happened. This meant the `FileManager` needed a tightly integrated `ErrorClass`. There was a problem, however, because sometimes you don't actually *want* the system to scream blue murder if it doesn't succeed. The fact the class failed is all the information you require. So most of the methods have "Try" equivalents, i.e. the `Open` method has a corresponding `TryOpen` method. The Try means that the method simply returns an error code rather than taking action on its own (such as presenting an error message).

Another subtlety is aliases. In Clarion aliases are treated pretty much as separate files. Whilst this is good from the IDE end of things it makes some of the object orientation horrible if files are mapped one-to-one with `FileManagers`.

Suppose the `FileManager` provided a virtual method to initialize a record into which you drop some embed code. Then whenever you create a record your code gets executed, unless of course you happen to be using an alias, which results in an integrity problem. Now you need to embed the code for every alias as well. Bleh! So we decided to get creative. The `FileManager` for an alias class contains a reference to the `FileManager` for the “real” file. Then whenever the alias `FileManager` is called it simply recalls on to the actual `FileManager` to do the work. Of course to do this we need to make frequent use of the File snapshotting technology.

The final tweak, one which came up fairly late and which is a mixed blessing, is the notion of `LazyOpen`. The idea here is that if you use a lot of files in a procedure it is better to open them one by one when required than open them all in a burst at the head of the procedure (which can cause a delay in SQL). This is no problem for people using ABC (as the ABC classes always know if they are about to touch a file) but it causes a bit of a headache for people doing hand-code as they need to warn the ABC system (using the `UseFile` method) that they are about to drill underneath the ABC layer. (Incidentally, this is one of the problems that will go away once the `FileClass` is implemented).

Initialization

Originally the initialization of the `FileManager` was very complex. The template generated code called many `Addxxxx` routines building up within the `FileManager` a replica of the data stored within the file driver itself. From beta three onwards the file drivers were extended so that the information within them was more readily accessible. For this reason the initialization interface withered down to two genuine methods, both used to tie the `FileManager` into the outside world.

I have added two others, `Kill` and `SetThread`, as they are both tied to `FileManagerManagement` rather than `FileManager` itself.

`FileManager.Init` PROCEDURE(`File` `File`,`ErrorClass` `E`)

There is a one-one correspondence between instances of the `FileManager` and files used in the program. The `File` parameter sets up this mapping. The `ErrorClass` defaults to the global error manager (when the template generated init code is called). The advantage of having this reference as protected is that a given file (or batch of files) can have a different error manager installed to allow (say) less extreme responses to a given set of files being missing.

The coding is fairly straight forward, the usual array of queues being NEWed and class variables being set.

The only real magic is in the `AddFileMapping` method. This is there to help with lazy open, or more specifically, to help when a view is used. The view driver can return a list of files that are used in a view, but this is not quite what we want. We want a list of `FileManager`s. The mapping code is thus there to provide a mapping from file references to `FileManager` references. The mapping takes advantage of the fact that both file references and `FileManager` references can be used as LONGs. The `Filemappings` will be able to die once the `File` class comes along, and are thus private. In fact they use Clarion's rather nice "in module" private technology to allow the private methods to be completely removed from the .inc file.

FileManager.AddKey PROCEDURE (KEY k, STRING Desc, BYTE AutoInc)

This is the one remaining `Add` method. It is required because there are two pieces of information stored within the dictionary and required by the `FileManager` that are not available to the FILE driver. The first is the textual description of each key, and the second is the auto-increment nature of the key. The `AutoInc` value specified is the component of the key which is auto-increment. Within the templates this will always be defined as the minor-most component of the key although this restriction is not imposed by the base classes. (That said, non-minor-most component auto-incrementation is much less tested than the more normal form!)

Whilst the method itself is of little strategic importance it does use a number of relatively new or advanced coding techniques so it is worth a little further investigation.

The principle job of `AddKey` is to form a new record in the `FileKeyQueue`. This is a private property. Because Clarion allows unresolved forward references (i.e. allows opaque types) the queue structure is not defined within the .inc file, only within ABFILE.CLW.

Assigning the key reference and description is fairly straightforward. Use of the description field as fixed length (rather than assigning a reference) is not as painful as you may think because Clarion queue buffers are compressed when stored in memory so any trailing spaces simply squash up.

The interesting code starts with this reference assignment:

```
self.keys.fields &= NEW KeyFieldQueue
```

This code is setting one of the fields of the `FileKeyQueue` to be a new instance of the `KeyFieldQueue`. That means there will be a `KeyFieldQueue` for every record of the `FileKeyQueue`. In other words this is a queue of queues. This is a very powerful technique, but it is also something to be careful of. In the `Kill` method for the `FileManager` we have to step through the `FileKeyQueue` freeing up each one of these sub-queues.

Next note that `SELF.HasAutoInc` is set if `AutoInc` is true. `HasAutoInc` is set on a per file basis, not per key. The information is really redundant since `HasAutoInc` could

always be computed simply by stepping through the values in the key queue, but `HasAutoInc` is hit sufficiently often it was deemed worth the extra cost (1 byte!) of storing a copy. `SELF.PrimaryKey` is stored in the object for similar reasons.

Finally there is a loop filling in the nested queue with one entry for each key component. This queue has two gotchas to watch for. The first is it contains an ANY. You must *always* CLEAR a queue buffer containing ANYs before assigning to the fields (see the Language Reference Manual). The file driver can return the field number of a key component, and the `FileManager` has the record buffer in `SELF.Buffer` so we can use `WHAT` to return an ANY referring to the field of the key component. This then gets stored in the queue.

The other required piece of information is the “bind” name of the key component. This comes from `PROP:Label`, but the code does one more piece of pre-computation. If the key is case insensitive and the field is not numeric then we store `UPPER(Fieldname)` rather than `Fieldname` in the queue. This is neat as it means routines further down the line can simply use the stored field name without regard to case sensitivity and the issues are dealt with for them.

FileManager.Kill PROCEDURE

I suspect some would argue that `Kill` is not an initialization method, but I disagree. `Kill` is really the mirror image of `Init`. I nearly always write them at the same time (literally a line of `Init`, then a line of `Kill`). The job of `Kill` is to undo anything that has been done during the `Init`, and sometimes it has to undo the work of later routines too, but acting as an un-`Init` is a must.

Notionally this routine is very straightforward but there is some quite involved picking apart of the queue of queues. The outer loop steps through each key and the inner loop steps through each key component. Finally for each key component the name is disposed and the ANY variable is freed up.

Failure to do any of these steps would result in a memory leak.

FileManager.SetThread PROCEDURE

As suggested in “Static Usage,” p. 460, one of the problems the `FileManager` has to contend with is that a file effectively has a new instance on each thread. This needs mirroring in the `FileManager`. Rather than clone the `FileManager` for each thread (and support an inter-communication layer) it was decided to support a queue within the `FileManager` where each thread was mirrored by a record in the queue. This may not please the object purists but as the thread specific data totalled 10 bytes compared to >1K for the whole object, the engineers will understand the logic.

The methodology is thus quite simple: each procedure starts by calling `SetThread` which pages in the correct data for the current thread; the procedure can then simply access the data using `SELF.Info`. There is a slight gotcha here. Because `SELF.Info` is a queue

buffer, any method altering a value in SELF.Info must then do a corresponding PUT(SELF.Info) or the value altered will be lost upon the next call to SetThread.

The implementation is straightforward. If the file is threaded (early ABCs ignored this condition!) then the current thread is read from the system. The current list of thread data is looked up, and if it doesn't exist then a new record is cleared and created.

The final line of this procedure is actually a cheat. It has nothing to do with setting the thread; it simply takes advantage of the fact that all FileManager methods will call SetThread before commencing execution. SetThread thus because a suitable repository for *always call* items. In this case the error manager is being primed with the current file name so that any errors within the methods themselves can simply Throw, confident that pertinent details have been filled in ahead of time.

The FileManagers are coded in a fairly unusual context which has greatly influenced some of the design decisions taken during their implementation. Whilst some of the aims of the class can be gleaned simply from reading this chapter, most fruit is available to those that actually settle down and read the ABC code along with the corresponding comments. This is actually something I would always encourage you to do. The backbone of ABC amounts to around 4,000 lines of code, so if you aim to master 100 lines a day you will understand the basic ABC paradigm completely within eight weeks! The FileClass amounts to 25% of that work.

Administration

This section details those methods provided almost entirely as wrappers upon internal information for the benefit of higher level methods and/or methods outside of the FileManager.

```
ClearKey PROCEDURE (KEY K, BYTE LowComp, BYTE HighComp |
, BYTE High)
```

This method is there to provide a shortcut for a piece of template code that occurred very frequently. Essentially it handles the problem of a multi-component key where you wish to perform a SET(KEY, KEY) but you only know the major components. In order to ensure the SET(KEY, KEY) gets you to the start of all the records you require you need to clear the low order key components. Clarion does not have a CLEAR(KEY) so the FileManager provides one for you.

Rather than clear the whole key the routine allows you to specify the low (majormost) and high (minormost) components you want cleared. This is to allow minor component clearing to happen when the major components have already been filled in.

The method works by first performing a SetKey so that the current record of the FileKeyQueue holds information for the current key. Then the key components are stepped through from low to high and the KeyFieldQueue is fetched to retrieve the information for the current component (see the AddKey method). The GET is error trapped with a simple return if the component doesn't exist. This is to allow the HighComponent to be specified as 255, meaning "to the end."

The XOR logic illustrates a useful trick (and hides a complexity!). Remember that as you are trying to get *all* the records in a SET (KEY, KEY), you might assume that means you just CLEAR all the key values low. But wait a minute; suppose you are about to do a PREVIOUS rather than a NEXT. Then you need to clear all the values high. This works in the common case of ascending key components, but remember it is possible to have descending key components. Worse yet you can mix ascending and descending in the same key. If you sit down with pencil and paper (you may be able to do this in your head, but I needed pencil and paper) you will find you need to clear a component low if it is ascending and you are clearing low, or if it descending and you are clearing high. This can be expressed using a disjunction (OR) of conjunctions (AND) but the XOR operator wraps it up perfectly and goes down to one machine instruction. I could have coded this more tightly still as:

```
CLEAR (SELF.Keys.Fields.Field, |
    CHOOSE (~ (SELF.Keys.Fields.Ascend XOR High)))
```

But I thought that might be just a little *too* scary.

```
GetComponents PROCEDURE (KEY K), BYTE
```

This simple little method simply returns the number of components in a key. It uses SetKey and the fact that there is one KeyFieldQueue record for each component of the key.

GetEOF PROCEDURE, BYTE

The FileManager has a very specific meaning for EndOfFile: it means the last attempt to NEXT or PREVIOUS a record failed because the end of file has been reached. Specifically, if you have a file with 10 records EOF is true after the 11th NEXT, not the 10th. As such GetEOF is really just a short hand to detect a specific error condition.

The functionality could almost certainly be achieved by looking at the return code from NEXT/PREVIOUS and then delving to see what the error identifier was. Again this is a situation where the FileManager does work simply to reduce the amount of coding required by users of the object.

GetField PROCEDURE (KEY K, BYTE Component), *?

This method is used to return an ANY variable corresponding to a given component of a key. I didn't want to have to protect the rest of my code against GetField returning a null

so the procedure ASSERTs that the incoming component will be found. In other words, GetField gracelessly handles out of range components.

This does illustrate another agenda within ABC: offensive programming (<http://www.users.globalnet.co.uk/~dabay/offensiv.htm>). Defensively I would have coded so that an out of range value returned a null, which would take two lines of code. Then on the receiving end nulls would have been handled, presumably in some “see if we can still keep going” fashion.

There are four calls to GetField in abfile (i.e., this method is relatively underused). Each would have had to temporarily store the GetField result, test for the null and do something smart with it. This might have taken five lines of code each (one for the declaration, one for the extra assign, two for the null test, one to handle the null case). In total I would now need 22 lines of code to handle something that should *never* happen as opposed to the one line of code used in ABC. Doing that throughout a heavily integrated file like ABFILE could turn 2000 lines of code into 40,000 lines of code 95% of which would be rarely executed and thus minimally tested. QED.

GetFieldName PROCEDURE (KEY K, BYTE Component), STRING

This is really there for the benefit of methods using the PROP:Filter technology on a view. It provides the BIND name of a give key component.

GetName PROCEDURE, STRING

The FileManager has to cope with two possibilities for the name of a file. It may either be a constant or a variable (the latter corresponds to the case where the NAME attribute on a file contains a string variable). GetName is there to encapsulate this dilemma from the rest of the class. If a variable file name has been assigned then it returns that, otherwise it assigns the constant provided to it by the driver itself.

KeyToOrder PROCEDURE (KEY K, BYTE MajorComp), STRING

This method really takes GetFieldName one logical step further. Rather than just return a field name corresponding to a key component, this method returns an ORDER clause (in Clarion syntax) that is equivalent to this key starting at component MajorComp. A value of one thus gives the whole key as an order clause, two skips the leading component etc.

Note that the null key case *is* defended against. This is because it is totally reasonable to have a null key specified as the sort key of an object (corresponding to not specifying a key in the file schematic).

The only real complexity is in the RetVal assignment. The first CHOOSE is there to prepend the field name with a comma only if the string being built up is non-null. The second CHOOSE is there to place a leading ‘-’ before a descending key component (the

view driver treats ?string as a descending string, it does *not* convert it to a number as the language would).

SetKey PROCEDURE (KEY K) , PROTECTED

SetKey is used to fetch the correct record within the FileManager key queue for the usage of a key passed in to it. You cannot sort a queue on a reference field so the method has to loop through the queue finding a match. Files don't have *that* many keys so this should not be too onerous. I could start the method with a check to see if the current record value already matches as a kind of *first level cache*, but the downside is that this would hide a raft of bugs where people had not done a PUT after modifying the key information.

The loop illustrates an interesting and occasionally useful quirk of Clarion. You can have loop head *and* loop tail conditions (WHILE and UNTIL) in the same loop. The conditions are tested (and code body executed) in the order they appear lexically.

Again note the assert. A failure to set the key throws an error.

SetName PROCEDURE (STRING Text)

This method is a counterpart to GetName; it only allows the name to be assigned if there is an underlying variable for the NAME attribute of the file. By having the GET/SET in the FileManager the burden of tracking the global variable name disappears (it simply becomes the province of the dictionary). This makes it far easier to have an automated path assignment system built in.

Error Handling

Each FileManager re-vectors the error manager calls through its Errors property. This serves one main purpose: it allows a global object to be referenced from within base class code. The secondary purpose is to make the error handler used by the FileManager re assignable. This is useful as the file system is one of the major generators of errors and the file calls are usually out of the direct control of the programmer. The ability to intercept errors on a file by file basis allows fine grain recovery mechanisms to be written. In addition to having a single vector point, the FileManager has a small suite of routines through which all FileManager/ErrorClass interaction is managed. Again the purpose is to make errors and recovery mechanisms overridable with a minimum of effort.

GetError PROCEDURE , SIGNED

The FileManager stores the last file error thrown within it. The number is the ErrorClass number, and it has nothing to do with ErrorCode or Error. It should be noted that ErrorCode et al are *not* valid upon return from FileManager methods. In

particular it is quite probable that the `FileClass` (coming in a future major release) will not utilize `ErrorCode` and `Error` in normal operation and thus the `FileManager` will not even have error codes available. The error suite is one of the instances of the `FileManager` trying to smother an encapsulation leakage coming from underneath.

SetError PROCEDURE (USHORT Number)

This method separates out the recording of an error condition from the `Throw` (or exception) that the error could raise. Occasionally this is used to simplify internal coding, but more usually it is used in the `Try` methods so that they can return an error signal and leave the `ErrorClass` able to `Throw` the error if the caller requires.

Throw PROCEDURE (USHORT ErrorNumber) ,BYTE ,PROC ,VIRTUAL

This function is purely a syntactic convenience. It is equivalent to a `SetError` followed by a `Throw`.

Throw PROCEDURE ,BYTE ,PROC ,VIRTUAL

This routine takes the last error number (as recorded by `SetError`) and simply forwards it to the `ErrorClass` stored in `SELF.Errors`. The main purpose of this routine is simply to provide a common focus point (and thus override point) for the `FileManager` error handling. The return value comes from the `ErrorClass` and denotes the severity level as attached by the error class. This could be used to provide a sophisticated error recovery mechanism, by default most `Throws` are considered fatal and this facility is not used.

It is worth noting that although `Throw` does not pass on the file label at this point, the `ErrorClass` does have access to the file name as this has been set up by the `SetThread` method as detailed earlier.

**ThrowMessage PROCEDURE (USHORT ErrorNumber ,
STRING Text) ,BYTE ,PROC ,VIRTUAL**

This is a simple extension to `Throw` to allow an extra message to be passed on to the `ErrorClass`.

Snapshots

The snapshot interface's purpose is to allow file state and buffer contents to be saved and restored by anyone without them having to know the structure of the file. The routines all use a handle to denote a particular state. This handle is undefined (presently it is an ID number within a queue.) Eventually these routines will become vectors for fresh instances of the file class to be created and destroyed.

The words *buffer* and *file* have specific meanings. Buffer means the *contents* of the current record; that is the record buffer but also the memo contents. Blob contents are not stored as the overhead is potentially too onerous. File means buffer plus additional file state information such as Held, Watched, auto-increment done etc. For this reason all of the Buffer methods are fairly cheap involving only memory copies, while the File methods also involve disk access.

EqualBuffer PROCEDURE (*USHORT Handle), BYTE, VIRTUAL

This method is used to check if the current record contents differ from those when the snapshot (denoted by the parameter) was taken. For example, this might be used to see if a cancel on a form should be allowed to happen without user intervention.

First the Handle is looked up in the buffer queue; this gives the previous contents of the record buffer which can be compared byte for byte against the present values (this function is boolean - it doesn't say *how* the two buffers differ). If the two record buffers are the same the routine steps through the memos of the file seeing if they differ. The stored memo buffers are (by convention) stored consecutively in the queue following the record buffer. The present contents of the memos are retrieved by using MyFile{PROP:Value,-memonumber} (the negative number indicates this is a memo). This was necessary as it is not possible to store ANY references to memos as memos are created on the heap at file open time (on each thread) and are thus highly treacherous when involved with references.

RestoreBuffer PROCEDURE (*USHORT Handle, BYTE DoRestore=1)

This routine is used to restore the contents of the file buffer to the point they were when the SaveBuffer was called. If you pass in a zero as the second parameter then no restoration is done but the memory is freed. Commencing with C5EEA this routine actually becomes a shell that calls into RestoreBuffer(handle, filemanager, byte).

RestoreBuffer PROCEDURE (*USHORT Handle, | FileManager FM, BYTE DoRestore = 1), PRIVATE

This routine allows the contents of a buffer to be restored to the present file buffer from contents snapshotted by the passed in FileManager. Now *in general* restoring to a file other than your own is a dangerous, unmaintainable and generally very stupid thing to do (this is why the only public interface to RestoreBuffer passes in SELF). However in the particular case where the "other" file is absolutely identical structurally to your own, and is guaranteed to be so, it does give an extra degree of flexibility. We use this facility when dealing with aliases. However when reading this code you should generally assume that Frm and SELF are the same thing (if you're writing it, the distinction is vital of course!) Other than that, this code is essentially analogous with EqualBuffer, the only extra being KillBuffer which first frees them memory used for the buffer contents and then kills the queue record.

RestoreFile PROCEDURE (*USHORT Handle)

This is used to restore a file to the state it was in when the snap shot was taken. The current file position, sort sequence, held and watch state are all recorded along (since C5EEA) with the auto-increment state. Note that additionally the record contents are restored after the file position. This is to allow for instances where the current record had begun to be modified at the point the snap-shot was taken.

As with RestoreBuffer, RestoreFile has been split out to aid the use of aliases, or more specifically, to allow FileManagers of aliased files to re-vector their methods through the Filemanager of the actual file without corrupting the current state of the actual file.

RestoreFile PROCEDURE (*USHORT Handle, FileManager FM), PRIVATE

The file state (as opposed to record contents) is retrieved from the Saved queue. The Saved.Key element is the key *number* of the key active when the snapshot was taken. If this is non-zero then the key reference is found from the file driver and used in the RESET (otherwise the File is used). Because Watched and Held are read-only properties in the file driver they have to be restored by re-arming them and applying a NEXT. Having performed the NEXT (and thus “corrupted” the buffers) the buffers are restored. The auto-increment state is then put in place. Note the PUT on the SELF.Info to store that information for the current thread.

Actually this raises a slight cheat. Many of the file methods need to start with a SetThread for reasons previously described. Many then needed a UseFile to prime the lazy open. UseFile also needed to do a SetThread, so SetThread was often called twice. This is clearly inefficient so we cheated and allowed an *information leakage* that stated that UseFile does, and will always, perform an implicit SetThread. Again we find that ABC is not just about science, it is also about engineering. We allowed for one assumption and removed 15 lines of code and an efficiency drag on most of our core functions, and also lost some conceptual purity.

SaveBuffer PROCEDURE, USHORT

This code snapshots the current contents of the record buffer; most of the code is analogous to EqualBuffer. The interesting piece is the allocation of the Id to act as a handle to the outside world. At first sight you can simply get the number of records in the queue, add on one and you have your new Id. Better yet, you don’t need to store the Id in the queue; you simply use the Id as a record number.

Further, in just about all the testing you ever do, it will work beautifully. But sometimes, somehow, it will corrupt when the users use it. The reason is that simply counting the records only works if Save/Restore pairs are performed in a stack-wise manner. If a deletion from the queue has happened *in the middle* then the next RECORDS will return a

value lower than the current highest `Id`. Actually it will even work if the restores are not done stack-wise provided the result has been stack-wise by the time you do the next `Save`. If you do the `Save/Restores` in an unpaired way you will actually get the identifiers duplicated in the queue and havoc ensues. The solution is that you get the final record in *sorted order* and then add one on to whatever you receive back. `DupString` is a private member function used to allocate heap for, and copy the value into, a temporary string (like `strdup` in C++).

SaveFile PROCEDURE , USHORT

This method is the mirror of `RestoreFile`. Note that rather than replicating the buffer storage code, `SaveFile` simply calls on to `SaveBuffer` and stores the result. It is worth mentioning that the handles returned from `SaveFile` have no relation to those returned by `SaveBuffer`. You cannot `SaveFile/RestoreBuffer` or vice versa.

One slight tweak is the storage of the current key. You cannot simply save a key reference as that will not work when you are restoring to a different `FileManager`. Instead you have to store an ordinal number corresponding to the declaration order of the key. That number is computed using the loop. Note too the usage of a cast from `CK` (which is a long) to a key reference:

```
K &= (CK)
```

The rule is that a numeric value can be assigned in place of a valid reference of the right type. `CK` in itself is not a value (it is a variable) so the parenthesis is used to form a value. This form of casting (which can be used in conjunction with `ADDRESS` and references) allows all of the (horrible) type conversion common to C++. It should be used extremely sparingly, but when needed it is brilliant.

A Thought

If you have been following this chapter in the source code, reading and understanding as you went, it is quite probable that by this point you are thinking. Hey! This stuff is all obvious; what is all the fuss about? If so, this chapter has worked. If not it may be worth your while backtracking to see where the confusion enters. Object systems are hierarchical, one layer builds upon another. Therefore, *comprehension* of object systems tends to be hierarchical. If one layer doesn't make sense it typically means you didn't *quite* catch hold of the layer underneath. Happy hunting...

Now it's time to get down to the meat of the class; the dictionary interaction and the file access itself.

Record Initialization and Validation

Whilst initialization and validation are logically distinct tasks they are grouped together here because in the fullness of time they will form the basis of what is really one concept: *business rules*. People can use business rules to mean just about anything they like. I use it to mean information *about* the data that is not contained explicitly *within* the data.

Some of these rules are already handled by the file driver. For example the DUP attribute in the Clarion language (upon a key) specifies something about the data you cannot get directly from the data (although you may be able to intuit it). The Clarion dictionary gains some of its power by specifying initialization values and validation values within a single repository. In Clarion 2.003 this information was then scattered (at code generation time) throughout the application. Any form upon a file would contain code to initialize and then validate the record buffer. In ABC this code is contained in the (derived) `FileManager` object.

One of the ABC aims is to keep the initialization and validation information in one place so that if the rule is dynamic (i.e. has to be done with an embed point) then that embed only needs to be placed once and *all* accesses to the data obey the new rule. This is particularly important as ABC was to have edit-in-place browses and automatic updates from drop-combos. In other words, we could no longer rely on forms being around to act as guardians of the file.

A consequence of the heavy dictionary tie to these routines and the need to pander to embed code is that a number of these methods are blank; they are placeholders for template generated code. In these instances I will describe the code that I envisage will go into the derived form of these methods (and which, purely coincidentally, the templates generate.)

```
CancelAutoInc PROCEDURE(<RelationManager RM>) |
    ,VIRTUAL, BYTE, PROC
```

Initializing a record with an autoincrement key results (in our implementation) in a record being stored on disk to act as a placeholder for the autoincrement value. If you cancel the operation then that record needs to be deleted. The `CancelAutoInc` is the clue to the `FileManager` that the record that was initialized is about to be thrown away. The `RelationManager` parameter is a solution to the problem often referred to as the “orphaned children problem.” In a 2.003 application if you insert a record with an autoincrement key, go to the child tab, insert some child records and then cancel the form, the child records persist. Of course you can’t see them until you insert a new record and find it automatically gains some children! This is really scary on forms with many children where the child tabs may not even be perused during the insert. ABC gets around this problem in that the form procedure passes the `RelationManager` in to the `CancelAutoInc` and the `CancelAutoInc` undertakes to delete any children (or refuse to cancel the autoincrement) as appropriate.

ABC Database Class Design Notes

The implementation is much simpler than the description: if an autoincrement has happened then either `DELETE` or the RI `DELETE` are called. In the latter case the response is noted, as an RI relation of restrict means that the autoincrement *cannot* be cancelled whilst the children persist.

```
PrimeAutoIncServer PROCEDURE (BYTE HandleErrors),!  
    BYTE,PROC,PRIVATE,
```

`PrimeAutoInc` and `TryPrimeAutoInc` are really just two virtual hooks on the `AutoIncServer`. Non-zero for `HandleErrors` implies the server will take all possible steps to ensure the autoincrement happens. The `TryPrimeAutoInc` only is called when `TryInsert` has been called. This is never done with the shipping templates but has been added at the request of a third party.

The routine starts by checking the guard variables. `PrimeAutoInc` can be called multiple times to allow for the cases where priming is done in the browse or in the form. It also allows for inserts to be done *without* pre-priming of the record. This is an instance of what I call “objects with attitude.” Basically the `FileManager` knows that priming should be done once, and only once, and thus it does it at the first opportunity it is asked, or at the last minute if no-one asks it!

The algorithm for autoincrementing is essentially the one used to return a unique handle in `SaveBuffer`, find the last element and add on one. However, as you shall see, some of the little details make things a vast amount more complicated.

The first `LOOP` is a loop to allow for multiple reruns of the bulk of the procedure in the case where an attempt to autoincrement failed. The failure is most likely to be because between the reading (from disk) of the current highest element, adding on one, and then `ADDING` the record, another station got there first and `ADDED` the record with that number! The simplest solution to this failure is therefore to try again from the beginning, and the outer loop encodes that logic. Of course you can’t keep doing that for ever because something may really be wrong. If you go to the end of the outer loop, after the `ADD (File)` you will see logic to trap the error.

If this is the third failure the error manager is invoked to see if the user wants to try again. If he does you simply try again (three times), and if he doesn’t you break out with a failure. If the add was successful the method simply notes the autoincrement has been done and then returns `Level:Benign` (which is zero and means “okay”).

The second (or inner) loop introduces a new complexity. Since it is possible for there to be multiple autoincrement keys in one file, this code has to find incremented values for each of them. So it loops on each key, executing the body of the loop if it is autoincrement. The `SaveBuffer` is important because `PrimeAutoInc` should only change those fields which are tagged in the dictionary as autoincrement (or it won’t be possible to support delaying the autoinc allocation until the insert point). The code then splits into two

branches, the relatively easy one where the key has one component, and the multi-component case.

One component is handled by fetching the component any variable and assigning to AutoIncField. The file is SET to key order and the final record (or first for a descending key) is fetched. If no record is found (i.e. the file is empty) the new autoinc value is one, else it is the highest value plus one. It's necessary to use AutoIncField/AutoValue rather than just using the underlying fields as the object will later restore the buffer (which will corrupt the field values) and *then* perform the autoincrement assignment.

The multi-component case uses exactly the same algorithm. The complexity is in finding the “final” record because you don't want the final record, you want the final record *that matches* the current record buffer in all components except the last.

ConcatGetComponents is a simple (and ugly) way of snapshotting the leading components of a key to later see if they have changed. The method then clears the minor-most (and thus autoincrement) key high and does a SET (K, K) followed by a Previous or Next as before. In the NoError case it has to check that the record fetched *did* match in the leading components; if it did then it can use the AI value fetched and add on one, otherwise it knows that no records currently match the major components and thus can use one.

Having computed the new field value (using either method) it restores the buffer contents and then assigns the new autoincrement field value into place. Once that is done that for all autoincrement keys it can try ADDing the new record.

If this procedure looks long and horrible it is because it is. My general rule of thumb is that any procedure more than a page long is a bug. Again you can see engineering and efficiency overcoming science. I could remove the “OneComponent” arm of the IF, and the only effect would be a few more string compares (no big deal), but this also changes this code:

```
CLEAR(OnlyKeyComponent)
SET(K,K)
PREVIOUS(K)
```

into this:

```
SET(K)
PREVIOUS(K)
```

When you consider that the one component case is the standard third normal form case it was considered that the code verbosity was worth it. That said, the file drivers now spot the above optimization in most instances so we may be able to simplify this code soon.

PrimeFields PROCEDURE ,PROC ,VIRTUAL

The default implementation of this method is blank; in the derived form the templates insert an assignment for each field that has a non-blank initialization value in the dictionary. The PrimeFields routine may *not* assume that autoincrement has been done. Any required blanking will have been performed.

PrimeRecord PROCEDURE (BYTE SuppressClear = 0) ,| BYTE ,PROC ,VIRTUAL

This method is called to prime the record *whatever that means*. In the current implementation that involves calling PrimeFields to prime the field values and then forcing the autoincrement to prime. Note that this method overrides the *attitude* built into PrimeAutoInc; when PrimeRecord is called a new autoincrement record *will* be made. The method has the facility to clear out any fields it doesn't explicitly prime or to leave them alone. This functionality is required to allow the ViewManager to place extra priming information into the record buffer (such as range-limited components of keys) and is controlled by the SuppressClear flag.

The primary case (where AliasedFile &= NULL) is quite straightforward. The interest comes when the file is an alias. Here you *don't* want to call the priming functions of the alias (because they don't have the required embed code); you want to call them in the "real" file. The code for this has changed in C5EEA; I am describing the *new* code.

First the "real" file has to be opened on this thread (it may not be), and then the file contents/position have to be snapshotted so that eventually the file can be restored. Now in the case where the clear is to be suppressed the code has to assume that the record (of the alias) contains interesting information that may be required by the field priming or autoincrement. So it has to get the information from the alias into the real file. This is done by the devious device of snap-shutting the alias file buffer and the restoring from the alias FileManager into the "real" file buffer. Having done this it can perform the PrimeRecord on the "real" file; if this is successful it needs to copy the result back to the alias file, done using the save/restore trick again. Finally the "real" file is restored to normality and closed. This may look odd: why restore a file then close it? Simply because Open/Close only increment/decrement counters. The Open only opens if this is the *first* open, similarly the Close only closes if this is the *last* close.

ValidateField PROCEDURE (UNSIGNED Id) ,BYTE ,PROC ,VIRTUAL

This function returns Level:Benign if the field is okay, otherwise it returns an error level. The default implementation only handles the alias case; the actual field validation is handled in a derived method. The Id is the number that would come back from a WHERE statement. The template code simply generates a CASE statement on the field number, it would be possible to produce a more sophisticated version using WHAT. We went for simplicity as this is a very common place to put embed code and also ValidateField is

hit quite frequently for control by control field validation and therefore performance was an issue.

```
ValidateFields PROCEDURE (UNSIGNED Low,UNSIGNED High,  
    <*UNSIGNED Failed>) ,BYTE,PROTECTED,PROC,VIRTUAL
```

This method is simply an encapsulated way of calling a range of ValidateField calls. It simply spools over the field numbers contained within the (inclusive) range. If one fails then the failure number is assigned to Failed. Again the alias case is handled by re-vectoring through the “real” file. It should perhaps be noted that for efficiency the alias code does a SaveBuffer, not SaveFile. This implicitly assumes that the field validation code will *not* mess with the current file state (i.e. no file I/O will be done on the primary file).

```
ValidateRecord PROCEDURE (<*UNSIGNED Failed>) ,BYTE,VIRTUAL
```

Another syntactic short-hand, this simply calls ValidateFields to ensure that every field in the record is validated.

File Driver Replacements

These methods are direct replacements for the equivalents in the file driver. They are generally there to perform advanced error handling or to ensure that other FileManager routines are called at the appropriate moment.

```
BindFields PROCEDURE ,VIRTUAL
```

This method is called at a suitable point to bind the fields. By default the code simply performs a bind on the record buffer. The templates further override this to perform binds on any memos that are available. The call can also be overridden by the user (in C5) to bind logical names (as opposed to labels) as required.

```
Close PROCEDURE ,BYTE ,PROC ,VIRTUAL
```

The close mechanism (tied to the open mechanism) is designed to avoid the needless opening and closing of files upon a thread. The FileManager therefore maintains a count of the number of times a file has been opened and closed. Upon a close it therefore decrements the counter. *If* this close has closed the final remaining open then the close is actually performed upon the file. The Used flag denotes if a file was *really* forced open (by an implicit or explicit UseFile) as opposed to just logically opened. Errors are not trapped by this routine as any *real* problem with a close will be picked up again when the file comes to be re-opened. For this reason ABC also eschews the TryClose.

Fetch PROCEDURE (KEY K) ,BYTE ,PROC

The Fetch routine is really just a wrapper for a file GET. The error case results in the buffer being cleared. Most of the work is done by re-vectoring through TryFetch, and though this doesn't save a great deal of code and is marginally less efficient than inline coding it *does* result in greater code integrity. Put another way, the code for fetching is in only one place so only has to be fixed in one place.

Insert PROCEDURE (BYTE HandleError) ,BYTE ,PRIVATE

Insert and TryInsert are just interface maps of this procedure. InsertServer is a fairly good illustration of the difference between the FileManager and the file driver equivalents. It is only *really* trying to do an add; all of the other code is there either to handle errors or to ensure *other* ABC methods get called as appropriate.

First UseFile is called. This registers that not only is this file logically open, it needs to be actually open. Then comes a call to ValidateRecord. If the record is *not* valid then the method returns. Note throughout ABC the fact that Level:Benign is zero is assumed to aid readability and brevity. There are then three cases:

- 1) No autoincrement keys. In this instance the record will not already exist so a new one can be added
- 2) There are autoincrement keys and the autoincrement has been pre-primed. In this case the record does exist resulting in a PUT rather than ADD.
- 3) There are autoincrement keys but they have not been pre-primed. Call the autoincrement logic to ADD the record (remember PrimeAutoIncrement does *not* corrupt the record buffer other than the autoinc components themselves).

There are then three error conditions to worry about:

- 1) NoError. In this case simply note that any previously primed autoincrementing has now been used and return. (Note this code assumes that PrimeAutoIncrement will not have left a value in ErrorCode if it was successful; I suspect that technically this is a bug.)
- 2) DupKey. This is the only error the method attempts to recover from gracefully, stepping through the keys and alerting the end user of any duplications that this record causes.
- 3) Everything else. Post a general (cryptic) error message to the user and return.

NextServer PROCEDURE (BYTE HandleError,BYTE Prev) | ,BYTE ,PRIVATE

Again Next and TryNext are just interfaces to this method. Since C5EEA Previous and TryPrevious have also become interfaces to this routine (the beauty of this method

being private!). The `NextServer` and `PreviousServer` methods of earlier versions differed only in one line which has now been parameterized with the `Prev` byte.

There are only two real points of interest. Firstly the `BadRecErr` sets the `EOF` flag (see `GetEOF`). Secondly ABC has a facility whereby a held record error can be treated simply as a `Skip` rather than as an `EOF` (which it was in 2.003). You can argue back and forwards for hours as to whether it is better to display a browse with information missing or to abort the display. ABC takes the approach that that decision is best left in the hands of the developer (as the real answer is probably dataset specific) and so provides a property for her to register her decision.

```
Open PROCEDURE (BYTE HandleError, BYTE IncrementUsage=True, |
    BYTE ForceOpen=False), BYTE, PROC, PRIVATE
```

`Open` and `TryOpen` are interfaces to this method. The second and third parameters are really for `UseFile`. They allow an actual open to be forced without a logical open happening. Specifically, setting increment usage to false means that a corresponding close is not required. `ForceOpen` is used to force the file open even when the `LazyOpen` status would suggest the open can be deferred. Note that `ForceOpen` is quite safe as the routine takes a failure to open because the file is already open as a success. Again most of the work of the routine is in error handling, and in this case some moderately sophisticated recovery is allowed for.

- 1) `NoError` or `FileOpen`. Both treated as a success, internal state variables cleared (on this thread).
- 2) `RecordLimitError`. This code is just there for the evaluation edition. An attempt has been made to open the file in read/write mode with greater than the set number of records in the file. The code therefore sets the `OpenMode` to read-only and cycles (the loop will then cause another open attempt).
- 3) `NoAccessError`. Read-write access could not be acquired so the system tries to open the file in read-only mode (having first warned the end user)
- 4) `NoFile`. Provided the create mode has been set the routine will attempt to create the file, if that fails a fatal error is thrown.
- 5) `BadKeyErr`. For those file drivers with independent key files (notably Clarion) a corrupt key is non-fatal and the system will try to rebuild the keys so that processing can continue.

The `UNTIL 1` at the end of the loop means that any code falling through to the end of the loop will cause the loop to terminate. The `LOOP` is not a real loop; it is simply there to allow some of the recovery routines to attempt to open the file again without a `GOTO` statement. It should be noted that a `CYCLE` statement bypasses the loop *tail* termination condition but not the loop *top* termination condition.

Position PROCEDURE () , STRING

This method differs from the file driver equivalent in that it will issue a `UseFile` and if a primary key is available it will use that to form the position, or if there is no primary key it will use the file itself. In general (and increasingly) the ABC system assumes (and functions most efficiently providing) that all files in the system have a primary key. Note that this position string can only be used to perform a `TryReget`; it is not as general as the Clarion language `Position`. Whilst this functionality restriction does not give us much presently it will eventually allow extra efficiencies within the forthcoming `FileClass`.

TryFetch PROCEDURE (KEY K) , BYTE , PROC

`TryFetch` only really does a `UseFile` before passing control to the file system, although since C5EEA it has also performed a `SetKey` in debug mode purely to verify that the key passed in is valid for this file. (The ? is one of my favorite C5 features; it allows you to write code very cleanly which will not be executed when debug is turned off. This allows you to put in quite a few safety checks with zero overhead in the final shipping code.)

TryGet PROCEDURE (STRING Position) , BYTE , PROC

Perform a `Reget` from a string provided by `Position`. The `Reget/Position` pair give the `FileManager` user one extra piece of encapsulation: independence from key structure. Without them you need to know *from outside the class* how to uniquely identify a record. This illustrates an important aim: localizing information to reduce maintenance.

UpdateServer PROCEDURE (BYTE HandleError) , BYTE , PROC , PRIVATE

`Update` and `TryUpdate` are interfaces to this procedure. Clearly this is similar to `InsertServer`. The main extra comes from concurrency issues. ABC implements a technique called optimistic concurrency. Put simply, this means the algorithm assumes that no one else will ever change the record being edited locally, and then panics if they did. It relies upon a `WATCH` having been issued before the record (now being updated) was fetched. In standard ABC usage the `WATCH` is issued before the `view REGET` in the browse `UpdateViewRecord` method.

To handle this the code first takes the position of the current record, then it tries the `PUT`, if this returns a `RecordChangedErr` then the user is notified. (To help the end-user the form template passes in 2 as the `HandleError` value which prompts the use of a fairly verbose error message which tells the user of such things as the history key). Then the record saved by the other station is loaded (i.e. corrupting the local file buffer) and control is handed back.

UseFile PROCEDURE () , BYTE , PROC

The `UseFile` method is there simply to perform a real open (using `OpenServer`) if lazy open is currently on and the file is not open. This routine has one of the few bits of defensive coding in the whole of ABC; it actually preserves the file buffers across the Open

call just in case the file driver (which could be supplied by a third party) corrupts the file buffer upon the open.

Conclusion

I hope this chapter on the `FileManager` has helped you understand some of what we were aiming for (and have achieved) when we coded this class. It is one of our largest and most complex, and it also presently forms the base of what I call the *spine* of ABC:

`FileManager -> RelationManager -> ViewManager -> BrowseClass`

`FileManager` is also the class the developer most frequently needs to interact with (at least as much as `BrowseClass` and `WindowManager`). As such I believe an understanding of the principles involved will send you well on your way towards mastery of the ABC system.

ABC Database Class Design Notes

INSIDE ABC: THE RELATIONMANAGER

by David Bayliss

In the previous chapter I explained that the `FileManager` was logically there to embellish the underlying file drivers with information from the Clarion dictionary. The `RelationManager` class takes this dictionary embellishment one stage further to add the notion of related files. Currently there are three features this brings to the table:

- **Referential Integrity.** It is quite possible for a file to be physically correct, pass the file level validation constraints, and yet still not correctly relate to the other files. The `RelationManager` therefore duplicates a number of file access functions, and the use of the `RelationManager` versions of these functions ensures that the file is correctly linked to other related files.
- **File Unification.** This allows primary files which are linked to secondary children to be treated as a logical unity. This is a concept I occasionally refer to as BILF management (BILF stands for Bloomin' Irritating Little Files). A primary file could contain 100 fields, 10 of which are linked to children. Yet those child files don't actually *mean* anything; they are just created as part of the data normalization process. It is really ugly if every time you use a BILF you have to go throughout your code opening it, preserving it, etc. The

RelationManager therefore replicates some FileManager functions where the only service it performs is to perform the action upon all the related files in the tree.

- **Information provision.** Other parts of ABC sometimes need to know information about relations (notably linking fields and keys). The RelationManager provides a portable interface to this information.

Considerations

To some extent all the considerations mentioned in “Inside ABC: The FileManager,” p. 459 apply to the RelationManager, although less so. The RelationManager is built on top of the FileManager; specifically there is a one-to-one instance link between RelationManagers and FileManagers. As such the RelationManager always tries to use a FileManager function for a given activity if it can. This is not sheer laziness. By utilizing the FileManager, any overriding of the FileManager automatically works for code using the RelationManager.

There were a couple of new issues too. One was sheer complexity (and thus the need for safety). The legacy referential integrity (RI) code went through at least a couple of iterations and to this day it still falls over some cases and corrupts file buffers at will. For ABC we wanted an RI system that was rock solid, but also efficient. Legacy had another problem that for large dictionaries (especially heavily related ones) the code bloated horribly, and we wanted to reduce that drastically.

Further we wanted (in the future) to be able to extend the system to allow one-to-one and many-to-many relationships. Finally we wanted the RI code to simply drop away if it is handled by the back end (usually on an SQL database). That’s a pretty long shopping list!

As I head through the code overview I will warn you that the RI methods are by far the most complex procedures in the *whole* of ABC. They are an interesting example of my belief that you should isolate complexity. Don’t smear it throughout code (where everyone can stumble over it) but focus it into a small space that you can approach with caution. Well, here are six small procedures (the largest is 60 lines) that get the Bayliss classification of *ice pack jobs*. It is my job to make them clear enough that everyone (at least everyone who is prepared to try) can understand them. I hope I succeed. For the sake of brevity I shall assume that you have read the FieldPairsClass design documents (“Inside ABC: FieldPairsClass and BufferedPairsClass,” p. 447).

Coffee... Icepack Action (On the plus side, if you can handle this then you are over the ABC learning curve. From here it is just more, not harder).

I strongly urge you to have the source code to hand whilst going through this chapter; it really will make everything a bit clearer.

Initialization

The file drivers have no knowledge of the relationships provided in the dictionary; for this reason all the relation information has to be provided by the templates to the base classes. This is done by the templates overriding the `.Init` method and making a succession of `Addxxxxx` calls.

AddRelation PROCEDURE (RelationManager RM) , PROTECTED

A Clarion relation can be viewed from either end and it is not enforced that both directions have a key (although you do need a key both ways for RI). This `AddRelation` method is called when the file being initialized is related to the file being passed in but where there is no linking key on the file being passed in. You may prefer to look at this as saying “he is related to me.”

**AddRelation PROCEDURE (RelationManager RM, BYTE UpdateMode, |
BYTE DeleteMode, KEY His) , PROTECTED**

This method gives the ability to note a fully fledged relationship. The `RelationManager` passed in denotes the related file, `His` is the key you fill to get at his data. `UpdateMode` and `DeleteMode` specify the action to be taken upon a potential RI violation.

This `AddRelation` method has an interesting side effect: it primes the object to start accepting `AddRelationLink` method calls. There are OOP purists I know well (some I work with) who frown upon this kind of state within an object (the problem for the purists being that `AddRelation` *must* be called before `AddRelationLink`), but pragmatically it is efficient and encourages the object user to write readable code. What is actually happening is that this `AddRelation` creates a `BufferedPairsClass` which will then be filled with the linking fields of the relation.

AddRelationLink PROCEDURE (*? Left, *? Right) , PROTECTED

There are two other `AddRelationLink` functions besides this one, but the variations are simply there to save code size. (A `*?` parameter takes about 50 bytes of code to pass, `*LONG` parameters take four bytes, `*STRING` parameters take six. Given that `LONG` and `STRING` cover 90% of all linking fields this efficiency is worth having.) What is going on here is simple, but needs grasping. This method is called from the templates with something like:

```
Relate:File1.AddRelationLink(File1.KeyField1, |  
File2.KeyField1)
```

The *? parameter means the *address* of these fields is passed in and squirreled away for future use. Once this has been done for all the linking fields it is possible to assign from one set of linking fields to another using a single statement.

```
Init PROCEDURE(FileManager FM, BYTE UseLogout=0)
```

The base `Init` method simply ties in the `FileManager` this `RelationManager` is based upon. It also creates a queue for the relations and sets an internal property to denote whether transactions are to be framed within `LOGOUT/COMMIT` sections. Remember however that in template usage the `Init` method will typically be derived (in generated source) and the derived method will be full of calls to `AddRelation` to describe the dictionary fully within ABC.

If used fully this approach gives tremendous flexibility. It is quite possible to add files into the RI tree/or cut them out dependent upon system configuration. For example, you could have a file that is only shipped to certain customers but which is in an RI chain *if* it is shipped.

```
Kill PROCEDURE, VIRTUAL
```

This method simply steps through the relation queue, killing off any `FieldPairs` classes that have been created (for the `RelationLinks`) and then disposing them.

```
SetAlias PROCEDURE(RelationManager RM)
```

This method is used to specify that the current `RelationManager` is managing an alias (“Inside ABC: `FieldPairsClass` and `BufferedPairsClass`,” p. 447) of the passed in `RelationManager`. This method doesn’t really do anything; it is simply there to enable the `AliasFile` property to be private. I didn’t want the property public as I expect it to die when the `FileClass` comes along.

FileManager Replacements

These are substitutes for the `FileManager` equivalents. As such their basic semantics are the same. The difference is the related files are taken into account. For ease of explanation I am not tackling these in alphabetical order.

```
CancelAutoInc PROCEDURE(), BYTE, PROC, VIRTUAL
```

This method enables the form to readily tackle the problem of orphaned child records. (See “Inside ABC: The `FileManager`,” p. 459). The form can simply call the `RelationManager` equivalent (you should always consider `Relate:File.Thing`

as "Access:File.Thing(Taking into account related files)". The RelationManager calls down into the FileManager (passing in itself) to ensure children are taken care of.

Close PROCEDURE (BYTE Cascading=0) ,BYTE ,PROC ,VIRTUAL

This method simply issues a FileManager close on the current file, and all the child files, grandchild files etc. You would think this is quite easy, and in principle it is, but there is one little gotcha that makes the code quite complex. First consider the logical implementation. To Relate-Open file Fred you first open Fred then you open all of Fred's children. Then somehow you need to get the children to open their children.... Hang on, that's easy. Instead of opening Fred's children, you Relate-Open them and it all works. So a simple recursive solution would be:

```
RelationManager.Close PROCEDURE
I BYTE,AUTO
  CODE
    ASSERT(NOT SELF.Relations &= NULL)
    SELF.Me.Close()
    LOOP I = 1 TO RECORDS(SELF.Relations)
      GET(SELF.Relations,I)
      SELF.Relations.File.Close(1)
    END
```

Beautiful, elegant, efficient and liable to lock your machine the first time you try it. Imagine you have relationships A <—>> B <—>> C <—>>D and A <—>>D. Technically this is illegal in the Clarion paradigm (you need an alias for the second usage of D) but in practise you can *usually* get away with this (few procedures will have A, B, C and D all populated) and peoples dictionaries are littered with cyclic dependancies.

Now the recursive solution dies horribly. Suppose you close A. This closes B which closes C which closes D which closes A which closes B which closes.... You get the picture.

There are many sophisticated and elegant algorithms for detecting loops in graphs; we opted for a simple one. The idea is roughly this: when you get the first (top-most) call to close then you note the time. You then recurse as before but when you do the close you note inside the RelationManager the time you did the close. Then when you call a RelationManager to close it, you see if it has been closed since (or at) the top-most call. If it has then you have already been here before so you exit without recursing. You can actually implement this using CLOCK but there is one more little trick to spot. You don't have to use real time; any time will do. So for efficiency I made my own time stored in the Epoc variable. This time only ticks when the top-most call is made.

Here's a look at the code. First I check the cascading flag. This flag is purely there to indicate whether this is the "top" of the tree. If it is the top of the tree (cascading false) then I increment the epoc timer, if not then I check if for a touch in this "time-zone." If there has been a touch then the code returns; if not then I update the "last-touched" to prevent

further recursion. Then it is just a case of closing this file, and then stepping through the children closing them. One extra tweak is an *early out* mechanism. Essentially if any of the `FileManager.Close` calls fail the tree walk stops. This is not particularly useful in the `Close` case but in general a `FileManager` method returning an error could easily have put up an error message to the user. If that has happened once the last thing the user wants is to step through error messages for each of the 150 related files as well.

Open PROCEDURE (BYTE Cascading=0) ,BYTE ,PROC ,VIRTUAL

The `Open` code is actually very similar to `Close`. I'm surprised I didn't use a parameterized private method. Watch this space, as it is possible `Open` and `Close` will both have become shells for an `OpenCloseServer` by the time you read this. As an aside, I wonder if that seems unprofessional to you? Making mistakes, owning up to them and go fixing them? I never cease to be amazed by the people who write their code badly and then consider it inviolable. Encapsulation, a key feature of ABC, enables us to get the code right. Not *okay*, not *working* but *right*.

The one tweak is the `LazyOpen` mechanism. The `FileManager` has an attitude that says it won't actually open a file just because you asked it to. However we felt it is reasonable that the primary file *should* be opened straight away so if this is the top of the open call tree (and cascade is thus 0) we call `UseFile` to force the file open.

Delete PROCEDURE (BYTE Query=1) ,BYTE ,VIRTUAL ,PROC

This method is the first of the nasties. `Delete` is really just there to delete the primary record. There are two main complications: the first is the need to check that you can delete the primary record (i.e. there are no RI constraints), and the second is the need for transaction framing (the ability to abort the delete process halfway through if something goes wrong and you need to undo all the mess you made).

First is a fairly simply query as to whether or not the user actually wants this record deleted. One little trick is the use of the guard flag on the left hand side of the AND and the Throw on the right. This relies upon the fact that the compiler does short-circuit evaluation of logical conditions. In other words the compiler guarantees that if it knows the result of a logical expression simply by evaluating the left hand side then it will *not* evaluate the right. So if query is zero the Throw will not be done.

Next is the LOOP that operates the "Retry the delete?" message if the first attempt at deleting failed. Then the position of the record to be deleted is taken and is `TryFetched`. This is because the record needs to be full and accurate to allow the child links to be found and I cannot assume someone has made a record accurate just to delete it. Between the position and `TryFetch` is a block inside an `IF SELF.UseLogout`. This code is a horribly complex way of doing a simple thing. `LogoutDelete` (documented in part two of this chapter) simply finds out which files *may* be altered by this delete and adds them to the transaction frame.

Following this code is the main loop, which steps through all the relations calling DeleteSecondary for all files which are related with some form of constraint on the delete. (In C5 the LocalAction function filters out the RI done upon the server which does not require assistance from ABC). Note that DeleteSecondary is a method in the *related* RelationManager. This is a vital point! You do *not* go around deleting other RelationManagers' records; you ask them to do it for you. What gets passed in is the key of the His that this RelationManager is related to, the FieldPairsClass containing the list of linking fields, and the action mode to say whether restriction, cascading or deleting is called for.

How does this function work? From the perspective of the current RelationManager, the answer is “Don’t know, not my problem,” but it *does* matter that I know *if* it worked. If it didn’t I must stop processing myself. Note the little CheckError routine calls are pernicious: they can cause the whole method to be aborted. This code assumes the DeleteSecondary will have issued the ROLLBACK if required.

Assuming the children were okay then the RelationManager deletes its own record and handle any errors (including transaction rollbacks of child deletes if required).

Update PROCEDURE (BYTE FromForm=0) ,BYTE,VIRTUAL,PROC

The update code is very similar to the delete code so I’ll focus on the differences. There is no need for the “Are you sure?” query. There’s also no need for the Position/Reget as the code can assume someone doing an update has valid records in the buffer! Because updates cannot be restricted it’s okay to update the primary record before cascading to the children. Again any errors are handled.

NAME='Update'

The real interest (and new code) comes in the secondary loop. Note the call to EqualLeftBuffer. When an update is commenced in a form the RelationManager’s Save method is called which snapshots all of the values of the linking fields of the relations into the Buffer portion of the linking fields BufferedPairsClass. Thus at the update it’s possible to compare the left (primary) record with those stored values. If they haven’t changed (even if the record has) then there isn’t anything to cascade.

Suppose the cascade fails. Now there’s a primary record (in memory, the disk image will have been rolled-back) with linking fields that now *don’t* point to the children. Yuk! So upon failure the code copies the linking fields from the child back into the parent to tie the records together again.

Services

```
ListLinkingFields PROCEDURE (RelationManager Him,  
FieldPairsClass Trgt, BYTE RightFirst = 0)
```

This service routine provides the caller with a FieldPairsClass that has been filled with the linking fields of the two RelationManagers (SELF and Him). If RightFirst is zero then the Left of the FieldPairsClass will be filled with fields from the RelationManager denoted by SELF.

In a nice world this code would simply step through the relations, find the relation to Him and copy the Fields.List property into the target element by element (the inner loop). That is almost what happens. The complexity is that only one side of the relation actually stores the field list. So if the code finds that it has a suitable relation but doesn't have the field list it asks the related RelationManager to provide the list, but it has to switch the RightFirst parameter so that the Left/Right fields are correctly oriented in the result.

```
LogoutPrime PROCEDURE, BYTE, PRIVATE
```

This method is really just an error wrapper around a PROP:Logout assignment. This property is used as an alternative to the older LOGOUT(n, File1, File2, File3, File4...) procedure call. Using PROP:Logout you simply set the property true on all the files you wish to logout before issuing the LOGOUT statement. The advantage of this mechanism is that it removes the 52 file limit on the logout and also it means that the files to logout can be selected one by one rather than all needing to appear in one place. This is vital if files are to be switched in and out of the logout to support flexibility of referential integrity.

The code just checks that the file is open, then if logout is required the property assignment is done. An error of 0 means all went okay. An error of 80 means logout is not supported, in which case UseLogout is set to zero to prevent the error happening multiple times (this is a programmer error; there is no advantage to informing the user). Any other error is treated as ugly and the user is informed.

```
Save PROCEDURE, VIRTUAL
```

This method steps through all the relations and snap-shots the linking fields in the primary into the buffer component of the linking fields buffered pairs class. This is latter used as a sophisticated “record changed” tester for the Update Cascade code.

```
SetQuickScan PROCEDURE (BYTE On, |  
BYTE Propagate=Propagate:None) , VIRTUAL
```

This interesting little method is a variation upon the open/close theme. Essentially it just walks the relation chains using the Epoc to ensure it doesn't hit a cycle. Bit 080H of the propagate flag acts as the Cascading flag of Open/Close fame. The tweak is that this

time the caller can specify which type of relations are walked down: one-to-many, many-to-one or all. The work is done inside the loop, and the code uses the fact that a relation is considered many-to-one if this relation has a key to use for a lookup into the other RelationManager.

The Plug-Uglies

These routines are all extremely similar. Essentially they all do the same thing: they walk over the relation tree performing some action upon each RelationManager they encounter. Before attempting these make sure you are happy with the EpoC idea encountered in the Close method. These routines are also all private. They are just too prone to change and re-adjustment to have people rely upon them.

Note that these routines have been simplified in C5EEA (LogoutDeleteClear has been removed).

CascadeUpdates PROCEDURE ,BYTE ,PRIVATE

This is the simplest of the remaining routines so is probably a suitable juncture to explain what all of these routines are doing. CascadeUpdate is called when one or more fields in the record buffer for the current file have been changed. The cascading part of the job is to see if any of those fields that are changed are also linking fields to a child file. If they are then the corresponding fields have to be changed (or some other action) in the child file to keep the database consistent.

The code steps through all of the related files checking if there is an RI Update constraint (using LocalAction) and then checking to see if the linking fields of these two files have changed since the last call to save. If they have changed then the secondary is told to update itself. If the secondary is unable to do so (that is, an errorcode was returned) then the primary is modified to unchange the fields that were causing the problem. Assuming all the children were changed (if required) then the primary itself is modified to disk.

DeleteSecondary PROCEDURE (KEY MyKey ,| BufferedPairsClass Links ,BYTE Mode) ,BYTE ,PRIVATE ,VIRTUAL

DeleteSecondary starts by checking that the file is open, and then preserves the contents using SaveFile. This is important, because the person doing the delete expects the primary file to be touched, but may not even know about the secondary file(s). Those files must have their current contents preserved as well (the legacy templates don't bother so be very careful doing RI with legacy templates).

After the file is cleared, the LeftToRight assignment then fills in the linking fields in the child (i.e. this) file with the values from the parent file. Now usually the linking fields will be the whole of the key but it needn't be the case (you can have unassigned key

components), so this method uses `ClearKey` to clear down the remaining key components so that the following `SET(MyKey, MyKey)` picks up all matching records. (The file clear is not enough, for a descending key clearing down means clearing high!)

Next is a standard ABC sequential processing loop. If the `NEXT` throws a fatal error (end-of-file is only a notify) then it's necessary to abort the processing, rollback the transaction and get out fast. Well actually not that fast. The exit is always done via a standardized routine that restores the child file to its original state. In less dire circumstances the code checks for two things: hitting end-of-file; or reading a record that no longer matches the parent. In either case all the children have been dealt with so it's possible to return gracefully.

If there is a child then the action depends upon the RI action that has been specified.

- Restrict: In this case the parent cannot be deleted (because there are children) so the method throws an error, rolls back the transaction, and notifies the parent to abort its processing too.
- Clear: This specifies that in the child the link to the parent is blanked out. A simple `ClearRight` blanks the linking field. Now for the little twist. If you think about it, the children aren't being deleted; they're being modified. Instead of calling `DeleteSecondary`, the code calls `CascadeUpdates`.
- Cascade : This is one for the power hungry. The code is very similar to `CascadeUpdates`. It steps through the children telling them to apply the `DeleteSecondary` criteria to themselves. Assuming they all manage then it's okay to can delete the parent record with suitable grizzling if unsuccessful.

LogoutDelete PROCEDURE, BYTE, PRIVATE

This is a tree-walking algorithm in fairly pure form. Its function is to guess all of the files that are likely to be touched by the delete process. It does this by logging out itself, then stepping through each child with an RI Delete constraint. If the constraint is cascade then `LogoutDelete` is called recursively on the child file (there's no need to worry about cycles as delete constraints only go from 1->Many). If the constraint is clear then the call could be something else but this time it is the `LogoutUpdate` procedure.

LogoutUpdate PROCEDURE, BYTE, PRIVATE

This method is very similar to `LogoutDelete` except that it checks the **RI Update** fields rather than **RI Delete**. Both cascading and clearing count as modifications so the job can be done by recursing.

```
UpdateSecondary PROCEDURE (KEY MyKey, |  
    BufferedPairsClass Links, BYTE Mode), BYTE, PRIVATE, VIRTUAL
```

And finally, the real nasty one. That said this code is very similar in principle to DeleteSecondary (although I doubt I will ever common them up; the task would be just a little too scary). The main changes are:

- The primary record will already have been modified when this routine is called so it doesn't work to call AssignLeftToRight to fill in the linking fields. Instead the code uses AssignBufferToRight where the buffer has been set up by the preceding Save call of the parent.
- If there is a child record then immediately issue a Save call on SELF (the child). This is to preserve any linking fields to the children (the grandchildren of the original record)
- Restrict: As well as aborting if there is a restriction clause the parent record must be modified so that it still points to the children.
(AssignBufferToLeft)
- Both Clear and Cascade cases fall down into CascadeUpdates which then propagates the changes to the grandchildren.

Summary

So what have you learned? “Never try to read DAB’s code!” Well, possibly, but go and have a look at the thousands and thousands of lines of RI code that a decent dictionary generates in legacy templates. Then remember that ABC RI doesn’t corrupt your file buffers, can be extended to many-to-many or one-to-one, can cut files in/out of the chain at run time and provides strong BILF management for free!

In some ways the RelationManager heralds all that is good and bad about ABC. It is highly functional, highly efficient, extremely concise, extremely flexible and entirely impenetrable to the casual observer. This brings me to a phrase I used at the ‘97 DevCon which Steve Parker likes to dispute at all reasonable opportunities: “Don’t Know, Don’t Care.” One of the features of OOP is encapsulation, which means that all of this stuff is safely under wraps. You don’t need to understand any of the above, just call the function and have done with it. (You don’t even need to call the functions, the templates do this for you.)

Specifically, if you are working at the app level then this is how you should work. However, as my article “Clarion For Schizophrenics” (www.clarionmag.com/col/98-05-schizo.html) suggests, all good OOP programmers have a dual nature: one side doesn’t

ABC Database Class Design Notes

know or care, and the other understands the object and can extend it. These design documents are aimed squarely at this alter-ego. As such I trust I have provided it with some food for thought.

INSIDE ABC: THE VIEWMANAGER

by David Bayliss

Having dealt with the `FileManager` and `RelationManager` classes the final part of the file system I need to deal with is the `ViewManager`. It could be argued that having three different objects dealing with files is a little excessive. In a sense this is true; logically the `ViewManager` brings fairly little to the table over a `RelationManager` (it “just” deals with a bunch of files). However, the `ViewManager` does handle the very important special case where a bunch of files are being used to retrieve a series of records (including child lookups) in a nominated sequence. This was so vital I felt it deserved a special object. It turns out that the `ViewManager` is almost never used as a `ViewManager`, but it *is* the base class used for many of the higher level data access objects.

Room With A View

The `ViewManager` is to a `View` structure what a `FileManager` is to a `File` structure. Specifically, the `ViewManager` is there to act as an OOP front end to the view. It is also there to provide some logical sophistication not present in the native view. The main logical elements brought to the table are:

- 1) Range Limits: The ABC and legacy template chains have the notion of a range limit, which is the ability to restrict the records retrieved to those matching one or more of the major-most elements of the key order. Tied in with the notion of range limits is the notion of a free element (the major-most element of a sort order that is not range limited).
- 2) Multiple Sort Orders: A Clarion view structure only supports a single sort order (although the current sort order can be changed at will). The ViewManager is to support multiple sort orders (simply).
- 3) Flexible filters: The Clarion view structure has one assignable filter. The ViewManager provides for multiple filters active at once.
- 4) Attitude: A key requirement of ABC was that we wanted optimal (or near optimal) browse performance. It was felt that an essential element of that was ensuring that views were handled cleverly. Rather than us having to build the *cleverness* into every object that used the view (browse, drop down list, drop down combo, report, process etc) the ViewManager watches the commands being sent to the view structure and translates these commands (where required) into a more intelligent sequence.

Initialization

The initialization section of the ViewManager is quite large, and it also offends the OOP purists as it contains state; the order in which the methods are called is significant. These two are tied together. A significant procedure (say five browses and 15 drop combos) will contain 20 view initialization sequences and possibly 100 sort order creation sequences. We felt that having these sequences concise, readable and efficient was more important than sheer hygiene.

The call sequence is:

```
Init
[ Repeat 1 or more times
  AddSortOrder
  AppendOrder ! Optional
  AddRange    ! Optional
]
UseView
  AddRange PROCEDURE (*? Field)
  AddRange PROCEDURE (*? Field, *? Limit)
  AddRange PROCEDURE (*? Field, *? Low, *? High)
```

The three AddRange methods set a range limit upon the current sort order (defined by the preceding AddSortOrder or SetOrder). They correspond to Current Value, Single Value and High/Low Value range limits respectively. The code for each is similar. The aim

of the code is to produce a RangeLimit queue with a queue record defined for each element of the key that is range limited.

The first line of code sets the type of the range limit. The second calls LimitMajorComponents. It is defined in ABC that if the range-limited field of a sort order is not the most major component then all the more major components are implicitly current-value limited irrespective of the limit-type of the more minor component. The call to LimitMajorComponents implements this detail.

The field(s) passed in is then added to the range-limit queue. Finally SetFreeElement is called to compute the free element of the sort order.

```
AddRange PROCEDURE (*? Field,RelationManager  
MyFile,RelationManager RelatedFile)
```

The purpose of this AddRange is the same as the other three, but the code is somewhat different because the information required to construct the range limit is not publicly available (it is hidden in the RelationManager). Essentially if two files are related by keys with, say, three components, then a FileLimit range limit corresponds to a single-value limit upon three different elements! The queue is thus filled in using the ListLinkingFields capability of the RelationManager.

```
AddSortOrder PROCEDURE (<KEY K>) ,BYTE ,PROC
```

The AddSortOrder method actually performs the action of creating a new logical sort order. Each logical sort order can have a different range limit (and thus free element), and a different filter. This is stored in a queue (or Order queue) with a record pertaining to each sort order.

This method clears the queue record (it has to as it contains an ANY), stores the key, creates a new range-limit list and uses the first component of the key as the free element.

The sort order itself is computed by passing the key (if present) as a comma delimited list of components to the SetOrder method.

As it is impossible to remove sort orders it is okay to return the record number of the queue record added as the “unique identifier” of the sort order within the queue (for later use by SetOrder).

```
AppendOrder PROCEDURE (STRING Order)
```

Each time I look at the AddSortOrder/AppendOrder relationship I oscillate between deciding it is a beautifully elegant engineering solution and fretting that it is a complete hack. The issue is this: *Proper* database theory will tell you that sort orders should be defined in terms of fields and ascending/descending flags so that the logic of the program is nice and clean, and the ugliness of physical database design and actually getting performance out of the system can be left to some other poor schmuck.

The problem is that with many programmers coding furiously and requiring many different sort orders, the “poor schmuck” (typically the DBA) actually may not be able to get the system to perform at all! Worse yet he may be sufficiently senior that you can’t boss him around to make sure *your* program runs okay.

Thus the programmers have to get pragmatic and they start building keys into their program logic. Performance improves greatly; unfortunately applications have to use a restricted set of sort sequences or the number of keys explodes.

So along comes ABC. What do we do, “restrictive” or “slow”? Legacy took the restrictive approach, which I didn’t want, but the alternative wasn’t that nice either. What we settled on in ABC was an interface that *can* be used in a fully, logically pure way but that *encourages* the writing of efficient queries. It does this by defining a sort order in two parts. The first (optional) part is a key which defines the main part of the sort sequence. The second (also optional) part is the fields used to sort duplicates within the first key. Combining this with some special technology within the view driver we have the panacea of full flexibility that is usually fast.

To put some meat on the bones; assume you have an invoice file containing a customer id, and an invoice date (amongst other things). There is a key on `customerid`. You want to list invoices in customer order with invoices for a customer listed in date order. You do an `AddSortOrder(CustomerKey)` and then an `AppendOrder('INV:Dateordered')`. The view driver will then read in the records for the first customer, sort them, then the second customer etc. Given that sorting is at least an $N \log N$ process this “bucket sorting” can produce a massive time savings.

An additional tweak that should be available by C6 is that `AppendOrder` may start with a “*” character, meaning the specified order replaces the order specified by the key *after and including* the free element. This is useful for specifying range-limit information using a key but then ignoring any trailing key components.

Init PROCEDURE (VIEW V, RelationManager RM, <SortOrder SO>)

Much of the init code is straightforward; note though that the order property may be set up in one of two ways. If passed in then that version is used, otherwise one is created. This allows a derived class to construct a larger queue (more fields) than the `ViewManager` requires while still having the `ViewManager` perform the administration required.

The order queue is central to the whole `ViewManager` operation and stores all the information pertaining to a given sort order. When the `ViewManager` is derived by a `browse` it stores all the information for one tab of the `browse`.

Note also the default values provided to the view driver to enable the SQL buffering technology.

The Init method also ensures the UseView method is called, for reasons I'll discuss under that method name.

Kill PROCEDURE, VIRTUAL

This is one of those messy little procedures that only really exists to ensure that there aren't any memory leaks.

Essentially Kill just loops through the records of the order queue, and for each element of that queue it cycles through each element of the filter queue freeing up each filter element. Then it disposes of the filter and order-clause queue and nulls out the free element (which is an any).

Finally if the order queue needs freeing (meaning it wasn't passed in from outside) then it's disposed of. Finally the order queue is freed.

UseView PROCEDURE, PROTECTED

UseView has a simple task, to call UseFile on all the files a view references. (UseFile technology is explained fully in "Inside ABC: The FileManager," p. 459.) The requirement of calling UseFile is a little subtle.

Logically when the template uses a view (or a browse) it only knows about the primary file: the lookups are an implicit part of view functionality. But the implementation of the VIEW (within the Clarion language) has one or two legacy throwbacks. One of these is that the files have to be opened independently before the view will work. So the ViewManager "dresses" the view structure to tidy this up, making sure all of the underlying files have been used at an ABC level and are therefore likely to be open, thus satisfying the view.

A list of file references is available from the view driver itself. The FileManager reference is obtained from the internal lists so that UseFile can be called.

Application and Attitude

The following three methods really embody the vast bulk of "the smarts" within the ViewManager. Their job is to construct a filter and order clause to provide a record set equal to the one currently requested. They also have an alternative agenda, which is to avoid resetting the VIEW's order and filter clause unless absolutely required. This avoids busy work in the view driver.

ApplyFilter PROCEDURE, VIRTUAL

This function really does three separate jobs. Firstly it constructs a filter equivalent of any range limits provided; then it concatenates any filters provided; finally, it applies the filter to the view and monitors the error condition.

ABC Database Class Design Notes

The CASE statement is simply to branch between current, single and file range limits, all of which result in a filter of the form “field1 = xxx AND field2 = yyy” and the pair range limit which produces “field1 >= xxx and field1 <= yyy”.

For the common case the code sits in a loop for each element of the range limit queue. For each element the field name is extracted from the file manager, then CasedValue is called to compute the right hand side of the equals sign for the given condition. CasedValue allows for the right hand side being a string (in which case quotes are used), the key being case insensitive (in which case an UPPER will be present of the field name and should be present on the constant name) and even the value containing quotes!

The range filter assignment line has an interesting tweak, which is the use of an inline choose statement:

```
CHOOSE (I = 1, '' , ' AND ')
```

This solves the age old problem of having a list of items that you want separated (with an AND in this case). The traditional solution (using an IF statement before the concatenation) can make a simple loop look complex. The CHOOSE handles things in a very compact way.

The Pair code is a little more complex and illustrates nicely the treatment of key components which come before the component being range limited. If you look at the case within the loop the second branch is only taken for those major components. They are simply “=” limited, exactly the same as for a current value range limit. In other words, the standard ABC library deals with multi-component range limits *without any clever tricks being needed*.

The RRL-1th element is the lower bound of the range leaving the upper bound to be computed outside the loop.

The filters supplied by SetFilter are then appended in turn (there can be any number of them, each with a different ID). Each supplied filter is placed inside parenthesis to avoid any unexpected operation precedence problems.

Finally the filter is assigned and the error trapped.

ApplyOrder PROCEDURE, VIRTUAL

This method doesn't have any complexity. It just assigns the order clause and traps any errors.

ApplyRange PROCEDURE, VIRTUAL, BYTE, PROC

ApplyRange is where the system gets some attitude. The idea here is that sometimes the window manager knows “things have changed” and wants to alert the ViewManager that it needs to refresh itself. But the ViewManager shouldn't refresh itself if nothing of interest has actually changed. So ABC has the ApplyRange method.

For every range limit (other than current value) a mirror value is stored to reflect the state of the range limit the last time the browse was refreshed. When `ApplyRange` is called it checks the new values against those in the mirrors. If nothing has changed then `ApplyRange` simply returns; otherwise `ApplyFilter` is called to handle the changes in the data.

For the `Pair` case both the upper and lower bound have to be checked. This is done by comparing the right (which acts as a buffer) with the left. If there is a difference then left is assigned to right (for both bounds) and `ApplyFilter` is called.

There is a neat trick here: the `Single` clause is introduced by `OROF` not `OF`. This means that the `Pair` case will *also* fall down into this code.

The `File` case looks a bit more complex but it isn't. For the file range limit both left and right values are used (the left is the file being limited, the right is the file doing the limiting) so if the file doing the limiting has changed the new value is assigned to the buffer part of the `BufferedPairsClass` before the `ApplyFilter` is done.

Lights, Cameras ... Action

The following methods are really the ones that are called to *do things* to the view during normal operation of the `ViewManager`. They have names and semantics similar to equivalents in the underlying file managers and relation managers.

```
PrimeRecord PROCEDURE (BYTE SuppressClear = 0), |
    BYTE, PROC, VIRTUAL
```

The purpose of this method is to allow a record to be cleared/prepared for sending to an Update (or similar code) for insertion. This work is done in three stages,

- 1) Each file buffer connected to the view is cleared. This is done by quizzing the view driver for what the files are and then simply calling `Clear` for each record.
- 2) Next comes a bit of intelligence. Suppose you have a browse that is range-limited to only display addresses in Florida. Then assume an insert button is pressed: it is reasonable to assume the new blank record should comply with the Florida range limit. So the range limit elements are stepped through and the key components of the sort order are filled in with the corresponding limiting value.
- 3) Finally any remaining "blank bits" from the primary record are filled in (such as any auto-increment key components). This is done by a call down into the `PrimeRecord` method of the underlying file manager.

Close PROCEDURE ,VIRTUAL Open PROCEDURE ,VIRTUAL

These methods close and open the view respectively. They are extremely simple. The Opened flag allow them to be called on a “check if it is open” and “check if it is shut” basis without generating errors. The Open method also applies the presently active filters and order so that the static filter/order of the view (usually declared in the generated CLW file) doesn’t have a chance to load any records before they dynamic filter/order (which override the static one) is applied.

SetSort PROCEDURE (BYTE OrderNumber) ,BYTE ,VIRTUAL ,PROC

This method switches the presently active sort order. After this call further calls to (for example) SetFilter/Reset mean “perform this action on the present sort order.” This is one of the main benefits of this manager, as I detailed earlier.

The value returned means “did anything happen.” This is part of the attitude mechanism: if a zero is returned the caller knows it doesn’t have to progress with any code that would been required for a new sort order. For example, the browse watches this to see if a reload might be required.

Reset PROCEDURE (BYTE Locate) ,VIRTUAL

Reset PROCEDURE ,VIRTUAL

Reset on a ViewManager (and on any ABC object for that matter) means “bring this object up to speed with any external data it references.” In this context it means bring the view up to date with any data on the disk and position it relative to data in the underlying file record buffers. The Locate byte denotes how many leading components of the presently active order are taken into account when performing the locate. In other words, if Locate is zero (or you call the unparameterized form of Reset) no location is done and the view is positioned at the beginning of the record set. If locate is (say) 2 then the first two fields from the order clause are used to perform the position.

Next PROCEDURE ,VIRTUAL ,BYTE

Previous PROCEDURE ,VIRTUAL ,BYTE

Next and Previous read the next and previous records from the current data set. This is pretty much just a call to the underlying view. Additionally the error conditions are converted into ABC error levels.

If the record is valid according to the view criteria then the ValidateRecord method is called. This can specify one of Ok, Filtered and OutOfRange. The OK from ValidateRecord simply causes a return from the Next/Previous method with a Level:Benign error rating. An OutOfRange return is converted into a Level:Notify, which is interpreted by the rest of ABC as an “End Of File.” In other words it stops all view processing until the next reset. The Filtered case is handled by simple continuation. This causes the outer loop to cycle causing the next record to be read

from the view. In this way no “invalid” records will ever be returned from the Next/Previous methods.

ValidateRecord PROCEDURE ,BYTE ,VIRTUAL

This method doesn’t really belong in this section but it is so closely related to Next/Previous that I decided to cheat. The default implementation of ValidateRecord is useless - it just returns Record:Ok. It is here so that people can override it (by dropping embed code into the ValidateRecord embed point) and thus define an additional programmatic filter for what does and doesn’t constitute a valid record. ValidateRecord should return Record:Ok, Record:Filtered or Record:OutOfRange as defined above.

HouseKeeping

The remaining methods are really housekeeping and are designed to allow people to fiddle with the internal state of the ViewManager object without having to know its structure. As such they are generally simple to use and implement. They all require the previous use of SetSort which sets the sort order to be acted upon.

GetFreeElementPosition PROCEDURE ,BYTE ,PROTECTED ,VIRTUAL

This returns the position within the present sort order of the field which is the free element. You can think of this as the “number you need to pass to Reset to get a search using fields up to (and including) the free element.” For example, if you have a three component key, and have range-limited the first element, this function will return 2. Now if you have values in component one (the fixed element) and component two (the floating one) a Reset (2) will locate to the first record matching both of those elements (but ignoring the third).

GetFreeElementName PROCEDURE ,STRING ,VIRTUAL

This method returns the *bind name* of the free element, which allows derived classes to perform some kind of location/free element manipulation other than a simple reset. For example the filtered locator class uses this function to construct a filter of the form SUB(GetFreeElementName(),1,2) = 'FR' or similar resulting in SUB(CUS:FirstName,1,2) = 'FR'.

SetOrder PROCEDURE (STRING Order) ,VIRTUAL

SetOrder is directly equivalent to setting the order clause of the present sort order for the underlying ViewManager.

```
SetFilter PROCEDURE (STRING Filter) , VIRTUAL  
SetFilter PROCEDURE (STRING Filter, STRING Id) , VIRTUAL
```

These two methods manage the filtering system. What we were aiming for was to enable people to write additions to (for example) a browse that enabled additional filterings. The classic is a QBE filter or a multi-component locator. The traditional problem has been how to set a new filter without clobbering the one provided by somebody else. Using the one parameter `SetFilter` you can't; whatever you set overrides the filter provided by the filter prompt in the template. However the two parameter filter allows you to specify a unique identifier for the filter being passed in. Provided that identifier does not clash with anyone else's (be inventive, this *is* a string) then all the set filters will be concatenated before being sent to the view.

There is an implicit priority mechanism in that the strings are alpha-sorted before going to the view driver. This may be important efficiency-wise as many expression evaluators (including ours) perform left-to-right short-circuit evaluation of boolean conditions. (Put another way, if your doing something slow, put it last filter as the system *probably* will have thrown a record away before the code gets called).

Summary

Phew! We have finally gotten to the end of ABFILE.CLW. Kinda tiring, huh? Well yes, although it should be remembered that ABFILE is *by far* the largest module in the ABC system (45% bigger than ABBROWSE) and the relation manager is probably the most complex part of the system.

I trust that the tour has provided you some insight into the operation of “the spine” and also into the coding techniques, styles and methodologies employed.

That said, the number one lesson I *hope* you have gleaned is that all the code is there, it is all fairly approachable and once you understand it, it really does begin to make sense.

Honest.

Database Tips & Techniques

CLARION FILE ACCESS BASICS

by David Harms

Almost all Clarion applications work with data files of one description or another. Most use databases with numerous different tables and files, and perhaps even more than one database or file system. And all of this is ultimately possible because of Clarion's file access grammar.

From the start Clarion has had a set of simple, elegant functions for accessing data files. CDD introduced replaceable file drivers, which let developers switch from one file system to another without necessarily changing any code. Clarion for Windows added VIEWS, or logical files. Now in ABC there are a set of classes which wrap around the file access grammar and handle error checking and file opening/closing, and which for the most part obviate any need on the part of the developer to access the files directly.

Note that I said "for the most part." There are still times in ABC when you'll need to use a non-ABC file access function, and if you're a legacy code it's your only option. So although I'll touch on some ABC functionality, I'll mainly talk about the Clarion language statements that work with data files.

File Drivers And Caveats

Part of Clarion's appeal for database development is its system of replaceable file drivers. You can use one set of statements for file access, and just by switching the driver work with a completely different system. An application that updates dBase files, for instance, could also be made to work with a MS SQL database by changing drivers. But not all drivers support the same data types or the same functionality, and often maximum performance means using features specific to one driver. Even within a product like Clarion, code portability exacts a price especially when using SQL (although ABC handles this much better than legacy code does). Look in the Help for each database driver for the heading Supported Commands and Attributes for driver-specific information on functionality and datatypes.

Creating Files

Should you need to create data files, just make sure you check the **Enable File Creation** checkbox in the file's dictionary properties. This will add the CREATE attribute to the file's definition. CREATE causes the compiler/linker to store information such as the field and key names in the application, rather than just basic symbolic information, which is more compact.

For flat file systems such as the TopSpeed file format, you can usually just run your application and any missing data files will be CREATED. For SQL databases, you'll need to do this through database's administrative facilities, or using an SQL script, or perhaps the synchronizer (unless you're using ODBC, in which case the synchronizer isn't an option). Generally speaking when working with SQL databases it's better to import into Clarion than to try to use Clarion to create the database.

You can also use the CREATE *function* to explicitly create new files. The files must be *not* be open, and CREATE will delete any existing file before creating a new one.

Opening And Closing Files

Before you can use a file (or table, if you prefer SQL syntax), you have to open it. In Clarion, this is done with the OPEN function:

```
OPEN(file, access mode)
```

The `file` parameter is the label of the file, the name. When you define a file in your dictionary, the first field you fill in is called **Name**. This is a bit misleading. The dictionary name is actually the label and is what you'll refer to in your code when you want to do

something with the file. If you don't specify a particular name for the data file itself, the label will be used. If you have a file called Names which uses the TopSpeed file driver, the default name of the file will be Names.TPS. But you can also specify a different name for the file. If you use a variable for the file name (in the dictionary, precede the variable name with ! in the **Full Pathname** prompt) you can specify the file name, including the path if you wish, at runtime. You might do this to create different datasets in different directories, for instance.

The second parameter to OPEN is the file's access mode. There are two parts to this: the current user's access, and others' access. The following table shows the possible values:

User	Equate (from tplequ.clw)
Current User	ReadOnly EQUATE (0H)
	WriteOnly EQUATE (1H)
	ReadWrite EQUATE (2H)
Other Users	AnyAccess EQUATE (0H)
	DenyAll EQUATE (10H)
	DenyWrite EQUATE (20H)
	DenyRead EQUATE (30H)
	DenyNone EQUATE (40H)

The access mode is a combination of rights for the current user and other users. Typically a generated network (as opposed to single user) application will use a mode of 42h, or ReadWrite+DenyNone. File operations which require exclusive access will typically use 12h (ReadWrite+DenyAll) or 22h (ReadWrite+DenyWrite). The latter allows others to at least read the data during the operation.

You close a file with the CLOSE function:

```
CLOSE(file)
```

It's important to keep track of when a file is opened or closed. Consider a browse and form. The browse opens a file, and calls the form for updates. Perhaps you can assume that the form will always be called by this browse (a dangerous assumption). You have an OPEN(file) statement when the browse starts, and a CLOSE(file) statement when it ends. But what if that browse is called from another procedure that has a similar idea? The first procedure opens the file. The second (browse) procedure also opens the file, which has

no effect, since the file is open. But when the browse ends it will close the file, causing problems for the first procedure which still thinks it has an open file.

In legacy applications code is generated which keeps track of a global counter for the file. The following shows example opening and closing code, with `MyFile::Used` as the global counter:

```
IF MyFile::Used = 0
    CheckOpen(MyFile,1)
END
MyFile::Used += 1

! procedure code

MyFile::Used -= 1
IF MyFile::Used = 0 THEN CLOSE(MyFile).
```

The `CheckOpen` function, which is generated into the `appname_SF.CLW` file, does all necessary error checking and will create files if necessary and permitted. As the code shows, the file is only closed when the counter hits zero, which should mean that there are no procedures still needing the use of the file.

In ABC, the `Access:file` object, an instance of the `FileManager` class, takes care of the error handling, and you open a file using the class's `Open` method, and close it with `Close`. The `Open` method returns an error level, so you can do something like the following:

```
IF Access:file.Open() <> LEVEL:Fatal
    ! do something
    Access:file.Close()

END
```

In general, you shouldn't need to open or close files within a template-based application, and wherever possible you should let the templates handle this for you.

Accessing Data

Once you have a file or table in hand, you're ready to start reading data.

As the following table shows, just a few statements handle pretty much any kind of file access:

Function	Purpose
SET	Set processing order

NEXT	Read the next record
PREVIOUS	Read the previous record
GET	Get a specific record

The SET command, when used with files, has the following formats, as taken from the C5 help file (my notes in regular type and parentheses):

SET(file)	Specifies physical record order processing and positions to the beginning (SET...NEXT) or end (SET...PREVIOUS) of the file.
SET(file, key)	<i>Specifies physical record order processing and positions to the first record which contains values matching (or nearest to) the values in the component fields of the key. NOTE: This form is rarely used and is only useful if the file has been physically sorted in the key order. A common mistake is to use this form when SET(key,key) is the actual form desired. (This usually happens when a GET(file,key) is changed to a SET, but which should be SET(key,key)).</i>
SET(file, filepointer)	Specifies physical record order processing and positions to the filepointer record within the file.
SET(key)	Specifies keyed sequence processing and positions to the beginning (SET...NEXT) or end (SET...PREVIOUS) of the file in that sequence.

SET(key, key)	<i>Specifies keyed sequence processing and positions to the first or last record which contains values matching (or nearest to) the values in the component fields of the key. Both key parameters must be the same.</i>
SET(key, keypointer)	Specifies keyed sequence processing and positions to the keypointer record within the key.
SET(key, key, filepointer)	<i>Specifies keyed sequence processing and positions to a record which contains values matching (or nearest to) the values in the component fields of the key at the exact record number specified by filepointer. Both key parameters must be the same.</i>

A few forms of SET cover almost all situations. SET(key, key) is probably the most common form, since it supplies the records in key order, beginning with a record matching (or nearest to) a specified value in that key. That's exactly what a browse needs. For processing all records in a file, in key order, you probably want to use SET(key).

If you're going to be updating records and these updates will change key values, you may find that not all records are processed or some records are processed more than once. This is because you're changing the data used to determine the order of record retrieval. In such a situation, you have several options. You can use a key you're sure will not be affected; you can use SET(file) and process in record order without regard to keys; or (related to the first option) you can do a two-pass operation, store pointers to the affected records, and then retrieve each one individually via a unique ID on the second pass and update the values.

Retrieving Records With NEXT And PREVIOUS

It's important to remember that SET by itself doesn't retrieve any records; it simply prepares the file driver for a subsequent NEXT or PREVIOUS call. A situation encountered in countless browses is a user using a locator field.

The SET function works with whatever values are currently in the file buffer. In a list of names, if a user is searching for the last name of “Smith” in a file called NAMES, with a key on the LastName field, the file operations under the cover would go like this:

```
CLEAR(NAM:RECORD,-1)
NAM:LastName = 'SMITH'
SET(NAM:LastNameKey, NAM:LastNameKey)
LOOP
    NEXT(Names)
    ..

```

In reality this will be complicated a bit by some error checking code and the need to limit the retrieved records to the number of lines showing in the browse, but that's the general idea. The first time NEXT is called it will retrieve the first record matching 'SMITH', or the next record in the key if there is no 'SMITH'.

Note the use of the CLEAR statement before setting the key value. If you're using a single component key, you don't need to do a CLEAR (which is the example uses a -1 parameter to set all fields to their lowest possible values). If you don't use CLEAR on a multi-component key, you may find that you don't get the records you expect to get.

A similar technique is used to accomplish autoincrementing of unique record IDs. In an autoincrementing key each value must be unique. In legacy apps the template-generated code to get looks something like this:

```
SET(myf:IDKey)
PREVIOUS(MyFile)
IF ERRORCODE()
    NewID = 1
ELSE
    NewID = myf:ID + 1
END
```

Here the SET(key) format is used to return the highest existing value in the key. If no record is found, it's the first and the value is 1. Otherwise the value is the highest existing value plus one. Again, the actual code is a bit more complicated, as it needs to preserve any primed fields, and also will try up to three times to get a new ID, just in case someone else adds a record between the time the current user gets the highest ID and adds a placeholder record.

In ABC the autoincrement code (hidden inside abfile.clw) uses the SET(key, key) format to support multicomponent autoincrementing keys:

```
CLEAR(AutoIncField,1)
SET(SELF.Keys.Key,SELF.Keys.Key)
IF SELF.Keys.Fields.Ascend
    PREVIOUS(SELF.File)
ELSE
    NEXT(SELF.File)
END
```

Here the CLEAR function sets the autoincrement field to its highest value via the 1 parameter (different data types will have different maximum values - let CLEAR sort it out) and PREVIOUS/NEXT are called based on whether the key is ascending or descending. The usual error checking and incrementing of the key value follows.

Using GET

The GET statement isn't used nearly as much with files now as it was years ago (although it is commonly used with QUEUES, but that's another topic), because the implementation of GET, at least in the forms that use pointers, can vary widely from driver to driver. The syntax for GET is as follows:

GET(file, key)	<i>Gets the first record from the file (as listed in the key) which contains values matching the values in the component fields of the key.</i>
GET(file, filepointer, [length])	<i>Gets a record from the file based on the filepointer relative position within the file. If filepointer is zero, the current record pointer is cleared and no record is retrieved.</i>
GET(key, keypointer)	<i>Gets a record from the file based on the keypointer relative position within the key.</i>

The GET(file, key) syntax is the safest of the three because it is applied consistently among drivers that support this statement. The same cannot be said of the other two ways of using GET.

For instance, one very common usage of GET, back when most Clarion developers used Clarion DAT files, was with control files (see "Working With Control Files," p. 547). You could use the following syntax to retrieve the first record of a DAT file (called Control, in this example):

```
GET(Control,1)
! Test for error
```

This form of GET relies on the driver assigning an index value of 1 to the first record in the table. That is true of DAT files, but not of most other file types. You cannot use this exact

same code with TPS files, since the expected value in TPS files is a byte offset, not a record count. And the ODBC driver does not support this particular use of GET at all. In general, the preferred way to handle this situation is to issue the following statements:

```
SET(file)
NEXT(file)
```

or, if you're using ABC:

```
SET(file)
Access:file.Next()
```

POINTER vs. POSITION

GET, in its second and third forms, expects a numerical pointer to a position in a file or key. If you don't know that value ahead of time, as in the DAT example of `GET(control, 1)`, you'll need to obtain it with the `POINTER` function. But it's generally recommended that you use the newer `POSITION` function instead of `POINTER`. `POSITION` returns a positioning string, and can be used on a FILE, KEY, INDEX, VIEW, or even a QUEUE.

REGET and RESET

Once you have the `POSITION` information, you'll need some way to apply that to the current file/table/view. The `REGET` function retrieves the record according to the position data, and the `RESET` function sets file/table/view processing to that specified by the position data. In particular, `RESET` lets you process a file/table/view in one sequence, then interrupt processing to process in a different sequence, then restore the original sequence so you can pick up where you left off:

```
! Currently processing in ID order
PositionString = POSITION(MyFile)
! Prime the key values
SET(MyFile, MY:NameKey)
! Do some processing in Name order
RESET(MyFile,PositionString)
! Continue processing in ID order
```

Summary

Clarion offers a number of powerful ways of manipulating files, tables and views, using core language statements including `OPEN`, `CLOSE`, `SET`, `GET`, `NEXT`, `PREVIOUS`, `POINTER`, `POSITION`, `REGET`, and `RESET`. Although the templates can handle many, if not most, situations for you, and the ABC class library adds another useful layer in the `FileManager` objects, there are still times when you need to get down to the metal and work with the data in a more direct manner.

MANAGING TABLE OPENS IN ABC

by Jim Morgan

Clarion applications have automatically managed opening and closing tables for years. ABC applications continue this practice. The templates, under control of global template options, handle this task using `RelationManager` objects. So you don't have to worry about how or when your tables are opened, right? Wrong.

Clarion makes management of table opens and closes easy, but it takes a very conservative approach. In ABC applications, the templates generate `Relate:.Open` and `Relate:.Close` (if this table is related to other tables) or `Access:.UseTables` (if there are no related tables) for each table in your tables schematic for every procedure. This means that a form in a browse/form pair will always attempt to open a table already opened by the browse that called it.

A table open is one of the most expensive operations you can perform. Threaded tables need to be opened only once per thread. Internally Clarion maintains a use count on each table. The use count is increased with `tablename.Open` and decremented with `tablename.Close` and their related operations. Clarion only attempts to physically open a table when the use count is zero. However, the management of the use count involves some overhead. This overhead is not significant when it involves a small number of tables from a typical user event. However, repeated table opens inside a looping process

make for extremely poor performance. This means that if you write intensive processes to process data using structured procedure and functions calls, you need to optimize the file openings for good performance.

The amount of time that is takes to execute a single `RelationManager` open call (look for `Relate:tablename.Open` in the code) can vary greatly, and is usually in the range of .01 to 30 seconds. Table opening time depends on, but is not limited to, the following factors:

- the number of related tables as defined in the dictionary.
- whether the table is already open in the thread or in the application.
- whether lazy opens are turned on or off for the application. (This is seen as ‘Defer opening files until accessed’ in the application’s global file control extensions.)
- the table driver being used.
- whether a connection to the database is already established.

Improper table management creates GPFs, especially with SQL. Different table drivers behave differently on opens. With SQL-based tables you’re not physically opening the table; instead you’re asking the SQL server for the data. The driver first checks to see if there’s a connection to the server, and if there isn’t one, it establishes one. The first connection to the table for the user typically takes 0.1 to 0.35 seconds per table. With lazy opens turned off, 40+ tables could be opened on thread initiation without any visible display to the user, resulting in delays of over 15 seconds. Therefore, you do not want to reestablish a server connection every time a thread is launched.

Use an `Access:tablename.Open` call during startup to obtain the necessary information like user rights and validate connections instead of a `Relate:tablename.Open`. This allows the absolute minimum number of tables to be open and the program loads quickly. Once the user signs in and the menu is displayed, open the essential tables that are always accessed in the Frame’s thread. This allows subsequent threads to start up quickly without reestablishing the connection.

Different table drivers have different tolerance for poorly managed table opens. Topspeed tables can be opened once and closed a hundred times without problem. Topspeed tables can also be opened on a thread and not closed. If the thread terminates and the table is reopened, that’s no problem. With SQL tables, you will generally get a GPF in these situations. The underlying classes don’t protect the application programmer with a complete termination of the thread, so you have to be careful with the code you write. This means that if you get sloppy and forget to close a SQL table and terminate a thread, the next time you start the same thread ID and open the same table you will GPF.

Note: At Mitten Software, we turn lazy opens off because of significant problems that it created in earlier versions of Clarion. I would love to turn it on. I have not tested this problem with C5.5g. However, I have spoken to others who say the problems still exist, especially with SQL. If you are having success with lazy opens, please post a reader comment below.

Reviewing the table open management can be helpful when you are tuning an application or solving GPFs in SQL. There are four different types of information you should track:

- 1) How many times was the table opened in total?
- 2) Does the table show no use when the thread is closed?
- 3) What is the use count of the related tables?
- 4) What procedures were executed, and in what order?

Supplementing the base classes

You can track this information by changing the Clarion base classes as we have done at Mitten. This will create future maintenance issues since your changes need to be reintroduced when the class is updated by Clarion. The changes to ABFile.clw are highlighted below. ABFile.clw has the source code to the FileManager and RelationManager classes. The FileManager class has most of the single table methods. The RelationManager class has most of the table methods involving related tables. All of the changes have a signature of the developer's initials and the date of the change. If you follow this practice you can compare the source code and quickly apply any future updates.

```
MEMBER
INCLUDE ('TraceOpt.clw') !JM 6/21/01
```

The include statement controls the behavior of the table debug settings. When you want to enable the debugging, just place a TraceOpt.clw file with TraceFiles, TraceCloses, and TraceProcs switch settings in the application directory. The default TraceOpt.clw without debug is stored in the \Libsrc folder. The redirection file will look at the current folder before \libsrc for CLW files. The file contains the three trace switches and a name for the open/close trace file, as follows:

```
!Traceopt.clw
TraceFiles EQUATE(0)      !Standard Clarion switch,
                           ! non-zero to trace tables
TraceCloses EQUATE(0)     !Non-zero adds table opens
                           ! information to the debug table
TraceProcs EQUATE(0)      !A non-zero adds a procedure
                           ! trace to the debug table.
e_iniName Equate('c:\temp\fileclos.txt')
```

Database Tips & Techniques

An additional variable is conditionally added to the FileThreadQueue in ABTable.clw, but with the compile directive to eliminate all overhead in the final release.

```
FileThreadQueue    QUEUE,TYPE      ! QUEUE of status of
                           ! all table buffers
Id                 SIGNED         ! Thread number
Used               BYTE           ! Set True when table is
                                ! actually opened
Opened             USHORT          ! Table opened counter
COMPILE('xxx',TraceCloses)      !jm 6/21/01
HardOpened         USHORT          ! This Table opened
                           ! counter !jm 6/21/01
                           !jm 6/21/01
xxx                BYTE           ! End of Table flag
AtEOF              BYTE           ! Auto-increment done or not
AutoIncDone        BYTE           flag
LastError          USHORT          ! Last error identifier
END
```

The FileManager.Construct method deletes the debug file every time the program loads. Treat the debug file as an INI file. This is easy to program and doesn't hurt performance too much, but you must be careful: if the debug file gets over 32K in size, the file will not be properly maintained.

```
FileManager.Construct PROCEDURE
CODE
COMPILE('xxx',TraceCloses)
Remove(e_IniName)
xxx
```

Next, change the FileManager.Close method to log soft and hard use counts.

```
FileManager.Close PROCEDURE
CODE
SELF.InClose += 1
SELF.SetThread
FileManager.NoteClose(SELF)
IF SELF.Info.Opened
  SELF.Info.Opened -= 1
  IF ~SELF.Info.Opened
    CLOSE(SELF.File)
    SELF.Info.Used=False
  END
  PUT(SELF.Info)
? ASSERT(~ERRORCODE(),
'Unable to store thread specific file information.')
COMPILE('xxx',TraceCloses)      !jm 6/21/01
If SELF.Info.Opened
  PUTINI('SoftCount', Thread() & NAME(SELF.File),
        SELF.Info.Opened, e_IniName)
Else
  PUTINI('SoftCount', Thread() & NAME(SELF.File), ,
        e_IniName)
```

Managing Table Opens In ABC

```
        End
Else
    PUTINI('SoftCount', Thread() & NAME(SELF.File),
           -1, e_IniName)
xxx                                !jm 6/21/01
END
COMPILE('xxx',TraceCloses)          !jm 6/21/01
If SELF.Info.HardOpened
    PUTINI('HardCount', Thread() & NAME(SELF.File),
            SELF.Info.HardOpened, e_IniName)
Else
    PUTINI('HardCount', Thread() & NAME(SELF.File),
            , e_IniName)
End
xxx                                !jm 6/21/01
SELF.InClose -= 1
RETURN Level:Benign
```

The FileManager.OpenServer handles the actual table opening; tweak it to log use counts:

```
FileManager.OpenServer PROCEDURE(BYTE HandleError,
                                  BYTE ForceOpen)
RVal BYTE,AUTO
CODE
    SELF.SetThread
    FilesManager.NoteOpen(SELF)
    COMPILE('***',Traces)
    IF TraceFiles
        FileTablesManager.Trace('Open'&CHOOSE(HandleError=1,
            '(Errors):',':')&SELF.GetName()))
    END
    ***
    SELF.BindFields
    IF ForceOpen OR ~SELF.LazyOpen AND ~SELF.Info.Opened
        RVal = SELF.OpenFile(HandleError)
        IF RVal
            RETURN RVal
        END
    ELSIF SELF.Info.Opened
        COMPILE('xxx',TraceCloses)      !jm 6/21/01
        PUTINI('HardOpen', Thread(),
                SELF.Info.Opened, e_IniName) !jm 6/21/01
        xxx
    END
    SELF.Info.Opened += 1
    PUT(SELF.Info)
?    ASSERT(~ERRORCODE(),
        'Unable to store thread specific file information.')
        COMPILE('xxx',TraceCloses)      !jm 6/21/01
    If SELF.Info.HardOpened
```

Database Tips & Techniques

```
PUTINI('HardCount', Thread() & NAME(SELF.File),
        SELF.Info.HardOpened, e_IniName)
Else
    PUTINI('HardCount', Thread() & NAME(SELF.File),
            , e_IniName)
End
If SELF.Info.Opened
    PUTINI('SoftCount', Thread() & NAME(SELF.File),
           SELF.Info.Opened, e_IniName)
Else
    PUTINI('SoftCount', Thread() & NAME(SELF.File),
            , e_IniName)
End
xxx                                !jm 6/21/01
RETURN Level:Benign
```

The RelationManager.OpenCloseServer procedure is tweaked to track times opened here and maintain the use count.

```
RelationManager.OpenCloseServer PROCEDURE(
    BYTE Cascading,BYTE Opening)
I   BYTE(1)
Res BYTE,AUTO
    COMPILE('xxx',TraceCloses)      !jm 6/21/01
MyCount  Long          !jm 6/21/01
          Xxx          !jm 6/21/01
CODE
    IF Cascading
        IF SELF.LastTouched = Epoc
            RETURN Level:Benign
        END
    ELSE
        Epoc += 1
    END
    SELF.LastTouched = Epoc
?    ASSERT(NOT SELF.Relations &= NULL,
         'Relation manager incorrectly initialized.')
    IF Opening
        Res = SELF.Me.Open()
        COMPILE('xxx',TraceCloses)      !jm 6/21/01
        IF Not Cascading
            Self.Me.SetThread
            Self.Me.Info.HardOpened += 1 ! File opened counter !jm
            PUT(SELF.Me.Info)
            MyCount = GETINI('TimesOpened', Thread()
                & NAME(SELF.Me.File), 0 , e_IniName) + 1
            PUTINI('TimesOpened', Thread() & NAME(SELF.Me.File)
                    , MyCount, e_IniName)
        End
        Xxx                                !jm 6/21/01
```

Managing Table Opens In ABC

```
IF ~Cascading THEN Res=SELF.Me.UseFile().
ELSE
  Res = SELF.Me.Close()
  COMPILE('xxx',TraceCloses)      !jm 6/21/01
  IF Not Cascading
    Self.Me.SetThread
    Self.Me.Info.HardOpened -= 1  !File opened counter !jm
    PUT(SELF.Me.Info)
  END
  Xxx                                !jm 6/21/01
END
LOOP UNTIL Res
GET(SELF.Relations,I)
IF ERRORCODE()
  BREAK
END
IF Opening
  Res = SELF.Relations.File.Open(1) ! Use
  'public' interface to pick up VIRTUAL ness
ELSE
  Res = SELF.Relations.File.Close(1)
END
I += 1
END
RETURN Res
```

You'll need to make some changes ABError.clw to trace the procedures. Add the include statement to check the debug equate settings and a few variables.

```
MEMBER
INCLUDE ('ABERROR.INC'),ONCE
INCLUDE ('ABERROR.TRN'),ONCE
INCLUDE ('TraceOpt.clw')          !jm 6/21/01
COMPILE('xxx',TraceProcs)        !jm 6/21/01

MyProcedureName     CString(51)  !Temp Storage for the last
                      !procedure recorded in ini file.
TraceNumber         Short
xxx
```

Enhance the SetProcedureName to log procedures. This method is called automatically for most template-generated procedures. You can add the statement

```
GlobalErrors.SetProcedureName('ProcedureName')
```

in source procedure init and "GlobalErrors.SetProcedureName" in procedure kill to clear it.

```
ErrorClass.SetProcedureName PROCEDURE(<STRING S>
  CODE
  IF OMITTED(2)
    IF SELF.GetProcedureName()
      DELETE(SELF.ProcNames)
    END
  ELSE
    SELF.ProcNames.Name = CLIP(S)
```

```
SELF.Procnames.Thread = THREAD()
    COMPILE('xxx',TraceProcs)          !jm 6/21/01
IF SELF.ProcNames.Name
    MyProcedureName = GETINI('LastProc',
        SELF.Procnames.Thread, , e_IniName)
    IF MyProcedureName <> SELF.ProcNames.Name
        PUTINI('LastProc',SELF.Procnames.Thread,
            SELF.ProcNames.Name, e_IniName)
        TraceNumber = GETINI('TraceNumber',
            SELF.Procnames.Thread, , e_IniName) + 1
        PUTINI('TraceNumber',SELF.Procnames.Thread,
            TraceNumber, e_IniName)
        PUTINI(SELF.Procnames.Thread & 'Trace',TraceNumber,
            SELF.ProcNames.Name, e_IniName)
    End! IF MyProcedureName <> SELF.ProcNames.Name
End! IF SELF.ProcNames.Name
    XXX                                !jm 6/21/01
```

Reviewing the debug file

The resulting debug file is shown below. It is divided into several sections depending on use, and within each section are variables that show the thread number and table or procedure name. [SoftCount] is the current use count and is incremented every time a table is opened directly or indirectly through a relationship. [HardCount] is the current use count showing direct opens only. [TimesOpened] counts every time a table is opened directly. [LastProc] is the last procedure to register. [TraceNumber] is used for internal counting. [nTrace] is the procedure trace by n Thread.

```
[SoftCount]
1N:\TenaSQL\SLConfig.Cfg=1      Current Use Count is 1 in Thread #1
1N:\TenaSQL\Machine.Cfg=1
1dbo.Contacts=1
1dbo.PSNote=1
2N:\TenaSQL\SLConfig.Cfg=1
2dbo.Cases=27                    Current Use Count is 27 in Thread #2
2dbo.Followup=27
2dbo.Answers=27
[HardCount]
1N:\TenaSQL\Machine.Cfg=2
1SLMul_=1
2N:\TenaSQL\SLConfig.Cfg=2
2dbo.Status=1
[TimesOpened]
1N:\TenaSQL\SSProg_.tps=1
1N:\TenaSQL\Machine.Cfg=6
2dbo.Tables=15
2dbo.TokenList=28
15 physical Opens in Thread #2
```

```
2dbo.QNotes=35          35 physical Opens in Thread #2,  
wasteful!  
[LastProc]  
1>Main                Main is last proc called in Thread #1  
2=LoadTokenList  
[TraceNumber]  
1=2                  2 procedures traced in Thread #1  
2=14
```

Analyzing the results

The [softcount] section is occasionally useful. If the numbers for a series of related tables is not the same, it indicates that an Access:.Close was used instead of a Relate:Tablename.Close. The [HardCount] section should be empty upon thread closure. If any SQL table shows a use, you will get a GPF the next time that thread is started and the table is opened. The [TimesOpened] counts should be modest. If a table has a high count, the table should be opened at a higher level on the thread. [LastProc] should tell you approximately, where a GPF occurred. [nTrace] will show you how you got there.

Summary

Tracing your use of tables is extremely helpful for debugging and tuning performance. Using the above techniques you can make your applications and threads load faster, your processes run faster and your applications more stable.

CREATING ODBC DATA SOURCES AT RUNTIME

by Jon Waterhouse

One of the drivers that comes with Clarion is the ODBC driver. Although Clarion deals with most of the problems of translating your file access code (e.g. OPEN, CLOSE, NEXT) into calls to the particular ODBC driver that looks after your data file, there is one area where Clarion ignores a potentially useful set of features of the ODBC design. These are the administration functions, which are required before you can access any data source through ODBC.

When you are using most of the file drivers, all you need to know is the file name. When you use ODBC, however, you specify the “file” as a “Data Source Name” (DSN). The DSN contains the information that points to a specific file or directory. In this chapter I’ll show you how to use the administration functions to create an ODBC DSN at runtime.

The problem

Imagine you have been sent one Access database for each of thirty towns, where all of the databases have the same structure. Your job is to amalgamate all of the data into one big file. If you were dealing with flat files you would probably just loop through all of the files you had to deal with. With the ODBC connection you have to have a DSN set up for each file you want to use. This means one of two things; either you manually set up (in ODBC sources in the Control Panel) all of your DSNs before you start, or you create the DSNs dynamically as you need them. As a long-term strategy, the second solution definitely sounds better to me.

There are of course, several ways to skin this cat. In ODBC each driver presents a standardized interface for a particular data source. The ODBC DLLs that come with Windows deal with adding new ODBC drivers, and pass data requests to the relevant driver. You therefore have the choice of talking to the Windows ODBC manager, or directly to the ODBC driver for the file you are interested in (assuming you have documentation for that driver). The Clarion ODBC interface approach is to talk to the ODBC manager.

Note: There is an ODBC back-end driver for TopSpeed files, but that's not what I'm talking about here.

The main procedure in the ODBC admin DLL that deals with setting up DSNs is called `SQLConfigDataSource`. This procedure calls a procedure in each particular ODBC driver called `ConfigDSN`. I'll look at calling `SQLConfigDataSource`, because it is more general than `ConfigDSN`, and will work whether your data source is Access, SQL Server or any other database with an ODBC interface.

The documentation for `SQLConfigDataSource` gives the prototype as:

```
BOOL SQLConfigDataSource (HWND hwndparent, WORD frequest  
    ,LPCSTR lpszdriver ,LPCSTR lpszAttributes)
```

In Clarion this translates into:

```
SQLConfigDataSource PROCEDURE (ULONG ParentWindow,  
    USHORT Request,*CSTRING DriverString,  
    *CSTRING AttributeString),BYTE,RAW,PASCAL
```

In general, when using non-Clarion Windows DLLs, you can rely on the following general rules:

- Handles are `ULONGs`
- A word is four bytes (a `USHORT` in Clarion), so a `DWORD` (double word) is eight bytes, or a `ULONG`

- LP as a prefix stands for long pointer. Pointers to various types of data in procedure calls are indicated by asterisks (*) in Clarion
- The RAW attribute means that strings and groups are passed without length information. In this example RAW is not strictly necessary, but it will most often be required when using external DLLs, so you might as well get in the habit of using it
- The PASCAL attribute means that procedure parameters are passed left to right on the stack, compared to C which passes them right to left on the stack, and the default calling convention used by Clarion, which is to pass parameters using registers. The PASCAL convention is used for all Windows API calls.

The first parameter to `SQLConfigDataSource` can either be NULL (in which case your activity will happen in the background without displaying a screen to the user), or you can pass it the handle of your procedure window (`O{PROP:Handle}`). The second parameter indicates what you want to do - add, change or delete a DSN. The documentation in the help file goes as far as telling you that valid values are `ODBC_ADD_DSN`, `ODBC_CONFIG_DSN` etc. You will have to look at the C header files that come with the Microsoft Data Access Components Software Developers KIT (MDAC SDK), available free from http://www.microsoft.com/data/download_260SDK.htm, to find out that the values corresponding to these equates (#defines in C) are 1, 2, etc.

The final two parameters are where you specify, respectively, the driver you want to add or configure, and details of your request. These details include what file you want attached to the DSN (for Access) and what DSN you want to give your data source . This attributes string can take several instructions of the form *Keyword=value*. The documentation suggests that each argument should be separated from the next by a NULL character, and the string terminated with a double NULL. In practice semi-colons seem to work just as well or better.

The basic steps to use this function in your application are:

- 1) Build a .lib file for the ODBCINST.DLL (which you will find in `Windows\system` (or `system32`))
- 2) Use Application|Insert Module|External DLL to add the .lib file you just created to your application
- 3) Add a Procedure to your application called `SQLConfigDataSource`. Specify that it is an external procedure, and type in the prototype as given above (starting with the first parenthesis)
- 4) In the procedure where you want to set up your new DSN create two local data fields, say `DriverString` and `AttribString`, as CSTRINGS. The

DriverString should be 33 characters, while the AttribString should be 255 characters.

- 5) Write the values you desire into your two CSTRINGS. In the driver string you should put the driver name exactly as it appears on the **Drivers** tab in the ODBC data sources control panel application (e.g. Microsoft Access Driver (*.mdb)). In the AttribString you need to put your keyword value pairs as described in ODBCJET.HLP file (for Microsoft data sources). The ODBC Setup Dialog Page is the most useful. For example, to set up a DSN for an Access MDB database file, you could use something like:

```
AttribString = 'DSN=MyDataSource;' |
    & 'DBQ='C:\my data\AccessData.mdb'''
```

Don't forget to double quotes where necessary.

- 6) Write a call in source, e.g.

```
retval =SQLConfigDataSource(0,1,DriverString,|
    AttribString)
```

- 7) Open your file (which has the DSN and any other needed values in the OWNER attribute) and use normally.

I suggested that you could use a scheme like this to process a whole bunch of similar files. In theory you could create a single DSN (say, temp), use it, make another call to SQLConfigDataSource to change the file the DSN points to (the DBQ= keyword value) and then read in your next file. In practice this doesn't work. If you reuse the DSN you will keep on being connected to the first data source. Therefore, you have to create a series of DSNs (temp1, temp2, etc.) and delete each (ODBC_DELETE_DSN=3) as you finish with it.

That's the basics. This scheme is demonstrated in the example program at the end of this chapter, which simply creates several DSNs on the fly.

The next step is to make the whole process of integration a little easier. Step 1 is required, because Clarion needs the .lib file in order to link in the ODBCINST.DLL. You can't do without Step 5, either; you have to specify the driver and the associated data source. However, steps 2,3,4 and 6 are potential candidates for a template; actually two templates. The first template is a global extension and declares the ODBC procedures in the global map. The second is a CODE template which declares the two local variables and prompts you for the driver and attribute strings. The templates can also wrap some extra code around the plain vanilla call to SQLConfigDataSource to get it to provide better error checking.

The SQLConfigDataSource procedure returns a 1 (success) or 0 (failure). There is also a function called SQLInstallerError that can be called to give more information

about errors. I've written a procedure called `CDSWrapper` that first calls `SQLConfigDataSource`, and then reports the particular error type if the function fails.

The global extension part of the template does four things:

- 1) Adds the ODBC library to the project
- 2) Adds the prototypes for `CDSWrapper` and all of the ODBC procedures to be used to the map
- 3) Declares a bunch of EQUATES
- 4) Generates the code for the `CDSWrapper` procedure

From a template writing perspective, the first two items are very useful things to know how to do. You can borrow from the techniques in this template to add outside libraries and global procedures to your applications.

Adding the library to the project is accomplished by the `#PROJECT` statement. This statement has the same effect as manually doing an Application|Insert module.

The `%CustomGlobalDeclarations` embed point is also the place to declare the `CDSWrapper` procedure and its prototype. This requires three lines of code.

```
#ADD(%CustomGlobalMapModule,%Application & '.clw')
#ADD(%CustomGlobalMapProcedure,'CDSWrapper')
SET(%CustomGlobalMapProcedurePrototype,
    (BYTE,BYTE,*CSTRING,*CSTRING),BYTE')
```

The first line says that the procedure is in the `%Application.clw` file. The second has the procedure's name, and the third declares the prototype. This is all that is needed to add the `CDSWrapper` procedure to the map, but the application will still need prototypes for all of the required procedures in the ODBC DLL. I have been lazy and hard-coded the name of the DLL(LIB) file. If I was a bit more professional about it I would extract the DLL name from the name of the LIB file, which is stored in the `%ODBClib` token.

The EQUATES are taken from the ODBC documentation that comes with the MDAC software developers kit. They are used in the `CDSWrapper` procedure. A number of the equates declare certain data types and handles to be equivalent to other data types. For example, `HENV` (an environment handle) is actually stored in a Clarion `ULONG`.

The `CDSWrapper` procedure is placed in the `%ProgramProcedures` embed point and will therefore show up at the end of your `%Application.clw` file. The procedure is a bit more elaborate than it really needs to be, but this should give you a good start if you decide you would like to use more of the administrative ODBC functions.

The first line of code:

```
whandle = CHOOSE(display,0{PROP:Handle},0)
```

sets the window handle to be either 0 (if the code template specified no display), or to the handle of the current window (`0 { PROP : handle }`). The next section of code is not really necessary. What it does is load the names of all of the available ODBC drivers on the machine into a queue. For reasons I don't claim to understand, this requires first setting up an "environment", which is what the calls to `SQLAllocHandle` and `SQLSetEnvAttr` do. The driver names are then retrieved in a loop that looks like this:

```
Get first driver
Loop
  If error
    Report error
    Break
  End !if
  Add driver to queue
  Get next driver
End ! loop
```

I mention this explicitly, because this type of loop structure is not very common in Clarion. This loop could be used to build a list of available drivers to be presented to the user.

The environment handle is then freed because it is not needed any more. The next section requires that at least one driver was found in the first part of the procedure. The driver name is checked against the list of drivers that are stored in the queue. If the driver is not there you are not going to be very successful setting up a data source that relies on that driver, so you get an error message and exit the procedure. If everything is still okay the procedure makes the call to `SQLConfigDataSource`. If `SQLConfigDataSource` returns an error, `SQLInstallerError` provides more information. `SQLInstallerError` does display a reasonably informative error message if you call `SQLConfigDataSource` with a non-existent driver name.

As I mentioned, the procedure is really more elaborate than it needs to be. The major value of the first part is that the `SQLDataSources` procedure takes exactly the same arguments as `SQLDrivers`. Thus if you want a list of all the currently available DSNs in a queue that you can display, a very modest adjustment to the first part of this procedure will give it to you. The `CDSWrapper` procedure returns either 0 (failure) or 1(success).

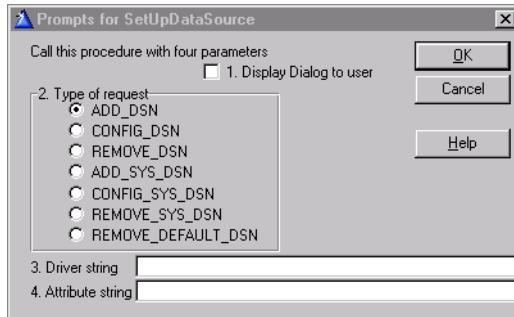
The second part of the template is a code template that can be easily added to a procedure to set up a DSN. It translates into three lines of code: one to set up the driver string (based on programmer input), one to set the attribute string, and a third to call `CDSWrapper` with the appropriate parameters. The template also sets up three data fields each time you add the template (a return value (`CDSretval`), a driver string and an attribute string).

The four parameters required are:

- 1) Display the dialog box or not
- 2) Type of action (add, delete, user DSN or system DSN)

- 3) The driver string
- 4) The attribute string

Figure 1: The SetupDataSource prompts



The example application does very little; it simply sets up two data sources. One is an Access data source (you will have to find an access (.mdb) file on your machine to reference in the DBQ field, which is set up without a dialog displayed). The other is a TopSpeed data source (for which you need the TopSpeed ODBC driver, included in CW5.5, but a separate purchase for earlier versions). This is called with a dialog displayed, mostly because it does not work otherwise. It returns a success code, but unfortunately does not set up the data source. I have version 2 of the ODBC driver, and this may be fixed in version 3.

Keep in mind that the template does not do the first step for you: you will need to make the .LIB file from ODBCINST.DLL before you use it. Also, the template can only set up data sources for drivers that are set up on the machine. If you don't have the TPS driver (read only) or the Microsoft Access driver on your machine, you'll run into errors, or you'll need to modify the example to work with data sources you do have.

Summary

Learning how to use the ODBC administrative functions has certainly made my life easier. The government department I work for has two applications that write to local databases; one database for each regional office. Every six months or so I get two CD-ROMs containing the data uploaded from each of the twenty or so offices. Previously I would have to manually set up forty DSNs (twenty offices times two data files) to read in the data to amalgamate into a single provincial data base. Now, I just have to provide my program with the names of the main directory and the subdirectories in which the data sits, and the program can create a temporary DSN for each office data base in turn.

For me, being able to use ODBC administrator functions just saves time. However, it is not hard to think of situations where you have to write a program that incorporates a legacy data source. If you can incorporate this data without having to train users to set up DSNs on their machines you will probably save yourself a lot of headaches.

Source code

See “Appendix A: Getting Support,” p. 601, for information on how to get the source accompanying this book.

- v3n5odbcadmin.zip

SECURING REMOTE DATABASE CONNECTIONS WITH SSH TUNNELING

by David Harms

I've mentioned several times on the newsgroups that I regularly access remote databases across the Internet. When I only had to do this intermittently I just used a plain, unencrypted connection, but now that I do so on a regular basis, I encrypt the connection with Secure Shell (SSH) tunneling. SSH is basically a secure Telnet, and SSH Tunneling is simply a way of piggybacking a database connection on that secure telnet connection. In this chapter I'll provide some background on the technique, and describe my implementation.

First, a little background. The SSH protocol was created by Finland's Tatu Ylonen as a safe replacement for the "remote" Unix commands, such as rlogin and rsh, which used unencrypted communication. There are both free and commercial implementations of SSH - the free stuff tends to run on Linux/Unix (e.g. <http://www.openssh.com>) and the commercial versions typically run on Windows or Linux/Unix (e.g. <http://www.ssh.com>).

To use SSH you need an SSH server program and an SSH client program, and of course they communicate using the SSH protocol. If you're connecting to a Linux server running PostgreSQL or MySQL, then there's a very good chance sshd, the SSH server, is already

running. If you don't have SSH on your server, then you'll need to investigate the available implementations for your server platform.

The protocol

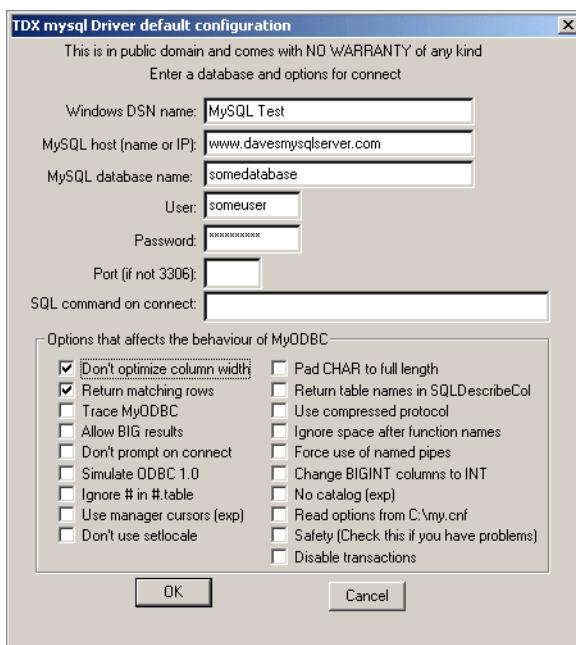
The current SSH protocol is 2.0; the 1.x protocols are still supported by most servers and clients, but for the best security you really should use the 2.0 protocol, and keep up with the latest patches.

As SSH stands for Secure Shell, the first and most obvious use of SSH is to let you securely run a command shell on a server, from a remote client, typically using port 22. Since I administer my servers, I use the command shell to do routine maintenance, run a local (to the server) database client for interactive queries, and so forth. It is also possible, however, to tunnel connections between other ports across the SSH connection, whether or not you use the shell capability.

Tunneling through SSH

If I want to connect to a MySQL server from a remote site, with or without SSH, I'll use the MyODBC (<http://www.mysql.com/>) driver. Figure 1 shows a typical MyODBC configuration window.

Figure 1: The MyODBC configuration dialog



Note that the port number defaults to 3306, the standard for MySQL servers. The connection between client and server takes place, in this example, across the Internet, between port 3306 on the server and whatever port happens to be handy on the client (but typically one of the unprivileged ports, i.e. higher than 1024).

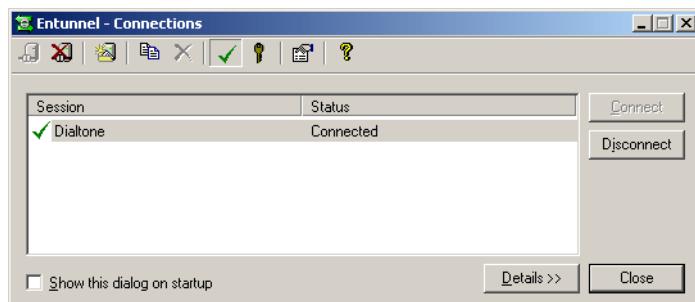
EnTunnel

To tunnel the MySQL connection through an SSH connection, I need to tell my SSH client the port numbers to use for the client end and the server end of the tunnel. Just about any SSH client has the ability to tunnel connections this way. For years I've used VanDyke's SecureCRT (<http://www.vandyke.com/products/secucrt/index.html>) SSH client to administer my Linux servers, and I've also used it for tunneling database connections. But

VanDyke has another, less expensive product called EnTunnel (<http://www.vandyke.com/products/entunnel/index.html>), which doesn't give you the shell capability but does handle the tunneling very nicely. This makes EnTunnel a good choice for deployment to your clients' machines - progressive discounts apply, so the more you buy, the cheaper the per-license cost, and in most cases you don't want to give your users the opportunity to run a command shell on the server anyway. Even if you don't use Entunnel, the following description will give you an idea of just how easy it is to use tunneling.

EnTunnel runs in the system tray - to configure it, double click on its system tray icon, or right-click and choose **Connections** from the context menu. You'll see a **Connections** window similar to that shown in Figure 2.

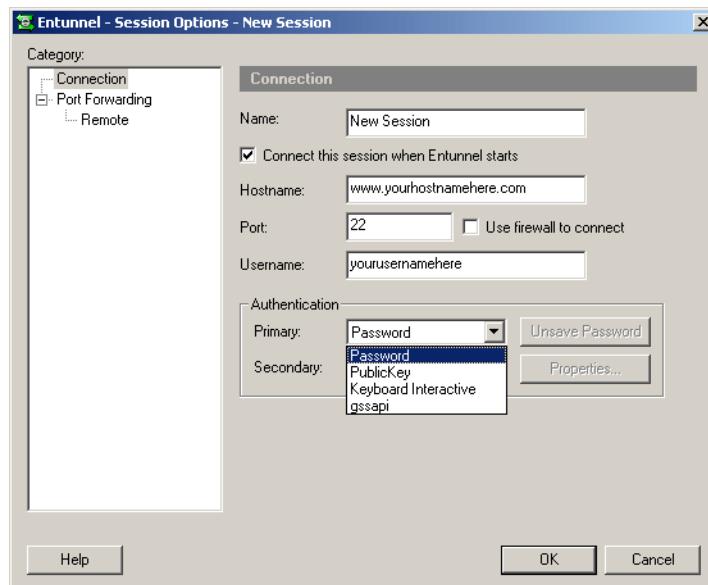
Figure 2: The EnTunnel Connections window, with one connection listed



Securing Remote Database Connections With SSH Tunneling

To create a new connection, click on the **Create a New Session** icon (third from left) or right-click in the list box and choose **New Session** from the context menu. You'll see the dialog shown in Figure 3.

Figure 3: The Create a New Session dialog



The session settings determine how you connect to the SSH server. You need to specify the host name or IP address, the user name you use to connect, and an authentication method. Note that authentication isn't the same thing as encryption - whichever method you use to authenticate yourself to the server, the connection is still encrypted.

If you choose password authentication, then all you need to connect is your username and password (and, of course, the name of the server). This is the lowest level of authentication - if someone can easily guess your username and password, then you don't have much security. Of course your SSH server has to be configured to allow password authentication, which may not be the case.

Public Key authentication involves a public/private key combination. These are small text files which, in combination, uniquely identify you. The idea is that you distribute a public key to the world at large, but you keep the private key all to yourself. Any message encrypted with the public key can only be decrypted by the private key. EnTunnel has a public key assistant (just click on the **Authentication Properties** button) which will create the key pair for you - you then upload the public key to the server. And as long as you carry your private key and some sort of SSH client with you (I suppose one of those USB "key" storage devices would be appropriate) you can connect to your server from anywhere.

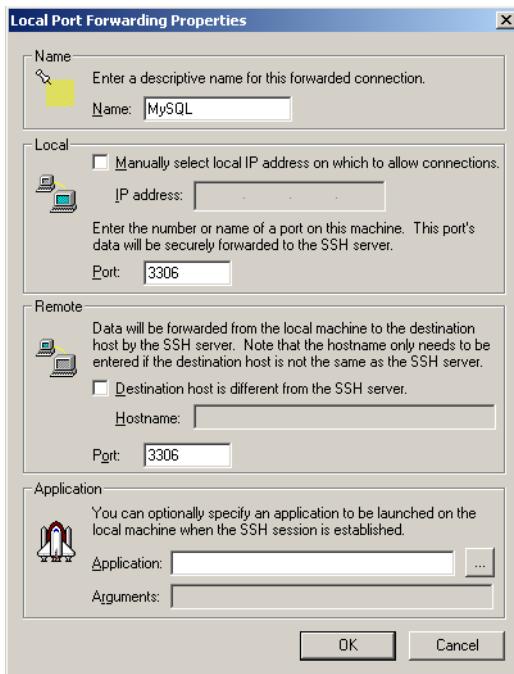
Keyboard interactive authentication simply means that you have the option to respond interactively to the server's request for authentication, and gssapi is Kerberos authentication, which I won't go into because I don't know anything about it!

Once you've established and saved your connection settings, you can click on the **Connect** button from the **Connections** window and make sure that you are, in fact, able to establish a session with the server. Of course you're not doing anything on the server yet, because you haven't told EnTunnel to forward any ports.

Forwarding ports

To forward a port, right-click on the session in the **Connections** window and choose Properties from the context menu. Select **Port Forwarding** in the **Category** tree, and click on the **Add** button for the **Local** (not Remote) **Port Forwarding** list. You'll see the dialog shown in Figure 4.

Figure 4: Forwarding a port

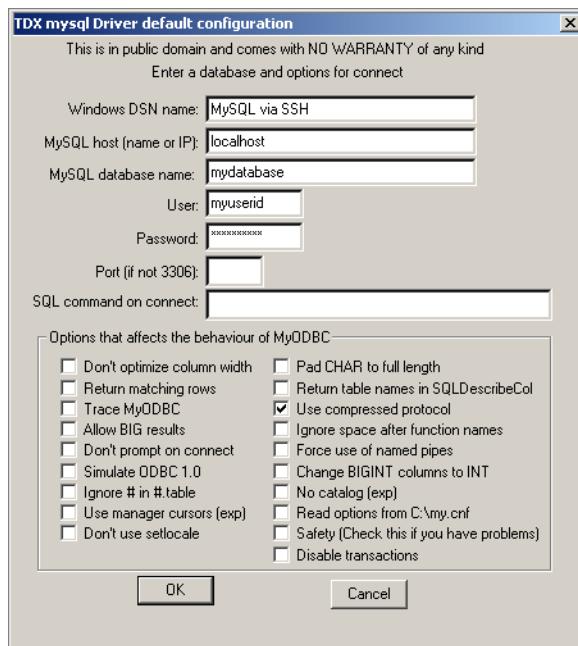


Choose a name for the forwarding settings, and the local and remote ports. In Figure 3 I've shown port 3306 (the MySQL default) as the port for the local and the remote sides of the

Securing Remote Database Connections With SSH Tunneling

connection. The critical one here is the remote side - that *must* be the port the MySQL server is listening on. The local port can be any valid number (although you may want to avoid the privileged ports), as long as the MyODBC data source is configured to talk to that port also. Figure 5 shows a MyODBC DSN configured to use the tunneled connection. In this example, I'm using 3306 on the local side as well, so I can leave the port number blank and use the default value.

Figure 5: Configuring a MyODBC DSN for tunneling



Fire up your application, and it will connect to the SSH client on the specified port. The client will encrypt the data and deliver it to the server, and vice versa. You're in business!

Summary

All you need to securely tunnel a database connection across the Internet is an SSH server (freely available on Linux machines), an SSH client, and an ODBC (or other) driver that can connect to a specified port on a specified server. You set up your ODBC data source to connect to a specified local port, and you tell your SSH client to forward that local port to a specified port on the server. You can also use port forwarding to encrypt email protocols and just about any other service with the (usual) exception of FTP.

Database Tips & Techniques

Port forwarding is an easy and effective way to encrypt database connections, and in many cases it's also the cheapest option available, especially if you're connecting to a database server on a Linux/Unix box and you can use one of the free SSH servers. I can also recommend EnTunnel as an effective and inexpensive tunneling client for Windows.

USING CLIENT-SIDE TRIGGERS IN CLARION 6

by Tom Giles

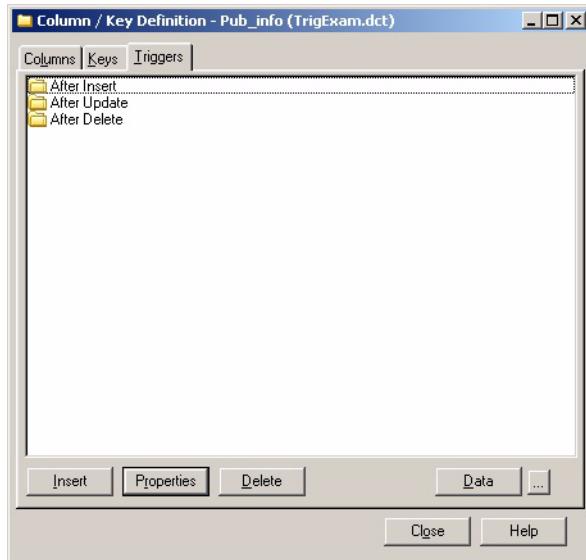
Client-side triggers are a new, very neat and useful feature that has been added to Clarion 6, both Professional and Enterprise versions. In addition, it is very easy to use. I guess you wouldn't expect any less since it is in Clarion. Client-side triggers are rules, added to the dictionary, expressed as Clarion code that is executed when a file (table) is accessed (unlike server-side triggers, which are SQL statements that execute on the server – see “Generating MS SQL Server Side Triggers,” p. 415). This code can run before and/or after the Add. You enter the code, say for error checking, field validation, computed values etc., and then specify when to invoke the code. At the specified time, your code will be automatically called.

There is an example included with Clarion 6. Documentation is sparse, really non-existent, but once you know the secret, client-side triggers are very easy to use. To use the demo, start Clarion 6, then click on the **Pick** icon, select the **Dictionary** tab, click on the **Open** button and navigate to the **Examples** folder. In the **Triggers** folder select and **Open** the TrigExam.Dct.

Database Tips & Techniques

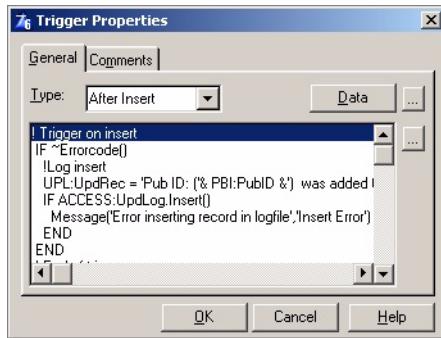
Highlight and double click on the `Pub_info` file in the **Tables** pane to bring up the records. Note there is a new **Triggers** tab on the **Column/Key Definition** screen.

Figure 1: The Triggers tab



Click on the **Triggers** tab, then on one of the three folders shown. A **Trigger Properties** form appears, as in Figure 2.

Figure 2: Trigger properties



On the **General** tab specify the timing using the **Type** drop down list. In the box below enter your code.

Click on the [...] button at the right to bring up the edit screen and enter any code desired, just as you would in any code embed point. There is also a **Data** button that will let you enter a new data field. The [...] button has the same function as in the procedure Data button – it exposes the actual declaration.

The **Data** section is for variables for your trigger code. These are separate from your normal **Data** fields in the **Properties** worksheet, which means you can use the same names. This is probably not a good idea unless you use a prefix, since it may be confusing to you later, but the compiler won't complain. I use a TRI:(gger) prefix for these to distinguish these variables from my LOC:(al), MOD:(ule) or GLO:(bal) variables. The TRI variables will not show up in the normal Module/Source files but instead in the ???bc0.clw files. This is because the triggers are generated as part of the file maintenance code, not the procedure code.

Another way to access client-side triggers is from the main Dictionary screen. Highlight the desired file, then click on menu item **Edit**, then **Triggers**, or highlight the file and press **Ctrl+Alt+T**. Either of these will take you directly to the triggers entry screen.

The main point to remember is the triggers are “triggered” on an **Insert**, **Update** or **Delete** file (table) action. Using triggers is similar to putting the code in the **Init** or **Kill** source code locations. One of my programs requires that I increment unique record identifiers manually, instead of using the Clarion autoincrement feature. Now I can add my incrementing code to the dictionary using triggers instead of having to add the code as a separate procedure routine. During certain operations I like to write what is happening to a log file, so this is also a potential trigger point. You can use triggers to update other files and do various calculations or other lookups. On an Invoice you might want a current amount due that must be calculated after reading part of payment file and summing all payments for that invoice. Certain referential integrity operations or complicated parent-child relationships can also be coded here.

You will find it takes a certain imagination and a little learning time before the full power of triggers will become apparent. Try them; they are quick, easy and powerful.

WORKING WITH CONTROL FILES

by Steven Parker & Nik Johnson

(Steve begins)

My very deep appreciation to Nik Johnson who bailed me out of control file purgatory (but you know what I really meant) when making the transition to ABC. The solution I'll demonstrate is his, as, of course, is his template which Nik explains in the second half of this chapter.

I've been using control files since... I can't remember when. When I began moving to ABC, I was confident. My experiences with the app converter had been by and large quite good. But the code that accessed and maintained my control files didn't work after conversion.

Consulting Richard Taylor's superb "Making the Transition to ABC" in the online help, I was able to make some adjustments to what the converter had done. Still, it didn't work.

Control files, clearly, had become a different breed of cat in ABC.

What Is A Control File? And Why Should I Use One?

A control file has two unique characteristics. First, a control file has only one record. Second, a control file has no keys (with only one record, a key is sort of pointless).

In many respects, control files duplicate the function of INI files. The important difference is that a control file, unless you use the ASCII or Basic driver (bad move), cannot be read with a text editor, is not so easily corrupted or manipulated as a text file and, should you so wish, can be encrypted. All or part of a standard file, which is what a control file is, can be made read-only.

The purposes for which you use or, indeed, whether you use a control file as opposed to an INI file is, of course, entirely up to you. But if you decide use a control file, *how* you use it is not.

The Issues

Just as with any file, there two kinds of things you will want to do. First, you'll want to write to the control file, adding and updating records... oops, *the* record. When adding, of course, you'll want to add *only* to an empty file. Adding a second record to a control file defeats its purpose. So, adding is a “one time thing.”

Second, you'll want access the record, i.e., read the file. Because a control file has only one record, by definition, and no keys, none of the standard template methods of file handling will work as expected. Indeed, they will not work at all.

Sequential processing is not possible (no “sequence,” you see). There is no key to prime. `Set(key)` and `Set(key, key)` have nothing to operate on. A loop, of course, is pointless. In plain English, the standard templates can do nothing for you except open and close the file.

Similarly, you can't simply add records to the file. A control file may have one and no more than one record. The standard templates will try to add multiple records.

The problem is determining (1) whether or not the file exists at runtime, (2) whether or not there is in fact a record in the file and (3) how to tell a file what you want to do (Add or Put).

(1) is usually only a problem the first time the app is run.

To make matters more interesting, the database driver that you use makes a difference in accessing single record files. TPS files, in particular, do not support the `Pointer()`

function for direct record retrieval while most other non-SQL databases do. Your choice of file systems will make a difference in your handling of control files.

Accessing Control Files

For the moment, let's assume that there is already a record in the control file.

If you want to display the user's company name and address in the header of a report or plug the city, state and postal code on data entry form, the data contained in a record must have been successfully read first. If you haven't first read the record:

```
CUS:City = CFG:City
```

isn't likely to give the desired result, is it?

In Clarion 2.0, or even in DOS, you would done something like the following:

```
Open(file)
Get(file, 1)
```

Take a moment to look at this code.

First, the file is opened. This does nothing but create a record buffer in memory. Nothing here prepares the file to be read. okay, it's not "nothing."

Second, normal file handling would follow with a `Next()` or `Set() / Next()`. But a control file contains only one record. If the driver fully supports `Pointer()`, the record can be accessed directly,

```
Get(file, 1)
```

The lesson? (1) Open, (2) read. Read = retain for later use.

TPS files don't fully support the `Pointer()` function (on a TPS file, `Pointer()` returns a valid pointer which can be used for direct retrieval but "1" is not a valid pointer with TPS files), so:

```
Open(Config, 42h) !or Share()
Set(Config)
Next(Config)
```

is required for TPS files.

Legacy command such as `Open()` and `Get()` can in fact be used in these circumstances even in ABC. Bad form, to be sure. But they will work. You must ensure that the file is closed at the appropriate point, if you `Open` the file directly.

In ABC, “good form” would be:

```
Access:Config.Open  
Set(Config)  
Access:Config.Next()
```

for TPS files.

Access:*file*.Open opens the file. Set() prepares the file for reading, as always. And Access.Next() reads the first (and in this case only) record, if any.

You will notice there is no ABC analog for the Get() command. Therefore, for file systems fully supporting Pointer(), you can continue using Get():

```
Access:Config.Open  
Get(Config,1)
```

With these file systems, xBase, Clarion, etc., I have had the Set() /Next() strategy fail. Thus, I continue using Get().

Maintaining Control Files

Since there is only one record, a browse is sort of... ah, pointless. Go directly to the form.

The problem is that a form needs to be told what to do. GlobalRequest is typically used to tell the form whether it is being call to add, update or delete a record. Without a legitimate value in GlobalRequest, the form won’t do anything.

Specifically, if called to add a new record, the form needs an empty buffer. If called to change or delete, the form needs a buffer with the correct record.

So, GlobalRequest needs to be set; what’s the big deal? The big deal is that you need to know whether or not the file has a record *before* you set GlobalRequest. If there is no record, GlobalRequest *must* be InsertRecord and if there is a record GlobalRequest *must* be ChangeRecord (you don’t ever want to delete a control record, do you?).

As it turns out, this is easily determined. If Relate:.Open returns a non-zero value, there was an error opening the file (e.g., the file does not exist and is not set up for create-if-not-found). If Access:.Next() returns a value, there is no record in the file. So the following code can be used to set up the call to a form:

```
! Listing 1  
IF NOT Relate:Config.Open()  
    SET(Config)  
    IF Access:Config.Next()  
        GlobalRequest = InsertRecord
```

```

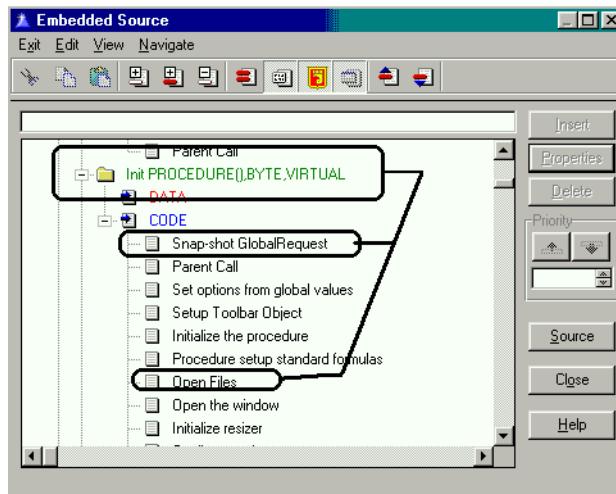
ELSE
    GlobalRequest = ChangeRecord
END
SetupForm
Relate:Config.Close
Else
    !action if file create not allowed
END

```

This ensures that GlobalRequest is always set properly.

Trying to determine the appropriate value for GlobalRequest from inside the Form procedure is a bit more difficult. It is more difficult because GlobalRequest is read before the file is opened (see Figure 1), early in the INIT method.

Figure 1: The Init method embed points



Of course, the code in Listing 1 work perfectly well before “Snap-shot GlobalRequest” (without the procedure call, of course) and not wreck havoc on the standard template code (so long as you close the file first).

To ensure that the configuration file is properly opened and read when the program is started, I use something like this:

```

! Listing 2
Loop
    Access:Config.Open
    Set(Config)
    If Access:Config.Next()
        SetupForm
    Else
        Break
    End

```

End

in the main procedure's INIT method. Notice that this code loops until there is no problem opening and reading the file. Combined with the code in Listing 1, the form always knows what is required of it.

Config files aren't especially difficult. But they do prove just how spoiled we are. Everywhere else, Clarion sets up the file buffer, accesses files and sets up forms. When there's only one record in the file, you're on your own as far as these go.

On the other hand, if you can manipulate a configuration file, you know how to access a file.

Nik and Steve's correspondence

```
> I have a single record (configuration) file and, based on what
> I did in CW2003, I used:
>
> Access:Config.Open
> Set(Config)
> Access:Config.Next()
>
> to get that record, but it doesn't seem to be returning the
> correct >field values. Could it be that the error is in my file
> access in the update form:
>
> Access:Config.Next()
> If ErrorCode()
>   If ErrorCode() = 35 then ThisWindow.Request = InsertRecord.
> Else
>   ThisWindow.Request = ChangeRecord
> End
> ThisWindow.OriginalRequest = ThisWindow.Request
>
> because I noticed (finally) that I had 15 or 16 records in the
> file. Ok, where'd I foul up? (Please...)
>
> TIA
>
> Steve Parker
```

Nik Johnson's reply

I think you have to be careful about timing here. What I would do is have a source procedure which sits between the menu (or whatever triggers the configuration file update process) and the update form:

```
LinkUpdate PROCEDURE  
  
    CODE  
    IF NOT Relate:Config.Open  
        SET(Config)  
        IF Access:Config.Next()  
            GlobalRequest = InsertRecord  
        ELSE  
            GlobalRequest = ChangeRecord  
        END  
        ConfigForm()  
        Relate:Config.Close  
    END  
    RETURN
```

This makes sure that the file is open, the record is current or the buffer cleared) and GlobalRequest is set -before- you ever get to the update form. In that way you don't have to worry about little things like the fact that the update window will internalize GlobalRequest before -it- opens the file. The update form at this point should see no difference between this call for a singular record and a request from a browse of many records. Objects are like people. They function best when given directions in a familiar context.

I (Nik) am an inveterate template tinkerer. It's a mixed blessing. By the time I abandoned my CPD 2.1 model file for the Logix Project Manager, there was more logic devoted to branching conditionally around various options than to perform the actual task at hand. By the time CDD 3007 rolled around, I had so much invested in modifying Todd Carlson's templates that I didn't dare switch to newer Clarion templates. But while they lasted these modified tools made me a lot more productive than I would otherwise have been.

I'm still tinkering, but the introduction of OOP and the ABC library has meant that I can now work *within* the TopSpeed framework rather than *around* it. Steve Parker's discussion of control files in the first half of this chapter provides an opportune way to illustrate the convenience and power of OOP/ABC while at the same time extending my toolkit.

Defining The Problem

What I want to build is a “set and forget” method of handling control files. As Steve points out, some file structures require a SET/NEXT approach, others do better with GET/POINTER. Still others may require other access methods. I want to spend an absolute minimum of time and effort incorporating control files into future applications, independent of the file system.

Steve has enumerated the things our tools need to be able to do:

- Open the file

- If the file is missing, optionally create it
- Read the file's only record in a way appropriate to the file structure
- If the file has no records, add one
- Update the record
- Close the file

The following narrative mimics the way I go about building tools for my own use. First, list the things that have to be done; second, establish a general design; third, build a skeleton; fourth, fill in the details. This works for a small shop, but if you're building tools for sale or working in a larger organization you may want to skew this pattern toward heavier documentation of the design in advance of coding.

This Wheel Has Already Been Invented ... Almost

The ABC FileManager class provides facilities to accomplish five of these six tasks, so it makes sense to start with that as a basis. A class based on FileManager can inherit all of its functionality and minimize the new work that needs to be done.

```
MyFileManager CLASS(FileManager)
Fetch      PROCEDURE, BYTE, PROC
          END
```

The new method provides a place to add Steve's access logic, but it doesn't include provision for specifying which version of that logic to use. ABC classes are insensitive to driver, but in this case that luxury is not available.

Two possibilities come to mind. First, the fetch algorithm could accept a flag to designate how access to the one and only record in the file is to be gained. A byte should be sufficient, since more than 255 variations on the get-only-record theme are unlikely. However, if the file driver changes, it could be inconvenient to chase down every Fetch and alter it. The second approach, adding a property to the derived file manager, permits the fetch algorithm to be specified once at the beginning of the program.

```
MyFileManager CLASS(FileManager)
FetchOnlyType  BYTE
Fetch          PROCEDURE, BYTE, PROC
          END
```

The compiler can distinguish between the new Fetch method and the one specified in the standard ABC FileManager class because their prototypes can be differentiated by the rules for procedure overloading. (See the Language Reference Manual.)

Building A Skeleton

A residence for record fetching logic having been established, the logic itself can be added.

First, set up an EQUATE for each value `MyFileManager.FetchOnlyType` can assume:

```
ITEMIZE, PRE(FetchMethod)
GetByPosition EQUATE
GetNext      EQUATE
END
```

Although these are the only two options at the moment, setting up this structure provides a clean way to add other options in the future.

The new Fetch method, by using the EQUATES, becomes self-documenting.

```
MyFileManager.Fetch PROCEDURE
ReturnValue BYTE, AUTO
  CODE
    ASSERT(SELF.FetchOnlyType)
    SELF.Open
    SELF.UseFile
    CASE SELF.FetchOnlyType
      OF FetchMethod:GetByPosition
        ! "get by position" code here ...
      OF FetchMethod:GetNext
        ! "get by set/next" code here ...
    END
    IF ReturnValue
      CLEAR(SELF.File)
      ReturnValue = SELF.Insert()
    END
    SELF.Close
  RETURN ReturnValue
```

This “shell” contains everything except the actual logic to access the configuration record if it exists. In adapting Steve’s logic, I’ve made a few assumptions based on my own expected use of the method:

- I may want to access the configuration record in a context other than an update form, so I’ve removed logic that refers to `SetupForm` from Steve’s code.
- I should never call this method unless I’ve set the `FetchOnlyType` property. The `ASSERT` protects against this. Since this would be a programming error as opposed to a data-related condition, I don’t need user-friendly error messages for this situation.
- If I ask for a record and can’t find one, I always want to add the record.

Your programming style may suggest a different set of design assumptions.

Putting Meat On The Bones

The only thing left to do is add appropriate code for accessing the configuration record. Whatever the method used, `ReturnValue` should be set to zero if the fetch is successful, something else if not. The CASE structure is the only part of the shell which changes:

```
CASE SELF.FetchOnlyType
    OF FetchMethod:GetByPosition
        GET(SELF.File,1)
        IF ERRORCODE()
            ReturnValue = Level:Notify
        ELSE
            ReturnValue = Level:Benign
        END
    OF FetchMethod:GetNext
        SET(SELF.File)
        ReturnValue = SELF.Next()
    END
```

The new method behaves very much like the standard ABC `Fetch` except that it doesn't require specification of a key, it expects one and only one record in the file, and, if the file is empty, it adds a cleared record.

Adding The New Class To The ABC Library

A little plagiarism is a wonderful thing. All the information needed to make making the new class look to Clarion like an ABC class is sitting in the `\LIBSRC` directory. First, set up files for the new class prototypes and methods. Call them something like `MyClasses.inc` and `MyClasses.clw`. Referring to similar files shipped with Clarion, set up these two files to match their style.

Here's the general setup of `MyClasses.inc`:

```
!ABCIncludeFile
 OMIT('__EndOfInclude__', MyClassesPresent__)
 _MyClassesPresent_ EQUATE(1)
     INCLUDE('ABFILE.INC'),ONCE
     ! class prototypes here ...
 __EndOfInclude__
```

The comment (`!ABCIncludeFile`) tells the IDE that this code follows the ABC pattern and should be treated as any other ABC include file. The `OMIT` structure lets this file be included anywhere the definitions are needed without fear of duplicating those definitions. For example, the definitions of the ABC `FileManager` are included above so that they can be used in the definition of the `MyFileManager` class.

The new `ONCE` attribute provides another mechanism for avoiding duplicate definitions, but that protection depends on the `ONCE` attribute appearing in every `INCLUDE`. Older

code may still require the protection provided by the OMIT structure, so it's a good idea to leave it in place.

A little more creative plagiarism provides the general setup of `MyClasses.clw`:

```
MEMBER
  _ABCDllMode_ EQUATE (0)
  _ABCLinkMode_ EQUATE (1)
  MAP
  END
  INCLUDE ('MyClasses.inc')
```

The MEMBER statement identifies this as a source module, something the compiler needs to know. The MAP structure is required, since it causes the compiler to include prototypes for Clarion language statements and functions. The INCLUDE of `MyClasses.inc` makes the new class definitions available to code in this module and, if the nest is not too deep (LRM page 95), also includes definitions from `ABFILE.INC`.

The two equates, for `_ABCDllMode_` and `_ABCLinkMode_`, implement the ABC library's method of determining where and how these methods will be linked and referenced. These definitions work with attributes in each CLASS statement which are required and will be added next.

In the skeleton version of the CLASS prototype, some necessary attributes were ignored. The full statement should have been:

```
MyFileManager CLASS (FileManager),
  TYPE ,
  MODULE ('MyClasses.clw'), |
  LINK ('MyClasses.clw', _ABCLinkMode_), |
  DLL (_ABCDllMode)
```

The TYPE attribute identifies this class as a prototype only. Actual instances of the class will be created (instantiated) when needed. The MODULE attribute tells the compiler where to find the code for the class methods. The LINK attribute tells the compiler to compile and link these methods if, and only if, `_ABCLinkMode_` is True. The DLL attribute tells the compiler to look for these methods in another DLL if, and only if, `_ABCDllMode_` is True.

Have Hammer, Need Nail

Having built a new class, the next challenge is to use it. There are at least two likely situations:

- Access within the context of some other process
- Access in conjunction with an update form

The first type of use is easy. Just use the `Fetch` method as you would the standard ABC `Fetch`:

```
IF NOT Access:MyConfigFile.Fetch  
    ! do something, possibly including ...  
    Access:MyConfigFile.Update  
END
```

(This code assumes that nothing goes wrong during the update. You are of course free to be as conservative as the situation warrants.)

The second is even easier. In the `ThisWindow.Init` method of a form, before the code stores `GlobalRequest`, insert:

```
IF Access:MyConfigFile.Fetch  
    RETURN Level:Fatal  
ELSE  
    GlobalRequest = UpdateRecord  
END
```

This allows the form to be called directly from a menu and, no matter how it is called, causes it to update the one and only record in our configuration file.

To make this capability available for a particular file, set `Access:MyConfigFile.FetchOnlyType` to a value indicating how the single record should be accessed. This can be done anytime after the `FileManager` instances are initialized and before the `Fetch` is called.

Building A Wrapper

It would be nice if all of the necessary housekeeping could be wrapped up in a simple package so that implementing a configuration file would require nothing more than, say, adding a word to the user options of that file in the dictionary. Making things that simple is probably overkill, though, since most projects will have only one file of this type.

An application level extension template can do the same thing cleanly and without the extra processing needed to check every file in the dictionary for a user option. Given the name of a file, the template can initialize the corresponding file manager's `FetchOnlyType` property and add setup code to any form procedure which uses the file.

This is as good a time as any to set up a file for home grown templates. Call it `MyTemps.tpl`. A single line identifies the template chain:

```
#TEMPLATE (MyTemps, 'Homegrown Templates'), FAMILY ('ABC')
```

Another line establishes the extension template:

```
#EXTENSION(ConfigFile,'Implement Configuration
File'),APPLICATION,MULTI
```

The APPLICATION attribute tells the generator that this extension is applied at the application level rather than the procedure level. The MULTI attribute permits more than one instance of the extension in a single application.

A couple of lines of documentation describing what this template is supposed to do are in order:

```
#DISPLAY('This extension adds code to handle a specified file')
#DISPLAY('as a configuration file containing one and only one')
#DISPLAY('record. Forms for which this file is the primary file')
#DISPLAY('can be called directly without an intervening browse.')
#DISPLAY(' ')
```

A prompt allows specification of the file:

```
#PROMPT('Configuration File:',FILE),%MyConfigFile,REQ
```

There are at least two ways to specify the access method. The easiest is to add a second prompt (or more accurately, prompt structure):

```
#PROMPT('Choose Access Method:',OPTION),%MyAccessMethod
#PROMPT('Get by position',RADIO)
#PROMPT('Get next',RADIO)
```

Another alternative is to have the template select the access method based on driver. This has the advantage of hiding the access method from the programmer, thereby reducing the number of things that have to be remembered when using the template. A good place to put this selection logic is in an #ATSTART section:

```
#ATSTART
#FIX(%File,%MyConfigFile)
#CASE(UPPER(%FileDriver))
#OF('CLARION')
#OROF('DBASE3')
#OROF('DBASE4')
    #SET(%MyAccessMethod,'Get by position')
#OF('TOPSPEED')
    #SET(%MyAccessMethod,'Get next')
#ELSE
    #ERROR('File driver not recognized by ConfigFile extension')
#ENDCASE
#ENDAT
```

At this point everything necessary to generate code is in place. First, provide for initialization of Access:MyConfigFile.FetchOnlyType:

```
#AT(%ProgramSetup)
#CASE(%MyAccessMethod)
#OF('Get by position')
Access:%MyConfigFile.FetchOnlyType = FetchMethod:GetByPosition
```

```
#OF('Get next')
Access:%MyConfigFile.FetchOnlyType = FetchMethod:GetNext
#ENDCASE
#ENDAT
```

Adding code to the input form is a little trickier. The basic structure is easy:

```
#AT(%WindowManagerMethodCodeSection,'Init','()',BYTE'),PRIORITY(0)
IF Access:MyConfigFile.Fetch
    RETURN Level:Fatal
ELSE
    GlobalRequest = UpdateRecord
END
#ENDAT
```

Specifying PRIORITY(0) puts the code at the very beginning of the method, which is necessary because GlobalRequest is internalized very early in the initialization process.

This code should only be generated if the window is an update form for MyConfigFile. The template language provides a WHERE attribute to allow this kind of selection. With that attribute in place, the #AT statement becomes:

```
#AT(%WindowManagerMethodCodeSection,'Init','()',BYTE'),      %
    PRIORITY(0),          %
        WHERE(%ProcedureCategory = 'Form'           %
            AND %Primary = %MyConfigFile)
```

One of the benefits of writing about programming is that you learn things in the process. Until I had to split a template instruction to fit the printed page I was unaware that the template language has a line continuation symbol and that it differs slightly from the continuation symbol used in Clarion code. You'll find it documented on page 503 of the Programmer's Guide.

Unfortunately, when I tried to use this syntax in an example, the registry rejected the template. You'll find a note which suggests that this might happen on page 525 of the Programmer's Guide. So the continuation symbols (%|) in the above example are there for readability only and *should not be used in actual code*.

The point here is not that the documentation is bad. In fact, it's very good. But the template language has always had more little vagaries and nuances than the Clarion language itself, which makes documentation a daunting task. With both template and Clarion code, I read the documentation, code accordingly, test, and modify until it works. But with templates I don't worry too much if I can't explain in detail why what works, works.

Testing also reveals that %Primary, a built-in symbol which the documentation suggests should contain the label of the procedure's primary file, is blank when the WHERE clause is evaluated. Why? I don't know. But replacing %Primary with %File, a multi-valued built-in symbol, works. Why? I don't know.

Applying The Tool

The three files which define the `MyFileManager` class and the `ConfigFile` extension are included in the downloadable ZIP file. Using them is very simple:

- Place `MyClasses.inc` and `MycClasses.clw` in your Clarion LIBSRC subdirectory.
- Place `MyTemps.tpl` in your Clarion TEMPLATE subdirectory
- Register the `ConfigFile` extension.
- For any file that you want to handle as a configuration file, add an instance of the extension to your application's GLOBAL properties.

You will also need to tell the generator to use `MyFileManager` instead of `FileManager` for any configuration files. You can do this either on the **Individual File Overrides** tab or the **Classes** tab. The difference will be whether all files use `MyFileManager` (**Classes** tab) or just the ones you want handled as configuration files (**Individual File Overrides** tab).

I usually choose the more general case, expecting to add other functionality to the derived file manager.

In summary, once you have found a solution to a particular coding problem, it makes sense to take the extra step and build tools which implement that solution. Clarion's tool-building facilities are within the abilities of the average programmer, and skill in using those facilities improves rapidly with practice. I hope this example not only proves useful to you, but tempts you to build other tools on your own. You have nothing to lose and a world of productivity to gain.

A Correction

After this article first appeared, I received an email from Jan Jacob de Maa. He had downloaded the associated ZIP file and was trying to use it. Unfortunately, he was encountering compile errors.

Jan Jacob's experience led to the discovery of three errors in the article, which errors are repeated in the ZIP file:

- In the header for the `MyFileManager` class, an underscore is missing in the `DLL` attribute. It should read `DLL (_ABCDllMode_)` rather than `DLL (_ABCDllMode)`. The missing underscore causes the `DLL` attribute to

remain off when it should be on, wreaking havoc when compiling and running in 32-bit mode.

- In the wrapper template, the line `IF Access:MyConfigFile.Fetch` is missing the parentheses necessary to tell the compiler that this is a function rather than a variable. It should read `IF Access:MyConfigFile.Fetch()`.
- Also in the wrapper, the priority of zero which was given for code to be inserted in `ThisWindow.Init` causes the wrapper to execute its `Fetch` operation before the file is open. Changing the priority to 8000 places the code correctly. Also, because at this new position `GlobalRequest` has already been internalized, `GlobalRequest = ChangeRecord` has been changed to `SELF.Request = ChangeRecord`.

Source code

See “Appendix A: Getting Support,” p. 601, for information on how to get the source accompanying this book.

- `v1n7control2b.zip`

CHANGING DICTIONARIES

by Michael Pickus

Have you ever wanted to import a procedure from another application and found it was created with a different dictionary? It happened to me recently, only I needed integrate six applications into an EXE with DLLs and each application had a slightly different dictionary. When I tried to change the first application's dictionary, I saw the following warning:

"This is only guaranteed to work if the dictionaries are the same. It is not sufficient to have the same files and fields."

Sounds ominous. Why isn't it sufficient to have the same files and fields? Because every file, key, and field in a Clarion application uses a pointer (a unique identifier called an IDENT) to a corresponding file, key, or field in the dictionary. That's why when you change the name of a field in your dictionary, your application still works.

When you change the dictionary of an application the IDENTs won't match unless the dictionary is the same. The trick is to remove the IDENTs so that matching is done by file/field/key names. The following steps and program should enable you to change to a dictionary that has all of the same files, keys, and fields, but different IDENTs.

- 1) Ensure that all of the files, keys, fields, and relationships in the application are also in the target dictionary. If the differences are minor, you can open both dictionaries and copy (Ctrl-C) and paste (Ctrl-V) the files from a source dictionary to target dictionary. If they are substantially different, you may want to export the source dictionary to a .TXD file (**File|Export Text**) and import the .TXD file into the target dictionary (**File|Import Text**). If you have the Enterprise Edition, you also have the option to use the synchronizer. Save the dictionaries.
- 2) Open the application and export it to a text file. **File|Export Text** creates a *AppName.TXA* file.
- 3) Compile the *DeleteIDENTS* program, shown below (which you can also obtain from the source zip). If you create your own project to go along with Listing 1 make sure you include the ASCII file driver). Run *DeleteIDENTS* to convert the *AppName.TXA* file into a new file, *AppName_NoIdent.TXA*, with the IDENTs removed.
- 4) Create a new application. Select the target dictionary (Dictionary File) and delete the MAIN (first procedure).
- 5) Import the new .TXA file. (**File|Import Text**) the *AppName_NoIdent.TXA*. When the IDENTs are missing, the import uses the file, field, and key names to match the same names in the dictionary. You now have a new application with a new dictionary.

Here's the source for the *DeleteIDENTS* program.

```
PROGRAM

MAP
END

IDENT      UNSIGNED, AUTO      ! INSTRING returns an UNSIGNED
CloseP     UNSIGNED, AUTO

eIDENT     EQUATE('IDENT(')
eCloseP    EQUATE(')')
eNoIDENTs EQUATE('NoIDENTS.TXA')
eDict     EQUATE('DICTIONARY')
! eMax varies with the # of relationships
eMax      EQUATE(6144)

OldFileName CSTRING(129), AUTO      ! TXA in
NewFileName CSTRING(129), AUTO      ! TXA out

OldFile    FILE, DRIVER('ASCII'), NAME(OldFileName)
RECORD
aLine     STRING(eMax)

.
.

NewFile    FILE, DRIVER('ASCII'), NAME(NewFileName), CREATE
RECORD
```

```
aLine           STRING(eMax)
..
CODE
LOOP WHILE FILEDIALOG('Choose TXA to Convert', |
OldFileName, 'TXA|*.TXA')
  NewFileName = OldFileName [1 : LEN(OldFileName)-4] |
  & eNoIDENTS
  CREATE(NewFile)
  OPEN  (NewFile)
  OPEN  (OldFile)
  SET   (OldFile)
LOOP
NEXT(OldFile)
IF ERRORCODE(); BREAK.

! Don't use the old dictionary
IF OldFile.aLine [1 : 10] = eDict; CYCLE.

! Is IDENT on this line?
IDENT = INSTRING(eIDENT, OldFile.aLine, 1, 1)
IF IDENT

  ! Find closing parenthesis
  CloseP = INSTRING(eCloseP, OldFile.aLine, |
  1, IDENT+6)

  ! String slice the offending IDENT
  NewFile.aLine = OldFile.aLine [1 : IDENT-1] & |
  OldFile.aLine [CloseP+2 : eMax]

  ! If there is nothing to save then cycle
  IF LEN(CLIP(NewFile.aLine)) < 5; CYCLE.
ELSE

  ! If there is no IDENT, save the whole line
  NewFile.aLine = OldFile.aLine
END
APPEND(NewFile)
END
CLOSE (NewFile)
CLOSE (OldFile)
MESSAGE('Done')
END
```

Editing exported dictionary and application text files is a powerful option. For instance, you can globally change all date formats from @d1 to @d2b or even change from the Clarion template class to the ABC template class. But always remember to create new dictionaries and applications rather than overwriting your old ones.

Source code

See “Appendix A: Getting Support,” p. 601, for information on how to get the source accompanying this book.

Database Tips & Techniques

- v1n7idents-1.zip (original source)
- v1n7idents-2.zip (updated app)

ALIAS - WHO WAS THAT MASKED FILE?

by Steven Parker

"File alias" is one of those concepts that is at once extremely difficult and extremely simple. On the one hand, the documentation is quite sparse. Alias is not part of the language, not a Clarion statement (at least in the context of files - there is an ALIAS statement which changes keycodes, but that's not what I'm referring to). The one reference in the online help is to a FileManager method. Little information and, therefore, guidance, is available. On the other, the notion of referring to something by a different name (which after all is what an alias is) is really quite simple: two names, one object.

The behavior of file aliases is entirely a consequence of some Clarion fundamentals: *label*, RECORD and NAME. When you understand these concepts, it is almost easy to use aliases to solve real world problems.

Why Aliases?

Aliases were introduced to allow multiple relations between files, a feature not supported by the Dictionary Editor¹. Aliases, therefore, provide a second record buffer on a thread for

a file. This second buffer also allows looking up recursively within a single file. Aliases are powerful stuff.

Aliases, therefore, are the answer to the question “How can I be in two different places in one file at one time?” Other ways of saying this include: “How can I do a lookup from a file into itself?” “How can I have two different relations between the same two files?” and “How can I relate one record in a file to other records in the same file?”

Recursive lookups and relations are not easily visualized. Perhaps an example or two will clarify the matter.

A fairly clear example is an employee file in which you want to display the employee’s supervisor (a job title file displaying the supervisor’s title is much the same thing and can serve as the basis for this sort of lookup). Since a supervisor is also an employee (okay, I know that many oughtn’t be), there is an interesting and thoroughly unattractive choice: either you can have two files (two copies of the employee or job title file) or you can have one file. Maintaining only one copy of the file means that you have to look up within the current file.

A recursive lookup like this will not work without a *major finagle*² and *that assumes that you can describe it well enough to create a specification (it’s not that easy; try it)*. Two copies of the same file simply for the purpose of a lookup seems...well, stupid, frankly.

In an inventory module, a bill of materials or kit presents a very similar situation. When displaying a BOM or kit, there will typically be a browse of the items comprising the kit. This is a parent-child relationship. The parent inventory item is the kit but the child items are (or were) also inventory items. Even if the component items are stored in a separate file, they start out in the inventory file. When displaying the components, you need to relate one record in a file to other records in the same file (either directly or indirectly) or keep two copies of the inventory file. When creating the kit, you will need to select items from the inventory file which you are already accessing to insert the kit record. Again, the choices are less than palatable.

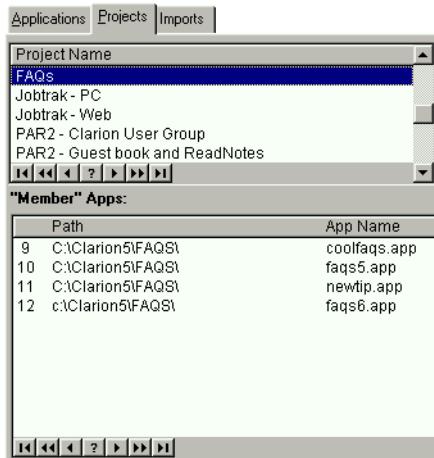
My own introduction to aliased files occurred while updating my *Go To Lunch* batch compiler, available at the CWICWEB (<http://www.cwicweb.com>) download site.

I needed two different browses of the same file on two tabs. On the first tab is a simple browse of the app list. On the second tab, the app list is displayed filtered (actually, range limited). Sounds like pretty standard stuff, right? On the second tab, however, the file is

-
1. The Relational Model also frowns on this practice. But, in the words of Randy Goodhew, the nice thing about the Relational Model is that applications following it rigorously are unusable. Reality often conflicts with theory and, in such cases, the nod tends to go with reality (at least among Topspeeders).
 2. If you are not familiar with this term from the pre-PC era, it refers to a programmer’s ability to successfully merge geometrically divergent shapes and/or expeditiously scale vertically enhanced edifices (putting square pegs into round holes and leaping tall buildings in a single bound, doof).

used as a child of another file (Projects) and the second tab shows a list box from the parent file. The app list is filtered on the currently selected parent record (see Figure 1).

Figure 1: The Go To Lunch Batch Compiler showing an aliased browse ("member" apps)



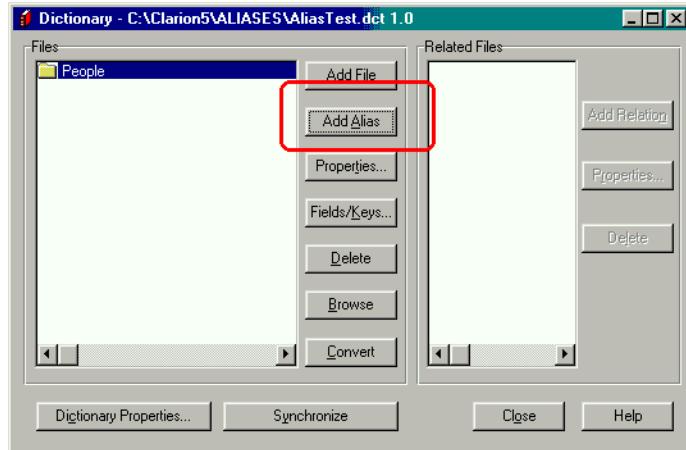
What is not standard is using a single file standalone and as a child in the same procedure. The standalone instance will set the buffer to the last accessed record. The second instance will always range limit the file on the relating key values. Mashed buffers are a certainty: the file will always end up range limited on the linking field value from the second tab. Even though the first browse appears normal, you will only be able to access the record that is active on the link as currently set on the second tab (opened last, the parent file "touched" the target file last and so gets its way).

Creating An Alias

Before continuing I need to describe how to create a file alias. It's a bit "cart before the horse," but this is the easiest way to present the information.

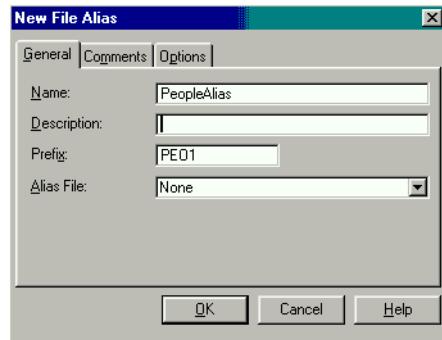
The Dictionary Editor makes creation of aliases simplicity itself: just press the **Add Alias** button (see Figure 2).

Figure 2: Creating an alias in the dictionary editor



This will call the **File Alias** worksheet, shown in Figure 3.

Figure 3: The File Alias worksheet

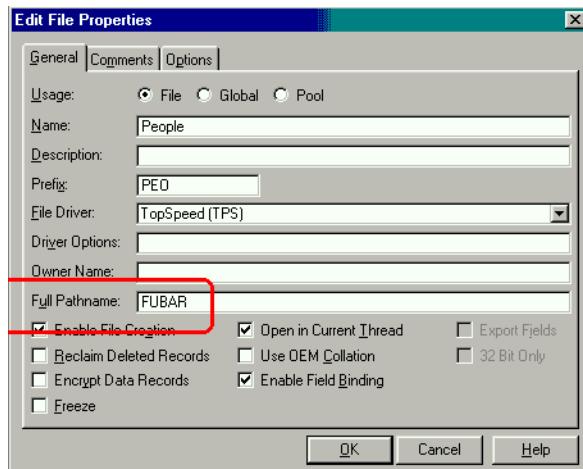


If you are not already using the NAME attribute on the file to be aliased, you will be notified that this is required.

Alias - Who Was That Masked File?

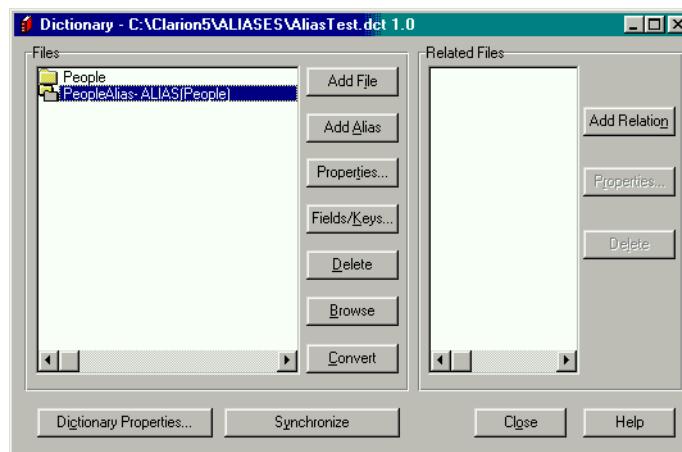
There is no particular reason to use a variable for the NAME, unless you *wish* to (see “NAME() Comes Of Age,” p. 591); you can just specify a DOS file name:

Figure 4: Completing the NAME() attribute for the file being aliased



When you complete all the prompts, your dictionary will look like Figure 5:

Figure 5: The completed dictionary worksheet



It is extremely important to note that if you do not complete the NAME attribute of the base file, your dictionary will still look like Figure 6. However, when you run the app, a file will be created on disk for the “alias” (if you have file create turned off, you will get an error

message). Of course, this defeats the purpose: file aliasing is supposed to work with a single disk file.

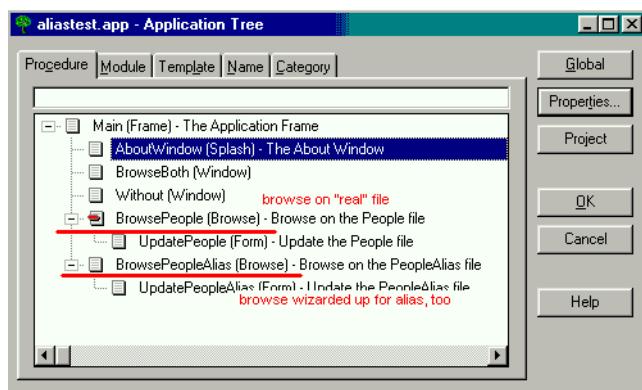
Note: While the environment will let you complete the worksheet without specifying a NAME, *don't*.

Finally, given the original purpose of aliases, you will find that the alias inherits the base file's fields and keys but *not* its relations. You may create any new relations needed, including relations to files already related to the base file. The idea is to use different keys to create new relations.

What Is An Alias?

With the dictionary complete, if you wizard up an application you will notice something quite remarkable. When you examine the application tree, you will see procedures generated for both the base file and its alias. It is as if the Application Generator is treating the alias as an entirely separate file. Think about this; of course it should!

Figure 6: Automatically generated procedures both “files.”



If you compile and run the application, you will be able to call both browses and their update forms without incident. Well, of course you should!

NOTE: Do consider checking the **Do Not Populate** box for any aliases in your dictionary.

If you examine the generated source for the attached sample application, the impression that there are two entirely different files will be further reinforced. Both files are declared separately and without apparent cross-reference in the main source module. There is a `FileManager` and a `RelationManager` for each. What you will not find is any

obvious connection between the base file and its alias, no reference to the fact that one is the “real” file and one (a sort) of pseudonym. Alias is not a Clarion statement, neither is it an attribute like NAME or CREATE.

If you search all the source files generated (or read the on-line help on the `AliasedFileManager` method), you will find (`bc0.clw`) a single reference to the alias:

```
SELF.AliasedFile &= Access:People
```

in the `Hide:Access:PEOPLEAlias.Init` method. By referring the alias’ `FileManager` to the base file’s `FileManager`, the ABC classes “know” which is the “real” `FileManager` (that is, it does not instantiate a `FileManager` for the alias but uses the base file’s `FileManager`). But you do have to dig to find this.

What is less difficult to miss (though it does not jump out and bite your...nose) is that both file declarations have identical NAME attributes. And this is the key to comprehending what an alias is, how it works and what it does.

Built On The Basics

If you examine the code in the main source module, there are three Clarion key words to consider. Combined, they make file aliases work.

First, you will find a label for each of the files. In the sample application, for example, you will find both `PEOPLE` and `PEOPLEALIAS`. Different labels allow you to refer to each of them separately (that’s what labels do).

Next, you will find matching and identical RECORD structures. According to the Language Reference Manual, a RECORD structure is what creates the memory buffer (per thread, if you select the THREAD attribute for the file). And, since there are two RECORD structures, with unique prefixes, there will be two buffers. This is what will allow two different records to be in memory simultaneously.

Finally, the *same* NAME attribute is used for both file declarations. Therefore, both RECORD structures/buffers refer to the same physical file on disk. And *that* is what makes an alias work (well, that and the use of the same `FileManager`³).

3. Of course, you can cut and paste a file declaration, changing the PREFIX, but use the same NAME. You will be able to access the single file from two browses and do so simultaneously. You will be able to update the file from multiple browses simultaneously. You will be able to use one as a lookup file for the other. okay, so you instantiate a second `FileManager`. But other than that, what’s the big deal? Clever, eh? Open two browses on a single file created in this way. Change an existing record in one of the browses. Now, highlight the “same” record in the second browse and press the Change button. It isn’t updated and, in fact, it won’t until you save your edits. So, the big deal is that cleverness can result in lost concurrency checking.

To reiterate, declaring a file alias creates both a second label and a second RECORD structure.

Buffers

”Buffer” is an extremely important concept in Clarion. A Record buffer is a memory structure which contains values from the currently accessed file record. Think of it as paralleling the file’s Record structure.

A Record buffer is created for each file in a procedure *when the file is opened*. Thus, record (file) buffers are thread specific.

Further, the buffer only contains “fields” for those fields actually referenced in the procedure. In a Form or Process procedure, this is all of the fields. In a browse, it is not. The buffer in a browse contains only those fields populated in the View. Thus, fields in the list box and fields added to the “hot” list contain current data. Any other file field is unreliable.

These are logical entities: like an old-fashioned line number, a label identifies a location in the code and a RECORD is a (dynamic) location memory.

If you run the sample application, you will see that is entirely possible to populate two browses of the same file on a single window *without* using aliases. It is possible to do this using different keys for each list (select **Browse|Browse Both -- Without Alias** from the main menu of the example application).

However, no matter what you do, only one record will be in memory. If you run the sample lookup procedure and select a record, you will overwrite the values in the initial record to those of the one selected in the look up. There is no way to look up a supervisor from employee data entry or to select kit items here. Try it, you won’t like it.

On the other hand, if you populate a browse from the primary file and a browse from an alias of it, you can have two different records in memory at the same time. Having two records in memory is, of course, not especially noteworthy. What is noteworthy is that both are from the *same* file, at the same time, on the same thread.

In the sample app, select **Browse|Browse Both** from the main menu. For each browse box there is a hot field indicating the record in memory for each list. Notice that they are only the same if you intentionally select the same record from each list. Run the matching lookup demo and you will see that it operates just as you expect, just as if there really were two copies of the file.

Using An Alias

An aliased file, of course, can be populated and used in exactly same way as any other file. After all, it has a `FileManager` (a reference to the `FileManager` for the base file) and `NAMES` a disk file. Thus, the “real” file will be affected by adds, changes and deletes. Even the Wizards know this.

But that is not what aliases are about.

The special need addressed by the second buffer that an alias provides is when you need to access the same data set twice on a single thread (since buffers are allocated on the thread; again, see the lookup examples in the demo application).

Say, for example, you have an inventory item that is a kit/bill of materials. On its update form you would like a tab with a browse box listing the component items. These component items may well be stored in a child file but their descriptions will most likely be in the inventory file. So, you need to be in the inventory file twice. The first access gets the data for the record being updated and the second gets the descriptions for the component items.

The browse box descriptions for the kit components would be populated from the aliased file (the alias being related to the child as a second parent). The primary record buffer will still contain the data from the inventory record being updated. Neat. No mashed buffers (which is exactly what happens if you use the inventory file directly for both purposes).

Similarly, if you have an employee record in access, a lookup procedure for the supervisor would be created from an alias for the employee file. Again, two different records from the same file in memory simultaneously with *no* cross talk on the thread.

Caveats

In reviewing an early draft of this chapter, David Bayliss pointed out that what I discuss here is the “proper” use of file aliases.

Prior to C5B, an alias is *always* required if you access a file twice on the same thread (not simply in the same procedure) for *any* reason. He provided the following illustration:

An Employee form has a lookup to Department. The browse on Department also looks up the supervisor name (an employee). There are two accesses to Employee here. They are in different procedures but on the same thread. Because they are on the same thread, they use the same buffer. Prior to C5B, either template set would have problems with this scenario. You could not reuse an existing Department browse.

Since then, intelligence has been added in C5B to try to detect this. Buffers are automatically saved and restored in many cases like this. Specifically, file usage in procedures is monitored (based upon the UseFile method). If a child procedure re-uses a file used by a parent *and if* the child doesn't explicitly say it wants to overwrite the parent record, Save/Restore file is placed around the child usage. David Bayliss has written a chapter on this subject ("Propitious Memory Corruption," p. 577).

Summary

If you need to access a file twice on a single thread and those accesses *may* involve different records, the conventional wisdom is to create and use an alias.

Recursive lookups are the obvious example of needing to access two different records in a single file simultaneously. The other "standard" case is when you need two different relations between the same two files. In these cases, an alias is appropriate and will always work and failing to use one will cause your app to fail.

If you absolutely insist on not using aliases, again see David Bayliss's "Propitious Memory Corruption," p. 577, which describes the enhancements to USEFILE.

Source code

See "Appendix A: Getting Support," p. 601, for information on how to get the source accompanying this book.

- v1n7aliases.zip

PROBITIOUS MEMORY CORRUPTION

by David Bayliss

I clearly remember my first introduction to the Clarion application paradigm. It was during the attempted closedown of the CFD 3 beta: the Florida R&D department was having trouble tracking down some problems with the templates and they sent them over to see if I had any ideas. They sent over a note: “Any ideas why this doesn’t quite work?” I responded: “I have loads of ideas why it doesn’t quite work, what I don’t get is how it works at all....”

To understand my response you need to understand my training. I came to computer science from pure mathematics, where my primary interest was computer languages. As such I had studied many of them, the distinctions between them, their strengths and weaknesses.

One of the main issues that we studied was minimization of scope. The logic is extremely simple. Programs work best when all the variables have the values they are supposed to have. Thus good program design maximizes the chance of each variable having the right value. You maximize the chance of a variable having the right value by minimizing the chance that someone gives it the wrong value. You do that by stopping them touching it, and you stop them touching it by making sure it isn’t declared where they can see it.

The simple rule to implement the above is *avoid global variables*. The more modern and OOPy way is to use the PRIVATE attribute as often as possible. In a language such as C++ you can go further and declare variables halfway through a procedure to stop people further up the procedure touching it. You can even define nested scopes so that a given variable is only visible over (say) three lines of a procedure.

Another key feature of safe languages is the ability to clearly define an interface, especially between procedures. In particular you minimize the number of procedures with side effects. Ideally you use functions that return results and if you want the result again you recall it. The ultimate is the “provable” language where you define what a procedure does purely by the incoming parameters.

Of course all of these heuristics are made to be broken, and where required one would slip in a global variable, or allow some procedure to have an external side effect. But each time I did so I was mindful that I was reducing the maintainability of the app, reducing my ability to scale the app and ultimately reducing my productivity.

Now the Clarion system has a proven ability to produce massive, maintainable apps, astonishingly quickly. Studying the legacy application paradigm was something I looked forward to.

Clarion Legacy Application Paradigm

What I saw took my breath away. The Clarion application paradigm is simple, audacious and astonishingly effective. Everyone just shoves values into global variables (usually file buffers), assumes everyone else has put just the right value into just the right variable at just the right time and carries on regardless. And it works, very very well.

In a browse a given variable could easily be used for six or seven different logical things. It could be the use variable of a hot field, a range-limit field in a filter, a locator field, the use variable for the locator entry control, a parent range-limit of a child browse, the selection field for a vertical thumb, the source for computed fields and the scratchpad I use to load the data before copying it into the queue.

Most germane to this chapter is the behavior across a procedure call. When a browse calls an update it just reloads the present record and calls the update form. When the update form returns it just assumes the record coming back is a good one, repositions itself and keeps going.

The “current value” range limit works similarly; it locks the browse to be range limited by whatever happens to be in the record buffer when the procedure is entered.

Selection from a browse again works simply. The browse just “leaves the record buffer” on the current record (every time you change the selected item the record buffer changes) and the update form uses any field it feels like.

So why didn’t the templates *quite* work? Well the first problem is the “six or seven logical things” I mentioned above. I suspect that as you read them you thought, “they’re all the same!” And they are usually, but not all of them all the time.

Consider the locator field when you have type in “fred” and the selected record is “gerry” (fred not existing), what should the locator field show? What should the attached entry read? What is the value of that field if it is also used as a hot field? As a range limit on a child browse?

Suppose I enter a filter that renders the browse empty. Is the field now blank? Even if it is a range limit field?

How about when I am loading in data to fill the browse; do I reset the field used as the range-limit for the child browse? Or do I wait to the end?

The Cost

Over many legacy template releases these questions were all answered to the point where buffer variables generally had what most people wanted in them most of the time, but there was a cost. Every time a logical part of the browse used a global variable it also had to post a message to the browse to refresh itself so that the expected logical value was in the corresponding global variable. The affect was that legacy browses always loaded the file data at least three times, and you could concoct examples (using child browses, locators etc) where it would re-load eight times for one notional “refresh”.

Another cost is that all the components felt they had to keep the buffers fully loaded. So a browse loaded a whole record for every new selection. A select browse did a full re-load (including child files) upon returning.

So Clarion had successfully married together productivity and functionality by applying a huge dollop of pragmatism.

The Kids

In many happy marriages there comes a point where the normal peace and harmony is shattered by the arrival of children, and nothing is ever quite the same again. This was true for the Clarion marriage too. CFD made a gigantic leap forward by the introduction of

referential integrity (RI) checking. You could now add and delete records from your data and your data would remain consistent, and it happened automatically. The dictionary editor/appgen were extended to understand the idea of relationships, especially the one-to-many relationship.

This technology was vital to the long term survival of the product: relational database programming had now arrived in a totally safe and automated way.

Usually.

The problem with adding RI to the templates (and other things such as “must be in file” validation) is that suddenly an innocent action upon one file didn’t simply change the fields in that file: it could change the fields in any related file too!

Try the following with the legacy templates. Give yourself a many-to-one relationship between two files with cascade on updates and clear on delete. On the update for the “many” file give yourself a field lookup button to a browse on the “one” file with a select button. Enter some “many” records each with “one” lookups (you will need to enter some values to look up). Now edit a “many” record (note the form contents), call up the lookup browse with the field lookup button and instead of selecting a record, delete one. Then cancel the select. You will find the “many” record has magically changed values! If you then press **OK** (not advised) your data is corrupted.

You can get similar effects with “must be in file” validation upon a form.

The simplest way to avoid most of these problems is to prevent insert/change/delete on select browses (the legacy wizards do this). The legacy RI *can* still scramble your data other ways but it is far rarer.

The Rope

With Clarion 1.5 we added another vital piece of technology that extended the relational nature of the Clarion language: the **VIEW**. The **VIEW** is brilliant. You simply declare the primary file, the fields you want from the secondary files and it handles all the rest for you. I genuinely believe the **VIEW** makes Clarion one of the simplest DBMS systems to use in existence.

Regrettably it makes it a little *too* easy to use. Whereas before people were quite content to have a few foreign ID fields floating around in their browses there was now no excuse. Simply populate the field from the child file and the **VIEW** handled the rest. The problem is that whereas a “book browse” would typically alter the “book” file it could now be loading values into subject, author, and publisher files without a moment’s thought on the part of the programmer.

Consider a book file related to subject (many-to-one). The subject form would thus have a child tab listing all the books on a particular subject. Now the book browse (on the child tab) would typically have a `SubjectId` field. This you remove and populate the subject name instead. Seems reasonable and looks reasonable and under legacy it will work...often. There are a few interesting quirks. Firstly, if you go to the child browse and delete all the children when you come back to the **General** tab (with the form information) all the fields will be blank. If you press **OK** you lose your parent data. The deadlier one is that if you edit one of the child records and on the child record alter the parent (i.e. you are moving it from one parent to another) when you come back to the form the data you are looking at will be correct but the *record* under the form will have switched to the new parent! Thus any edits you make to the parent record you can see will actually happen to the new parent of your ex-child!

The Call Tree

A subtler side effect of the rope given in 1.5 is that grandchildren started to become treacherous. Imagine I am editing a patient file. I have a lookup to the doctor browse, and from the doctor browse I call up the doctor update form to see if a given doctor can do such and such. The doctor update has a child tab of patients which *upon loading* writes changes all over the patient file. I cancel out of the doctor update, select the suitable doctor and keep going. When I press the **OK** button I either corrupt my data (legacy) or get an assertion (ABC).

Most people are aware that if they see the ***Recursive call message on the procedure call tree then they have a potential problem. Fewer people are aware that if *ever* any file appears twice in a procedure call tree (other than by design) then mayhem may ensue. The recursive warning is just a guarantee things are going to go wrong.

Then ABC Came Along And It All Stopped Working!

At least that is what I have been told by many people, at great volume. ABC *did* make some fundamental changes but I believe they are justifiable. I'll go through the four cases in sequence:

- 1) Inefficiency due to multiple usages of a variable within a procedure, especially the browse. This has been tackled in ABC by teasing out some of these logical uses, so locators have their own shadow, range-limits are stored by value, etc. By doing this ABC only ever loads data once, it avoids painful

REGETs, and generally is more efficient under client server and especially SQL. The down side is that hot fields have to be defined as hot fields and reset fields as reset fields (previously you could forget to fill them in and generally got lucky). Further, from a select browse you can only actually guarantee the linking field will be filled in (although C5 has a `SelectWholeRecord` property to force a full record reload upon selection).

- 2) Unexpected record corruption due to *automatic* file validation and RI. ABC has eliminated these problems using save/restore file technology. There *usually* isn't a down-side unless someone was using one of these corruptions to good effect.
- 3) Use of a file twice within a procedure. This is the one people don't like. In legacy this will usually appear to work, even when it doesn't work the form *looks* okay, it is just the data on disk that gets corrupted. ABC is rather different. In this situation it will Assert, throw up garbage screen displays (if the assertions are ignored), and generally make it clear something is horribly wrong. I have had many people ask "Why can't you get it to work like it did in legacy?" They never believe that legacy doesn't work until I take them through the steps. So what is better? A system that works 99% of the time and just subtly corrupts your customers data or a system that downs tools and refuses to budge until it is fixed? See some of my "offensive programming articles (<http://www.users.globalnet.co.uk/~dabay/#Documents>)" for the line I take.
- 4) Up until C5B ABC treated case 4 much like case 3. If the user dug far enough down the procedure tree to cause an unintended corruption then upon returning to the grandparent procedure it Asserts. Again I claim this is better than the "corrupt and continue" approach. However, it isn't quite as nice as case 3 because there is nothing to guarantee that the procedure tree will be fully "drilled down" during testing.

The only real solution to 3 and 4 is what I call "structural" aliases. Steve Parker's chapter, "Alias - Who Was That Masked File?," p. 567, deals with the use of aliases when there is a logical reason for them. Cases 3 and 4 require that sometimes you need to use aliases even when they are representing the same logical entity.

Automatic Aliases

C5B makes a slightly radical but very strategic step towards solving case 4. Essentially the idea is this each procedure declares which files it is using, then if it calls a procedure that uses the same file the child will save the contents on entry and restore them upon exit so

that the parent procedure has the record buffer it was expecting. This is managed by a global “FilesManager,” so if a great-great grandchild starts using a file it is again saved/restored as required. You can even go in and out of recursive procedures and everything still works. Note that the child does *not* get a cleared record buffer, so data transmission from parent to child still happens as before.

Of course this is not quite enough. Sometimes a child *wants* to change the data so that a parent can use the result. To enable this to be specified (and for the procedure to register the files it uses) the `UseFile` method was adapted to take a `UseType`. These are as follows:

- `Corrupts` – This file will be altered by this procedure but the procedure is not bothered about the contents of the buffer across procedure calls.
- `Uses` – This file is altered by this procedure but expects its children not to corrupt the buffer across procedure calls.
- `Returns` – This file will be altered by the procedure and the alterations should be transmitted back to the parent (note this overrides the `Uses` declaration in the parent).

The ABC templates have also been adapted so that the various control templates know the action they are supposed to have upon a file and make the `UseFile` calls accordingly. ABC template applications thus work unchanged from C5A to C5B. However hand-code (or non-TopSpeed templates) may need altering to register that a given child *wishes* to change a file buffer used by a parent.

In Conclusion

There is no such thing as a free lunch. The Clarion application paradigm has essentially bucked the trend for many years to deliver productivity and functionality. ABC has eliminated two of the compromises made by the legacy templates to achieve this. The fourth has now largely been tackled by C5B at the cost of some code compatibility. ABC is presently unable to tackle the third issue although it *does* flag the error enabling the user to take action.

DETECTING DUPLICATE RECORDS

by Gordon Smith

Recently on the newsgroups someone mentioned that TopScan didn't identify duplicate record errors while building the keys. After digging around in the source code I confirmed this to be indeed true and set about writing a "duplicate record" detector (DRD) which was file neutral. This chapter will take a look at the resulting class.

The Requirement

The DRD would have to perform the following functions:

- Locate all duplicate errors, for each key.
- For each duplicate error, locate all clashing records (remember each duplicate error can have more than two associated records)

The Solution Overview

The solution ended up being quite a bit simpler than I had originally envisaged, although on the down side it involves a brute force approach where every record in the file will need to be tested. The logical flow goes something like this:

- 1) Rebuild all keys (this is essential, as keys must be up to date to enable location of clashing records).
- 2) Loop through each record in file sequence.
- 3) For each record use the DUPLICATE (FILE) function to see if it is a duplicate.
- 4) For each duplicate record, loop through all the keys to find which ones are reporting clashes (remembering that it can be more than one key causing the error).
- 5) For each duplicate key, find the one valid record and associate the current (error) record with it.

It is important to note that when duplicate records exist in a given key there will always be one valid record, while the rest of the duplicates will simply not exist for that key.

There are several ways a duplicate record can be created, the following are probably the most popular:

- 1) APPEND: Since APPEND doesn't update any keys it is probably the easiest way to create duplicate records.
- 2) File conversion (common type A): If the conversion utility simply creates a new empty file and appends all the records from the "before" file, then this will be the same as 1.
- 3) File conversion (Common type B): It is possible in some file drivers to modify a key structure (remove a DUP attribute for example) and then simply delete the key (somefile.k01 for example) from the hard drive, allowing Clarion to rebuild it afterwards.

The Solution

The solution has been presented in the form of a class. (The attached source code also has a small example PRJ and CLW file). The definition looks something like this:

```
FindDupClass    class, type
init            procedure(file f)
kill            procedure
```

```
buildKeys      procedure
findDup       procedure
display        procedure
end
```

The class would typically be used as follows (and since it is a class, the actual code is appropriately simple):

```
TestFileForDup procedure (FILE TestFile)
cFind FindDupClass
code
cFind.Init(TestFile)
cFind.BuildKeys()
cFind.FindDup()
cFind.Display()
cFind.Kill()
```

The code “notables” now follow:

BuildKeys method

Rather than calling `BUILD(FILE)`, this method builds each key individually by using the `PROP:Keys` and `PROP:Key` file properties. The main reason for this is to avoid a nasty side effect of `BUILD(FILE)` where it aborts `BUILDing` all keys when an error is encountered (in this case a duplicate record error!). A nice addition to this method would be to actually check if any errors occurred during this operation, if not then there will be no need to check for duplicates. Another useful addition would be to use the `PROP:Completed` and `PROP:ProgressEvents` properties to display a nice progress window, with the option to gracefully cancel.

```
FindDupClass.BuildKeys procedure
i unsigned, auto
k &key
code
loop i = 1 to self.f{PROP:Keys}
  k &= self.f{PROP:Key, i}
  build(k)
end
```

The FindDup method

The `FindDup` method is the brute force part of the solution (it loops through the entire file, record by record). Since files can be large it uses the `EVENT:Timer` event on a

Database Tips & Techniques

simple status window. When a duplicate is found, a private method is called (`CalcDupInfo`) to analyse the duplicate.

```
FindDupClass.findDup procedure

Prog long(0)
Found long(0)

Window WINDOW('Caption'),AT(,,200,44)
    ,FONT('MS Sans Serif',8,,FONT:regular)
    ,TIMER(1),SYSTEM,GRAY,DOUBLE
        PROGRESS,USE(Prog),AT(5,10,190,10),RANGE(0,100)
        PROMPT('Duplicates:'),AT(5,27)
        BUTTON('&Cancel'),AT(150,25,45,14),USE(?ButtCancel)
        STRING(@s64),AT(50,27),USE(Found)
    END

i unsigned, auto
FinFlag byte(FALSE)

code
open(self.f)
assert(~errorcode())
stream(self.f)
assert(~errorcode())
set(self.f)

open(window)
window{PROP:Text} = self.f{PROP:Name}
?Prog{PROP:RangeHigh} = records(self.f)
accept
    case event()
    of EVENT:Timer
        if ~FinFlag
            loop 100 times
                next(self.f)
                if errorcode()
                    FinFlag = TRUE
                    post(EVENT:CloseWindow)
                    break
                end
                Prog += 1; display(?Prog)
                if duplicate(self.f)
                    Found += 1; display(?Found)
                    self.CalcDupInfo()
                end
            end
        end
    end
close(window)

close(self.f)
return FinFlag
```

The calcDupInfo method

This method goes through the following steps to find duplicates.

- 1) Save the current pointer and position for the current record.
- 2) Check each key to find the ones reporting the error.
- 3) For each duplicate record there will be one valid entry. This is located using GET (SELF.F, K) procedure. Since the current record buffer for the “error” record “matches” the one valid record, when the GET is called it will return the one valid record!
- 4) For each duplicate AddResult is called twice (**Note:** AddResult will only add records to the result queue if they haven’t already been added):
- 5) To add the valid record to the result.
- 6) To append the duplicate record to its associated valid record.

The original record is then restored with the RESET, NEXT combination (you must use the RESET (FILE) form of RESET, for obvious reasons).

```
FindDupClass.calcDupInfo procedure

j unsigned, auto
k &key
tmpPos like(ResultQueue.Pos)
tmpPoint long, auto

code
tmpPos = position(self.f)
tmpPoint = pointer(self.f)
loop j = 1 to self.f{PROP:Keys}
  k &= self.f{PROP:Key, j}
  if duplicate(k)
    get(self.f, k)
    assert(~errorcode())
    self.AddResult(pointer(self.f), k)
    self.AddResult(tmpPoint, k, TRUE)
    reset(self.f, tmpPos)
    next(self.f)
    assert(~errorcode())
  end
end
```

AddResult Method

This method adds the duplicate record information to the result queue for displaying later (in the display method). It checks that this particular duplicate record hasn’t been added

already (based on the key name and its POSITION information), and if it has it will append the current record number to it.

```
FindDupClass.addResult procedure(string pnt, key k, |
                                byte AppendPos = FALSE)

  code
  clear(self.qResult)
  self.qResult.Rec = pnt
  self.qResult.Key = k{PROP:Label}
  self.qResult.Pos = position(k)
  get(self.qResult, +self.qResult.Key, +self.qResult.Pos)
  if errorcode()
    add(self.qResult, self.qResult.Rec)
  elseif AppendPos
    self.qResult.Rec = clip(self.qResult.Rec) & ', ' & pnt
    put(self.qResult)
  end
```

Summary

All in all it was quite pleasing that the solution ended up being so straightforward. Something similar will undoubtedly find its way into TopScan. One more point: to verify the sample program, try using TopScan to view the created file.

Source code

See “Appendix A: Getting Support,” p. 601, for information on how to get the source accompanying this book.

- v1n9dupkeys.zip

NAME() COMES OF AGE

by Steven Parker

Various kinds of files, such as calendars, schedules, ASCII files, accounts, or files on other drives or other machines share the characteristic that multiple physical disk files can have the same structure. It is often desirable to have a single dictionary specification to declare that structure so that a single procedure set (application) can operate on them.

Clarion builds in this capability with the `Name` attribute but provides no management capabilities. It is up to the developer to manage which file is in process at any given time.

Of Syntax and Semantics

In Clarion, there are two concepts you must not confuse, *label* and `Name`.

Variables and data structures (which include windows, queues, views and files) are referred to by their label. Clarion code *always* uses the label.

`Name` is an attribute. Variables and structures may or may not have a `Name`; in fact, they usually do not. The *contents* of the `Name` attribute, if any, are for use of an external object.

This could be another program, like a database engine or the operating system, or a Clarion DLL, like a file driver.

In the case of data files, the Name attribute contains the fully qualified directory entry for the file. For example, a file declared as:

```
Customer File,Driver('Topspeed'),Name('foobar')
```

will look for or create a disk file called `foobar.tps` in the current directory. Your code will never contain this, only "Customer."

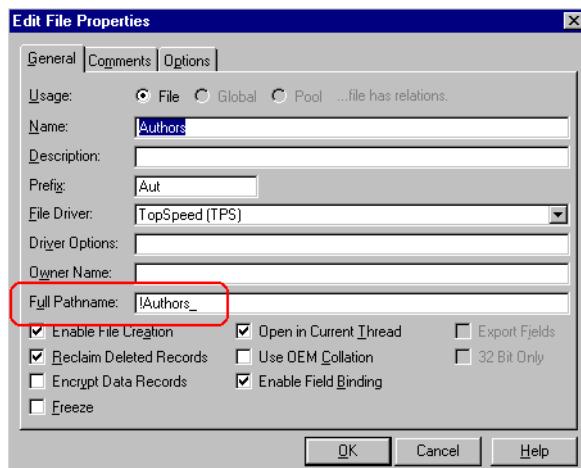
Multiple Files, One Declaration

To handle multiple physical files from a single logical declaration use a variable in the Name attribute:

```
Customer File,Driver('Topspeed'),Name(GLO:FileName)
```

You create this in the Dictionary Editor by completing the Full Pathname field on the **Edit File Properties** worksheet (Figure 1) and pre-pending an exclamation point.

Figure 1: Global Data Properties



This is where the fun starts.

If you declare a file like that in Figure 1, your app will not compile. The compiler error makes the reason clear: the variable was not declared. Not only must you declare the variable, you must do so *before* the file declaration. That means that you have to use the

Before File Declarations source embed because the **Global Data** button places variable declarations *after* the file declarations.

Thus, global files (files in the dictionary) require a global variable. Files declared at the module or procedure level, in hand code, require only that the variable have the STATIC attribute (though it can be at a higher scoping level).

Next, having added the STATIC attribute, if you try to run the app, it will crash and burn. The variable is empty and the app does not know the DOS name of the file it is supposed to open:

File() could not be opened. Error: Invalid Filename(45)

You may initialize the variable any time you wish but obviously not later than the first attempt to open the file.

SetName

The ABC FileManager class also provides the SetName method to set the value of the Name attribute. This method allows you to set the variable directly, without an assignment. Moreover, you can call this method in a procedure which does not actually use the file(s) in question (it's a property of the FileManager, hence available any time after the FileManager has been instantiated). The app frame comes to mind, as does the global Procedure Setup embed.

Citing advantages of SetName, Pierre Tremblay observes (in the OOP newsgroup):

The variable in the name attribute for the file structure doesn't need to be exported [from the data app]. The FileManager class is holding a reference for that var.

So, to initialize this variable without having it exported from the DLL, the SetName method is the only way to go.

Note what Pierre says: the variable "*doesn't need to be exported*." Yes, this means that you do not need to declare the variable in the non-data app(s).

Rules for Name-ing

- 1) Always use the file Label as declared on the **Edit File Properties** worksheet in Clarion language statements;

- 2) When using a variable in the Name attribute, initialize it before any attempt to open the file;
- 3) A variable used as a file Name may contain any O/S-valid string;
- 4) To use a variable, prepend the Full Pathname entry on the File Properties worksheet with an exclamation point;
- 5) The variable name must be unique (of course);
- 6) Declare the variable in your data application, before the file declaration.

Super Files

Topspeed files support a special syntax to allow multiple tables to be stored in a single file.

By using the special escape sequence ‘\!’ in the NAME() attribute of a TopSpeed file declaration, you can specify that a single .TPS file will store more than one table. (C4 LRM)

When using the TopSpeed driver, if you wish to store multiple tables in a single physical file, separate the file and table names with \!, as in 'TUTORIAL\!ORDERS'. This refers to the ORDERS table in the TUTORIAL.TPS file. (C5 On-line Help)

The format of the Name entry is *file_name\!table_name* which appears to preclude using a variable in the Name attribute of a TopSpeed file.

Appearances can be deceiving, for this is not the case. Designate your variable on the File Properties worksheet and initialize it in exactly the same way as any other Name, but use the special syntax:

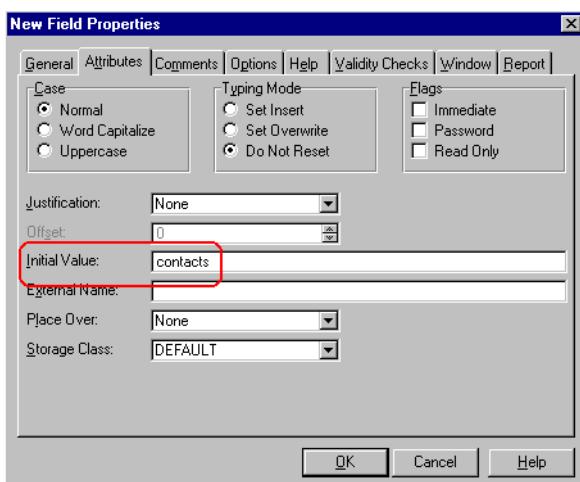
```
Authors_ = 'cwjfil\!Authors'
```

to initialize the variable. In this case, Authors_ refers to the Authors table in CWJFIL.TPS.

Multiple Locations

The executable resides in one place but you need the ability to address files anywhere. The file might be in the current directory or in another directory. The files might be on a network drive. You can even map an IP address to a drive letter and use the Internet.

Figure 2: Setting an initial value for a file name variable



Here, there is one file (set) but multiple possible paths. In these cases, the path tends to be variable while the actual name of the file tends to be a constant. If so, it makes sense to give the Name variable an initial value when it is declared (Figure 2). If you can get the path information separately, from an INI or configuration file or from `FileDialog`, just concatenate the two.

```
FilePath = GetIni('JtMatch','FilePath','','.\JOBTRAK.INI')
Contacts_ = Clip(FilePath) & Clip(Contacts_)
```

Multiple Files

You want to work on one company's accounts then another's. Or you need to update Alice's calendar then Bill's.

You do not have to leave the browse, if you do not wish to (though, obviously, you can and this would be easier to program). All you need to remember is that you *must* close the current file before selecting the new file. Select the new file, prime the Name variable and refresh the browse.

If you have multiple tabs and it is your intention that each Ttb show a different file in the same browse, you must close, prime and reopen in **ThisWindow.TakeNewSelection (ABC)** or the **?CurrentTab, New Selection (Clarion)** embed.

Multiple DLLs

Now multiple DLLs are *really* fun if you use variables.

Each variable must be declared in each of the project's apps.

If you're new to multiple DLLs, you should check the documentation. The important thing is that each app in the project needs to reference the files. But only one should allocate memory for them. This means that in all apps but one the files are declared and the EXTERNAL attribute added. The one app without the EXTERNAL attribute is usually referred to as the "global data DLL" and usually contains only the file declarations.

In the global data DLL, you simply declare the variable and, if desired, give it an initial value.

In every other app, you must also declare each of the variables. Each variable must also have the EXTERNAL and DLL attributes, similar to:

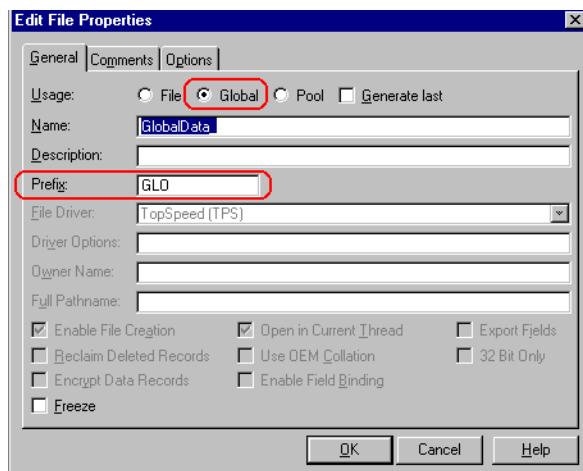
```
Contacts_ String(64), External, DLL(dll_mode)
```

to avoid compiler warnings (lots). (You do *not* need to do this if you use SetName).

Until C5, the method of choice was to create a text file with the necessary declarations and Include it in each application. The required file is usually created by requisitioning and modifying the data declarations from the main source file of the data DLL.

A major enhancement in the C5 Dictionary Editor makes this a thing of the past, totally automating the process.

Figure 3: Dictionary Editor Global Data settings.



Using the new **Global** option in the Dictionary Editor (see Figure 3) creates a file-like structure accessible in your applications as global data. That is, after you declare, say, `GlobalData`, you insert each variable in the same way you would add file fields. So, from the point of view of the Dictionary Editor this is a file but when accessed within an application, it is handled like global data, as intended.

The only restriction is that if you add several variables to this “file,” grouping your name variables, they will share a common prefix.

However, all of your declarations will be included and included correctly in each and every application. That is, your global application settings for dictionary handling (viz., the external flag) will be picked up and applied correctly without further intervention. (You still have to initialize variables, so *you* still have to do something.)

Note: As of C5EE SR1, Arnor Baldvinsson has discovered, this works correctly only with the ABC templates. The requisite template code was not retro-fit in the Clarion template chain. These lines are found in `ABProgram.tpw`, 218-37 and should be compared to `Program.tpw`, 90-99. He states that the new lines can be successfully substituted for the old. A copy of the modified template is available at <http://www.cwicweb.com/apps/cwlaunch.dll/download.exe.0>

Summary

The Clarion language has supported variable file names since its very beginning. In CDD, support was added to the Dictionary Editor and, partially, to the AppGen. But, until Clarion5, we had to hand code many, if not most, of the required declarations.

”When it rains, it pours.” Now you have two methods that are about as automated as you can get. How do you choose? This is the most difficult possible choice: two solid, reliable and easy to use techniques.

Source code

See “Appendix A: Getting Support,” p. 601, for information on how to get the source accompanying this book.

- v1n2name_app.zip

Appendices

APPENDIX A: GETTING SUPPORT

Getting the source code

Source code is available for download at:

<http://archive.clarionmag.com/books/dbsql/index.html>

If you do not have access to the Internet, please contact the publisher at the following address:

CoveComm Inc.
1036 McMillan Ave
Winnipeg, MB
R3M 0V8

Tel: 204-943-5165

Errata

Corrections to the book are listed at:

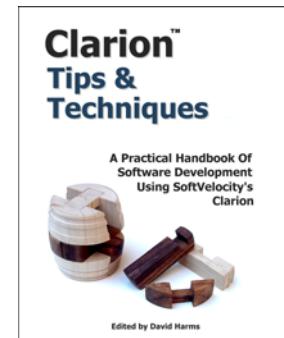
<http://archive.clarionmag.com/books/dbsql/index.html>

If you find an error in the text, please report it via the above web page.

APPENDIX B: RELATED ARTICLES

A number of database-related articles have been published in the book **Clarion Tips & Techniques**. For more information on obtaining this book please visit:

<http://www.clarionmag.com/books>



Database Tips & Techniques

A Class Wrapper For Files.....	145
Fast ASCII Files.....	157
How To Convert Your Database To SQL	159
Locating Records With PROP:SQLFilter.....	167
Recovering Deleted Records	169
A FileManager For Marked Deleted Records.....	175
Implementing SELECT DISTINCT in a TPS Database.....	181
A Class For The ASCIIng.....	189
Parsing Strings In ASCII Files.....	195
CLASSy ASCII File Importing	203
ASCIIng For More	211

AUTHOR INDEX

B

- Bayliss, David
 Inside ABC, FieldPairsClass and BufferedPairsClass 447
 Inside ABC, The FileManager 459
 Inside ABC, The RelationManager 483
 Inside ABC, The ViewManager 495
 Propitious Memory Corruption 577

D

- Du Beau, Vince
 Using The TPS ODBC Driver 97

F

- Ferrett, Scott
 How To Convert Your Database To SQL 175
Florek, Bill
 Using Dynamic Indexes With TPS Files 91

G

- Giles, Tom
 Using Client-Side Triggers In Clarion 6 543
Griffiths, John
 Date Filtering with MSSQL 429
 SQL Identity, Another Approach 399
Grospperrin, Bernard
 Creating Utilities For MS SQL 2000 405

H

- Harms, David
 An Introduction To SQL 141
 Designing Databases 1
 Getting Started With PostgreSQL 273
 Handling Many-To-Many Relationships 9
 Large Table Performance in MySQL 265

MySQL InnoDB Tables And Transactions 247
MySQL/MyODBC Notes 241
Reading Tables With ADO 103
Securing Remote Database Connections With SSH Tunneling 535
SQL Data Types Comparison 201
Using Clarion With MySQL 221

Hebenstreit, Tom
 Getting Into SQL On The Cheap 167

Heck, John
 Using Example Files With TPSFix 117

J

Johnson, Nik
 Working With Control Files 547

M

Morgan, Jim
 Managing Table Opens In ABC 517

Mull, Stephen
 Converting TPS To MS-SQL 183

N

Nicastro, Mauricio
 Avoid My SQL Mistakes! 197

O

Ogundahunsi, Ayo
 Converting Data With Linked Servers 357
 Converting The Inventory Example - Calling Stored Procedures 371
 Migrating The Inventory Application To SQL Server 317
 Using SQL Server's Data Transformation Services 341

P

Parker, Steven
 Alias - Who Was That Masked File? 567
 Displaying Normalized Data 55
 Name Comes Of Age 591
 Working With Control Files 547

Pickus, Michael
 Changing Dictionaries 563

R

Ruby, Thomas
 Managing Complexity, Rule 1, Eliminate Repeating Fields 21

Managing Complexity, Rule 2, Eliminate Redundant Data 29
Managing Complexity, Rule 3, Eliminate Columns That Don't Belong 39
Managing Complexity, Rule 4, Isolate Independent Multiple Relationships 45
Managing Complexity, Rule 5, Isolate Semantically Related Multiple Relationships 49
True Confessions, A Tale of Two Users 85

S

Smith, Gordon
 Detecting Duplicate Records 585
Staff, Brian
 Accessing TPS Files Via ASP 109
 Reading Tables With ADO 103
Stapleton, Andy
 The SQL Answer Cowboy 209

V

Vail, Eric
 Troubleshooting TPS File Corruption 125

W

Waterhouse, Jon
 AutoNumbering In Oracle 299
 Creating ODBC Data Sources At Runtime 527
 Referential Integrity In Oracle 295
 Transactions In Oracle 305

INDEX

Symbols

#PROJECT 531
ABCDllMode 557
ABCLinkMode 557

Numerics

64k limit
and OOP 448

A

ABC
BrowseClass
 ResetFromAsk 78, 80
 SelectWholeRecord 582
 SetQueueRecord 76, 78
 TakeRecord 96
 UpdateBuffer 82
 ValidateRecord 96
BrowseManager
 Ask 309
 TakeEvent 309
BufferedPairsClass 455–457, 501
 AddPair 456
 AssignBufferToLeft 493
 AssignBufferToRight 493
 EqualLeftBuffer 489
 Init 456
BufferedPairsQueue 456
ErrorClass 461, 469
SetProcedureName
 modified 523
FieldPairsClass 447–455, 489, 490
 455
 AssignLeftToRight 493
 AssignRightToLeft 453
 ClearRight 454, 492
 Equal 455
 EqualLeftRight 454
 Kill 455
FieldPairsQueue 456
FileClass 460
FileManager 484, 495
 AddFileMapping 463
 AddKey 463
 and dynamic indexes 93
 BindFields 477
 Buffer 464
 CancelAutoInc 473
 ClearKey 465
 Close 477, 510
 modified 520
 Construct
 modified 520
 Delete 474
 DeleteRecord 312
 EqualBuffer 470
 Errors 468
 Fetch 60, 96, 478
 FileKeyQueue 463
 GetEOF 466
 GetError 468
 GetField 466
 GetFieldName 467
 GetName 467
 HasAutoInc 464
 Info 465
 Init 462, 464
 Insert 478
 InsertServer 312
 KeyToOrder 467

Kill 462, 464
LazyOpen 462
Next 478
NextServer 478
Open 479, 510
OpenServer
 Modified 521
Position 480
PrimeAutoInc 474
PrimeAutoIncServer 300, 474
PrimeFields 476
PrimeRecord 476, 501
RestoreBuffer 470
RestoreFile 471
SaveBuffer 471
SaveFile 472
SetError 469
SetKey 468
SetName 468, 593
SetThread 462, 464
theory of operation 459–??
Throw 469
ThrowMessage 469
transactions 312
TryFetch 480
TryGet 480
TryNext 478
TryOpen 461, 479
TryPrimeAutoInc 474
TryUpdate 310, 480
Update 480
UpdateServer 310, 312, 480
UseFile 480, 488, 499, 583
ValidateField 476
ValidateFields 477
 ValidateRecord 477
FileManager theory of operation ??–481
FileManger
 debugging 519
 modified 519
RelationManager 16, 495
AddRelation 485
AddRelationLink 485
CancelAutoInc 486
CascadeUpdates 491, 492
Close 487
debugging 519
Delete 312, 474, 488
DeleteSecondary 489, 491
Init 486
Kill 486
ListLinkingFields 490, 497
LogoutDelete 492
LogoutDeleteClear 491
LogoutPrime 490
LogoutUpdate 492
modified 519
Open 488
OpenCloseServer
 modified 522
referential integrity 483
Save 490
SetAlias 486
SetQuickScan 490
theory of operation 483–494
Update 310, 489
UpdateSecondary 493
UseLogout 488
ViewManager
 AddRange 497
 AddSortOrder 497, 498
 AppendOrder 497, 498
 ApplyFilter 499, 501
 ApplyOrder 500
 ApplyRange 500
 Close 502
 filters 496
 GetFreeElementName 503
 GetFreeElementPosition 503
 Init 498
 Kill 499
 LimitMajorComponents 497

Next 502
 Previous 502
 PrimeRecord 501
 range limits 496
 Reset 502, 503
 SetFilter 500, 504
 SetFreeElement 497
 SetOrder 497, 503
 SetSort 502, 503
 sort orders 496
 theory of operation 495–504
 UseView 499
 ValidateRecord 503
 WindowManager
 Init 79, 96, 558, 562
 Reset 78, 82
 TakeCompleted 309
 Update 309
 Access 187, 533
 ADO
 cursor types 105
 RecordSet 104
 Open 104
 ALIAS 567–576
 see also file, alias
 ALTER TABLE
 see SQL, ALTER TABLE
 ANY 452
 and CLEAR 454
 memory use 453
 performance 456
 variant 452
 APPEND 586
 application
 design 577
 legacy 578
 splitting into DLLs 371
 array
 and many-to-many relationship 12
 and one-to-many relationship 23
 and relationships, one-to-many 4
 reasons for in files 27
 vs multiple fields 24
 Artigas, Roberto
 SQL templates 255
 AS-400 209
 ASP
 standard objects 112
 authentication
 public key 539
 AUTO_INCREMENT 202, 206
 autoincrement key
 see key, autoincrement
 autonumbering
 see key, autoincrement 12
 server-side vs client-side 303
 trigger 302
 AyoMSSQL template 379

B

BETWEEN
 see SQL, BETWEEN
 BILF 484
 BIND 194
 BINDCOLORDER driver string 255, 259
 Bomford, Geoff
 templates 328
 browse
 see also ABC, BrowseClass 76
 child 581
 hidden 17
 range limit 579
 update
 calling 578
 BrowseClass
 see ABC, BrowseClass
 BUFFER 192

BUILD 92, 94, 587
business logic
 and stored procedures 392
business rules 33, 317

C

CDD
 browse engine 451
CDSWrapper 531
CheckOpen 510
child record
 entering 64
CHOOSE 468, 500
Clarion
 standard date
 converting to SQL 154
 standard time
 converting to SQL 154
Clarion standard date
 converting to SQL 430
CLASS 557
class
 creating
 ABC compatible 556
design 578
TYPE 557

CLEAR
 and autoincrement 514
key 513
 vs &= NULL 454

Client/server 143

CLOCK 303

CLOSE
 file 509

code abstraction 450

COMMIT 303, 305

ConcatGetComponents 475

concurrency

pessimistic
 in Oracle 307
constraints 299
 see referential integrity 6
 server-side vs client-side 296, 299
control file 18, 514, 547–562
 and FileManager 554
 GET vs SET 549
 maintaining 550
 vs INI 548
CREATE 508
CREATE DATABASE
 see SQL, CREATE DATABASE
CREATE INDEX
 see SQL, CREATE INDEX
CREATE TABLE
 see SQL, CREATE TABLE
CREATE VIEW
 see SQL, CREATE VIEW
Crystal Reports 187
cursor 307

D

data
 converting
 with DTS 341–355
 with linked server 357–369
 normalizing
 see normal form
database
 audit trigger 417
 design 1–8
 relational 3
 secure access 535–542
database synchronizer 358

DB2
 see also SQL
 trial edition 172

DCOM
 and Windows 95 184

debugging

 table opening/closing 517–525

deep assignment 451

dictionary

 changing 563–565

 IDENT 563

 synchronizer 178, 329

dictionary synchronizer 188, 190–??

dirty read 309

DLL

 calling non-Clarion 528

 data 371

 data and SQL 371

 sharing variables 596

DLLs

 splitting application into 371

driver name 176

driver string

 BINDCOLORDER 243, 255, 259

 GATHERATOPEN 189

 JOINTYPE 243

 LOGONSCREEN 189

 SAVEDSTOREDPROC 189

 TRUSTEDCONNECTION 189

 USERINNERJOIN 244

 VERIFYVIASELECT 244

drop box

 file loaded 74

DROP TABLE

see SQL, DROP TABLE

DTS

 package 342

 as stored procedure 351

 creating 347

 process flow 346

see MS SQL

 DTS

VBScript example 353

DumbDict 328

dummy table

see also temp table 383

DUP 5

DUPLICATE 586

duplicate record detection

see record, detecting duplicates

dynamic index

see index, dynamic

E

Entunnel 535–542

EVENT

 Timer 588

Excel 187

 converting to SQL 364

 reading TPS files with 100

EXPLAIN

see SQL, EXPLAIN

F

Ferrett, Scott 184

field

 default values in SQL 331

 eliminate repeating 23

 hiding on form 16

 initial value 319

 linking 5, 32

 naming convention 22

 priming on insert 15

fields

 linking

 with keys 33

fifth normal form

see normal form, fifth

file

 alias 567–576

and FileManager 573
 and NAME 572
 and RelationManager 573
 automatic 583
 buffer 574
 creating 569
 potential problems 575
 using 575
 create 508
 multiple copies 592
 open 508
 pathname 321
 file loaded drop box
 see drop box, file loaded
 FileManager
 AddKey 93
 see ABC, FileManager
 filter
 using dynamic index 95
 Firebird 147
 see also SQL
 data types 201
 first normal form
 see normal form, first
 foreign key 6, 156
 see also SQL, foreign key 156
 see key, primary
 form
 on browse window 8
 fourth normal form
 see normal form, fourth

G

GATHERATOPEN driver string 189
 GET 155, 511, 589
 and control file 514
 syntax 514
 GlobalRequest 560
 GROUP

and date/time 155

H

Hoffman, Rick 184

I

IDENT
 see dictionary, IDENT
 IIS 110
 index
 dynamic 91–96
 declaring 92
 filtering records 95
 multiple users 92
 speed 93
 use in ABC templates 93
 vs key 40
 Interbase 147
 Inventory application 319
 ISAM
 converting to SQL
 see SQL
 converting to

J

jgNextID stored procedure 401
 JOIN 159
 JOINTYPE driver string 243

K

key
 see also ee also primary key 187
 autoincrement 12, 68, 162, 163, 165, 200,
 202, 239, 266, 270, 280, 281, 287,
 321, 322, 473–478
 and templates 55–57
 client-side vs server-side 282
 in Oracle 299–303
 server-side 163, 255, 262

case sensitivity 324
 clearing 513
 foreign 6, 56
 primary 5, 40, 56, 68
 in SQL databases 69
 surrogate 32, 86
 unique 5
 using DATE() and CLOCK() 188
 vs index 40
 when to use 91
keys
 linking 33

L
 labels 567, 591
 lazy open 518
 problems with 519
LazyOpen 488
linked server
 converting data 357–??
linking field
 see field, linking
Linux
 firewalls 224
 installing 222
 MySQL 221
 window managers 223
LOGONSCREEN driver string 189
LOGOUT 303, 305

M
many-to-many
 displaying 67–83
many-to-many relationships
 see relationship, many-to-many
many-to-one relationship
 see relationship, one-to-many
MAP 557

MDAC 531
MEMBER 557
memo
 converting to SQL 180
Microsoft SQL Server
 see MS SQL
MS Excel 362
 linked server 362
MS SQL 147, 183, 317–340, 341–355, 357–369,
 371–398, 399–404, 405–414, 415–427, 441
 see also SQL
 and Windows 95 213
 autonumbering
 via stored procedure 400
 autonumbering code 325
 Books Online 334
 cascade delete 188
 CAST() 335
 connection string 325, 379
 constraints 338
 problems in Clarion 338
 CONVERT() 335
 converting to 317–340, 341–355, 357–369
 CREATE DATABASE 329, 413
 CREATE PROCEDURE 392
 CREATE TRIGGER 416
 CREATE VIEW 366
 Data Transformation Services
 see MS SQL, DTS
 data types 201
 database
 create programmatically 406
 Database Designer 333
 date
 parsing 335
 problems with 334
 date filtering 429
 DATEPART() 335
 DTS 341
 converting Oracle to Sybase 342

DTS Designer 334
Enterprise Manager 332
GETDATE() 334
IDENTITY 399–404
OLE DB provider 343
OSSQL 330
Profiler 388
Query Analyzer 332
security
 Mixed Security Mode 325
 Windows Authentication Mode 325
stored procedures 371–398
trigger 214
trusted connection 381
View Designer 333
 vs Sybase 210

MS SQL Server
 trial edition 168

MSDE 212
 trial edition 169

MyFileManager 556

MyODBC 222, 235, 243, 541
 BINDCOLORORDER driver string 243
 data source 257
 installing 235
 installing update 255

MySQL 147, 221–239, 241–??, 243, ??–245, 247–264, 265–272, 441, 443
 see also MyODBC
 see also see also MySQL, privileges 224
 see also SQL
 administering 227
 atomic operations 248
 AUTO_INCREMENT 156, 202, 206
 BDB (BerkeleyDB) tables 249
 benchmarks 263
 browse speed 271
 building from source 227
 comparison with other databases 242
 connection problems 238, 244
 constraints 264
 database
 copying to another server 265
 create 232
 InnoDB 252
 select 232
 db table 231
 foreign keys 264
 Gemini tables 249
 GRANT 256
 HEAP tables 249
 hosts table 231
 importing tables into Clarion 238
 InnoDB tables 250
 installing 225
 ISAM tables 249
 large tables 265–272
 last_insert_id() 262
 LIMIT (on select) 271
 Max
 installing 251
 MERGE tables 249
 MyISAM tables 248
 mysql_install_db 252
 mysqladmin 227, 251
 syntax 233
 mysqld process 252
 mysqldump 267
 mysqlshow 228
 ODBC
 see MyODBC
 on Linux 221
 performance 263
 privileges 231, 256
 rc.local configuration file 252
 remote access 232
 replication 242
 root user 230
 password 230
 SHOW TABLE STATUS 266
 starting 228

subselects 247
table
 copying to another server 265
 importing, problems with 258
 maximum size 241
table types 248
transactions 242, 247–??, 260, ??–264
user table 229
USERINNERJOIN driver string 244
VERIFYVIASELECT driver string 244
vs “real” SQL servers 244
vs PostgreSQL 273
vs TPS 244
MySQL
 data types 201
MySQL Manager 239

N

NAME 567, 591–598
 and file alias 572
naming convention 323
 MS SQL stored procedures 393
network traffic
 minimizing 165
NEXT 511, 512
NOCASE 177
normal form
 fifth 49–53
 first 21–27
 fourth 45–47
 second 29–37
 third 39–43
normalization 342
 see also normal form
 overdoing 86
normalized data
 displaying 55

O

ODBC
 Administrator 98
 autocommit 311
 ConfigDSN 528
 connection strings 111
 creating data sources at runtime 527–534
 ODBCINST.DLL 529
 SQLConfigDataSource 528, 532
 SQLDataSources 532
 SQLDrivers 532
 SQLInstallerError 532
 TPS driver 97–101, 110
 DATE and TIME conversions 100
 developer version problems with ASP
 110
 OLE DB providers 343
oleTclType 106
ONCE 557
one-to-many relationship
 see relationship, one-to-many
one-to-one relationship
 see relationship, one-to-one
OPEN
 access mode 509
 file 508
Oracle 147, 295–298, 299–303, 305–313, 441
 see also SQL
 ADD CONSTRAINT 296
 concurrency
 optimistic 307
 pessimistic 307
 concurrency control 307
 converting data with DTS 342
 CREATE TRIGGER 297
 FOR UPDATE 307
 IDENTITY 300
 referential integrity 295–298
 scalability 211

transaction
 read committed 309
 roll back 306
 SAVEPOINT 306
 serializable 309, 311
 transactions 305–313
 trial edition 171
 update cascade workaround 297–298
 view 312
 viewing changed records 308

ORDER BY
see SQL, ORDER BY

OROF
 vs OF 501

OVER 155

OWNER 176, 180, 189, 194, 320, 530

P

performance
 and table opening 518
 keeping tables open 518

Pervasive
 autonumbering code 328
 trial edition 172

phantom read 309

POINTER 155
 vs POSITION 515

pool data 5, 10

port forwarding 540

POSITION 590
 vs POINTER 515

PostgreSQL 147, 273–291, 441
see also SQL
 administering 286
 ALTER USER 290
 and Windows 274
 authentication 289
 book review 435
 cast operator 282

connecting remotely 287
CREATE USER 290
 createuser script 289
 customizing 436
 data types 201
 database
 connecting to 278
 creating 278
DROP SEQUENCE 282
 function overloading 282
 generator 202
 initdb 274
 nextval 281
 pg_ctl 275
 pg_hba.conf 288
 postgres.conf 288
 postmaster 275
 psql 275
 commands 276
 restarting 288
 root user 278
 security 287
 sequence number generator 280
 vs client-side autonumber 282
 setval() 284
 SSH tunneling 289
 starting 275
 stopping 275
 TCP/IP 287
 tcpip_socket 288
 triggers 285
VACUUM 436
 version 279
 vs MySQL 273

POSTQUEL 273

Pressnell, Dan
 SQL utility 405

PREVIOUS 511, 512

PRIMARY 68

primary key 5, 239

see also key, unique 187
see also SQL, primary key 155
see key, primary
 process
 alternatives in SQL 306
 PROP
 Completed 587
 Components 93
 Disconnect 181
 Filter 467
 Handle 529, 532
 Key 587
 Keys 587
 Label 464
 ORDER 212
 ProgressEvents 587
 SQL 164, 191, 213, 262, 271, 301, 381, 385,
 388, 405
 and reports/lookups 216
 procedure wrapper 409
 SQLFILTER 212, 216
 Value 470

R

range limit
 file relationship 71, 74
 RECLAIM 189
 and SQL 177
 RECORD 567
 record
 changed by another station 50
 deleting, consequences of 87
 detecting duplicates 585–590
 identifying user 86
 record initialization 473
 RECORDS 92
 referential integrity 6, 177, 295–??, 298–??, 317,
 331
see also relationship 157

and templates 580
 Cascade constraint 6
 constraints
 see constraints 295
 Delete constraint 6
 MS SQL and cascades 188
 RelationManager 483
 Restrict constraint 6
 Update constraint 6
 REGET 515
 RelationManager
 see ABC
 RelationManager
 relationship 67–83
see also referential integrity 157
 constraints 69
 defining in dictionary 14
 diagramming 11
 many-to-many 9–18, 42
 creating in dictionary 11
 described 68
 displaying with check box list 75–80
 displaying with form and lookup 69–75
 displaying with selection pool 80–83
 linking table 12
 one-to-many 6, 23–??, 23, ??–27, 581
 displaying 25
 one-to-one 49–53
 harmful 49
 repeating fields 23
 RequestCancelled 80
 RequestCompleted 80
 RESET 515, 589
 ROLLBACK 305
 Ruby, Tom
 DumbDict 328

S

SAVESTOREDPROC driver string 189

second normal form
 see normal form, second
 Secure Shell
 see also SSH 535
see normal form, third
 SELECT
 see also SQL, SELECT
 into QUEUE 395
 SELECT INTO
 see SQL, SELECT INTO
 SET 96, 510
 syntax 511
 SetThread 471
 Smart NAMED Table Theory 383
 Sorzano, Troy 383
 SQL
 see also DB2
 see also ey
 see also foreign key
 see also Firebird
 see also MS SQL
 see also MySQL
 see also Oracle
 see also PROP, SQL
 see also PROP, SQLFILTER
 see also ts
 see also constraints 157
 ALTER TABLE 266
 and CPCS Reporting Tools 193
 and data DLL 371
 and LSPack 193
 and parent-child relationship 200
 and Princen-IT Sendmail 193
 and processes 200
 and Sterling Data templates 193
 and VIEW 217
 autoincrementing values 202
 batch inserts 269
 BETWEEN 429
 BINDCOLORORDER driver string 259
 BLOBs 201
 book review 437, 441, 443
 browse speed 211
 CALL 192
 converting DATE and TIME to DATETIME 186
 converting memo 180
 converting MEMO to STRING 186
 converting to 175–181, 183–195
 CREATE DATABASE 148, 232, 329
 CREATE INDEX 287
 CREATE TABLE 153, 264, 280, 285, 352
 CREATE VIEW 313
 CSTRING vs STRING 186
 cursor 307
 data conversion program 178
 data integrity 144
 data types 153, 201–202
 Clarion equivalents 184
 conversion 154
 DATETIME 154
 TIMESTAMP 154
 database
 creating 148
 maintenance 151
 date
 converting from Clarion Standard 430
 dates 154
 dates and times 177, 202
 DATETIME vs DATETIMESTAMP 186
 default values in Clarion dictionary 331
 DROP TABLE 282, 352
 dummy table 379
 execution plan 438
 EXPLAIN 436, 444
 FASTFIRSTROW 191
 force uppercase 210
 GATHERATOPEN 189
 group in key 178
 groups in 319

index
 5
 and case sensitivity 198
 date fields in 198
 defining in dictionary 158
integer size 201
introduction 141
JOIN 161, 366
 OUTER 161
key
 autoincrement 197
 case sensitivity 177, 187
key as relationship 157
key vs index 156
LIKE 199
locator
 filtered 199
logging 151
logon procedure 193
LOGONSCREEN 189
mass update 368
mass updates 162, 198
memos 176
NOLOCK 191
NORESULTCALL 192
ORDER BY 158, 212
performance 5, 392
performance issues 198
primary key 155, 159
 see key, primary
quick scan option 194
referential integrity 163
resources 194
SAVESTOREDPROC driver string 189
scalability 145
SELECT 160
 into QUEUE 406
SELECT INTO 367
server hints 191
server-side autoincrement 162
speed advantages 143
standards 145, 202
stored procedure 191
stored procedures 164, 371–398
STRING vs CSTRING 176
subselects 147
support in Clarion 146
switching to/from TPS 159
table
 close 382
 open 382
times 154
triggers 164, 285
TRUSTEDCONNECTION 189
tuning 437
unique key 177
views
 importing 191
 vs flat file databases 142
 vs non-SQL 159
 WHERE 429
SQL Server
 see MS SQL
SQL99 442
SQLTransact 261
SSH tunneling 535–542
sshd 536
standard date
 see Clarion, standard date
standard time
 see Clarion, standard time
STATIC 593
stored procedure
 autonumbering 400
 creating 392
stored procedures 371–398
 and business logic 392
 see SQL, stored procedures
Stupid Temp Table Theory 383
surrogate key

see key
 surrogate
 Sybase 147
 and Clarion 216
 and symmetrical multiprocessing 215
 converting data with DTS 342
 trial edition 170
 trigger 214
 vs MS SQL 210
 Sybase Central 150
 synchronizer
 see dictionary, synchronizer

T

temp table
 see also dummy table 383
 smart 383
 stupid 383, 406
 template
 AyoMSSQL 379
 continuation symbols 560
 language
 #ADD 531
 #ATSTART 559
 #EXTENSION 559
 APPLICATION 559
 MULTI 559
 #PROMPT 559
 symbol
 %File 560
 %Primary 560
 symbols
 %CustomGlobalMapModule 531
 %CustomGlobalMapProcedure 531
 %CustomGlobalMapProcedureProtocol 531
 writing 558
 third normal form
 ThisWindow
 see ABC, WindowManager

TopScan 585
 total field 17
 TPS
 accessing via ASP 109–115
 combined files 594
 converting to SQL
 see SQL
 converting to
 driver error codes 119–120, 136
 file corruption 119, 125–127, 129–136
 network redirector
 detecting version 136
 disable caching 135
 installing patch 119
 ODBC driver 344
 ODBC driver, *see* ODBC, TPS driver
 performance
 multi-user 303
 super files 594
 using example files 117
 TPS files
 indexing 91
 reading with Excel 100
 TPSFix 117–118
 transaction
 and FileManager 312
 transactions
 Oracle 305–313
 trigger
 see also SQL, trigger
 autonumbering 302
 client-side 543–545
 definition 416
 MS SQL 214
 see also SQL, trigger
 server-side 543
 Sybase 214
 triggers 297
 see also SQL, triggers 164
 advantages 415

client-side 415
server-side vs client-side 415
TRUSTEDCONNECTION driver string 189

U

unique identifier 3
unique key 5, 321
see key, unique
UNTIL loop 468
user
 identifying 86
USERINNERJOIN driver string 244

V

validation 473
VBScript 109
 and DTS 353
VERIFYVIASELECT driver string 244
VIEW 158, 159, 580
 and PUT 312
 and SQL 217
view
 Oracle 312
 when to use 91
ViewManager
 see ABC
 ViewManager

W

WHAT 411, 464
WHERE
 see SQL, WHERE
WHILE loop 468
WHO 412
Windows 95
 installing DCOM updates 184

Y

YIELD 302

Z

zip code database 343
Zonkers, Screaming Yellow 51

