



# **Delphi in Depth: FireDAC**



## **Cary Jensen**

---

Delphi in Depth:  
FireDAC

by

Cary Jensen

Delphi in Depth: FireDAC

Published by Jensen Data Systems, Inc., USA.

Copyright 2017 Cary Jensen, Jensen Data Systems, Inc. World rights reserved.

The code samples and example database depicted herein are fictitious, and were created for this book as copyrighted freeware to be used by this book's readers.

The information and code samples contained or referred to in this book are provided as is, without any express, statutory, or implied warranties. Neither the author nor Jensen Data Systems, Inc. will be held liable in any way for any damages caused or alleged to be caused either directly or indirectly by this book or its contents, the code samples and example database, or any other related online content and/or files for any particular purpose.

The copyright prevents you from republishing the content, code, example database, and any online accompanying content in print, electronic media, or in any other form without the prior permission and explicit written permission of the author. No part of this publication may be stored in a retrieval system, transmitted, or reproduced in any way, in the original or in a translated language, including but not limited to photocopy, photograph, digital, magnetic, or other record without the prior permission and explicit written permission of the author.

ISBN-10: 1546391274

ISBN-13: 978-1546391272

ISBN-10: (e-book edition)

Printed copies of this book are available for sale from companies listed at, and linked to from, <http://www.JensenDataSystems.com/firedacbook>. Any other download or sale outlet is likely to be illegal. This is not a free ebook - do not distribute it.

Project Editor: Loy Anderson

Contributing Technical Editors: Dmitry Arefiev, Holger Flick, Jens Fudge, and Bruce McGee

Cover Designer: Loy Anderson

Indexer: Cary Jensen

For more information and links for purchasing this book, visit:

<http://www.JensenDataSystems.com/firedacbook>

Delphi is a trademark of Embarcadero Technologies. Windows is a trademark of Microsoft Corporation. Advantage Database Server is a trademark of SAP.

Other product and company names mentioned herein or in accompanying online material may be the trademarks of their respective owners.

Chapter Titles v

# **Chapter Titles**

<b>Chapter Titles</b>	.....
v <b>Table of Contents</b>	.....
vii <b>About the Author</b>	.....
xvii <b>About the Technical Reviewers</b>	.....
<b>Acknowledgements</b>	.....
<b>xxi Introduction</b>	.....
<b>1</b>	.....
<b>Chapter 1 Overview of FireDAC</b>	..... 5
<b>Chapter 2 Connecting to Data</b>	..... 15
<b>Chapter 3 Configuring FireDAC</b>	..... 47
<b>Chapter 4 Basic Data Access</b>	..... 81
<b>Chapter 5 More Data Access</b>	..... 109
<b>Chapter 6 Navigating and Editing Data</b>	..... 145
<b>Chapter 7 Creating Indexes</b>	..... 165
<b>Chapter 8 Searching Data</b>	.....
<b>197</b>	.....
<b>Chapter 9 Filtering Data</b>	.....
<b>217</b>	.....
<b>Chapter 10 Creating and Using Virtual Fields</b>	.....

.....	259
<b>Chapter 11 Persisting Data</b>	
.....	297
<b>Chapter 12 Understanding FDMemTables</b>	
.....	329
<b>Chapter 13 More FDMemTables: Cloned Cursors and Nested DataSets</b>	
.....	369
<b>Chapter 14 The SQL Command Preprocessor</b>	
.....	397
<b>Chapter 15 Array DML</b>	
.....	
425	
<b>Chapter 16 Using Cached Updates</b>	
.....	439
<b>Chapter 17 Understanding Local SQL</b>	
.....	487
<b>Appendix A Code Download, Database Preparation, and Errata</b>	
.....	507
<b>Index</b>	
.....	
519	
Table of Contents	vii
<b>Table of Contents</b>	
Dedication	
.....	
3	
<b>Chapter Titles</b>	
.....	
<b>v Table of Contents</b>	
.....	
<b>vii About the Author</b>	
.....	
<b>xvii Cary Jensen</b>	
.....	
<b>xvii About the Technical Reviewers</b>	
.....	xix

Dmitry Arefiev	.....
xix Holger Flick	.....
xix Jens Fudge	.....
xx Bruce McGee	.....
<b>xx Acknowledgements</b>	.....
<b>xxi Introduction</b>	.....
<b>1</b>	
Who Is This Book For	..... 2
Conventions	.....
2	
<b>Chapter 1 Overview of FireDAC</b>	..... 5
FireDAC Features	.....
6	
<i>Cross-Platform Support</i>	..... 7
<i>Exceptional Support for Databases</i>	..... 7
<i>Flexible Queries Using the SQL Command Preprocessor</i>	..... 8
<i>Blazing Performance with Array DML</i>	..... 8
<i>Support for a Variety of Query Execution Modes</i>	..... 9
<i>Powerful Monitoring Capabilities</i>	..... 9
<i>Cached Updates</i>	..... 10

<i>Result Set Persistence</i>	10
<i>Data Type Mapping</i>	11
<i>Local SQL</i>	11
<i>Additional Features</i>	12
Connection Recovery	12
Advanced Transaction Support	12
Built-In Dialog Support	12
Support for Database-Specific Services	12
Customizable Data Access	13
Batch Move Support	13
viii Delphi in Depth: FireDAC	
Written in Object Pascal	
13	
<b>Chapter 2 Connecting to Data</b>	15
Creating Unnamed Connections	15
<i>Creating Temporary Connections</i>	

.....	16
<i>Defining a Temporary Connection Using FDConnection.Params</i>	
.....	26
<i>Defining a Temporary Connection at Runtime</i>	
.....	27
<i>Creating Named Connections</i>	
.....	28
<i>Creating Named Connections Definition Using the Database Explorer</i>	
.....	29
<i>Creating a Named Connection Definition Using the FireDAC Explorer</i>	
.....	33
<i>Creating a Persistent Connection</i>	
.....	36
<i>Creating a Private Connection</i>	
.....	39
<i>Connecting to Any Database Using ODBC</i>	
.....	41
<b>Chapter 3 Configuring FireDAC</b>	
.....	47
<i>Share Configuration Properties</i>	
.....	48
<i>Configuration Property Inheritance</i>	
.....	48
<i>Overriding Individual Configuration Properties</i>	
.....	50
<i>Restoring Configuration Property Inheritance</i>	
.....	50
<i>Configuring the Shared Properties</i>	
.....	51
<i>Fetch Options</i>	
.....	54
<i>General Fetching</i>	
.....	56

Items to Fetch	57
Items to Cache	57
Master-Detail	57
Live Data Window	58
<i>Format Options</i>	58
Data Mapping Rules	60
Handling BCD/FmtBCD Type and DataSnap Compatibility	62
Handling String Type	62
Parameters Default Type	63
Value Preprocessing	63
Dataset Sorting	64
Quotation Identifier	64
Default Field Formats	

65	
<i>Resource Options</i>	65
Command Text Processing	
67	
Persistence Mode	
68	
Command Execution	
69	
Connection Resources	
69	
<i>Update Options</i>	
70	
Table of Contents ix	
General	
Updating	
72	
Locking	
72	
Refreshing	
73	
Automatic Incrementing	
73	
Posting Changes	
74	
<i>Transaction Options</i>	75
Isolation Level	

.....	
76	
Update Ability	.....
.....	
76	
Automatic Committing	.....
.....	
77	
DBMS-Specific Parameters	.....
.....	
77	
Disconnection Action	.....
.....	
78	
Nesting	.....
.....	
78	
Understanding UpdateOptions.UpdateMode	.....
.....	78
<b>Chapter 4 Basic Data Access</b>	.....
.....	<b>81</b>
The User Interface and Data Binding	.....
.....	85
<i>Navigation and VCL Data Links</i>	.....
.....	86
The DBNavigator	.....
.....	
86	
Multi-Record VCL Controls and Navigation: DBGrid and DBCtrlGrid	.....
.....	89
<i>Navigation and LiveBindings</i>	.....
.....	91
<i>The BindNavigator</i>	.....
.....	92
<i>Position-Related LiveBindings</i>	.....
.....	94

Understanding FDTable	97
.....	97
Configuring an FDTable	97
.....	97
Executing Datasets at Design Time	99
.....	99
Executing DataSets at Runtime	100
.....	100
When Should You Connect?	100
.....	100
Live Data Window	102
.....	102
Executing Queries and Stored Procedures	103
.....	103
Executing Queries	103
.....	103
Executing Stored Procedures	105
.....	105
<b>Chapter 5 More Data Access</b>	<b>109</b>
.....	109
Parameterized Queries and Stored Procedures	109
.....	109
The Advantages of Parameters	110
.....	110
Greater Flexibility	110
.....	110
Improved Performance	110
.....	110
Prevention of SQL Injection	111
.....	111
Defining Parameters at Design Time	112
.....	112

<i>Parameterized FDQueries and the Query Editor</i> .....	115
x Delphi in Depth: FireDAC	
<i>Defining Parameters at Runtime</i> .....	118
Taking Control of Updates: FDUpdateSQL	
.....	121
<i>Defining the Query</i> .....	122
<i>Configuring the DataSet</i> .....	124
<i>Creating the Parameterized FDUpdateSQL Queries</i> .....	128
FDCommand, FDTableAdapter, and FDMemTable	
.....	130
Managing Transactions	
.....	133
<i>Implicit and Explicit Transactions</i> .....	133
<i>Transaction Isolation</i> .....	138
Dirty Read Isolation	
.....	139
Read Committed Isolation	
.....	139
Repeatable Read Isolation	
.....	139
<i>Nested Transactions</i> .....	139
Asynchronous Queries	
.....	140
Monitoring FireDAC Queries	

.....	141
<b>Chapter 6 Navigating and Editing Data</b>	
.....	<b>145</b>
Understanding Fields	
.....	145
The Current Record.....	
148	
Detecting Changes to Record State.....	148
<i>Calculating Performance</i>	
.....	152
Navigating Programmatically	
.....	153
<i>Basic Navigation</i>	
.....	154
<i>Have I Gone Too Far? Bof and Eof</i>	
.....	154
<i>Using MoveBy</i>	
.....	155
<i>Navigating Using RecNo</i>	
.....	156
<i>Scanning a FireDAC DataSet</i>	
.....	157
<i>Disabling Controls While Navigating</i>	
.....	157
<i>Navigating Using Bookmarks</i>	
.....	160
<i>Editing a DataSet</i>	
.....	161
<b>Chapter 7 Creating Indexes</b>	
.....	<b>165</b>
Index Overview	
.....	

166	
Temporary Indexes	167
.....	
<i>An Example of Creating Temporary Indexes at Runtime</i>	
.....	168
<i>Temporary Indexes and FormatOptions.SortOptions</i>	
.....	171
Persistent	
Indexes.....	
172	
<i>Defining Persistent Indexes</i>	
.....	173
<i>Creating Field-Based Indexes</i>	
.....	178
Creating a Field-Based Index at Design Time	
.....	178
Selecting an Index at Design Time	
.....	
179	
Creating a Field-Based Index at Runtime	
.....	180
Table of Contents xi	
Selecting an Index at Runtime	
.....	
180	
<i>Creating Expression Indexes</i>	
.....	182
<i>Creating Distinct Indexes</i>	
.....	184
<i>Creating a Filter-Based Index</i>	
.....	186
Two Runtime Index Examples: Sorting a DBGrid On-The-Fly .....	
190	
<b>Chapter 8 Searching Data</b>	
.....	

<b>197</b>	
<b>Searching FireDAC DataSets</b>	
.....	197
<i>Simple Record-by-Record Searches</i>	
.....	199
<i>Searching with Indexes</i>	
.....	200
<b>Finding Data</b>	
.....	
201	
<b>Going to Data</b>	
.....	
204	
<i>Searching with Variants</i>	
.....	207
<b>Locating Data</b>	
.....	
208	
<b>Using Lookup</b>	
.....	
213	
<b>Chapter 9 Filtering Data</b>	
.....	
<b>217</b>	
<b>Filters</b>	
.....	
217	
<i>Using a Range</i>	
.....	
222	
<b>Setting Ranges</b>	
.....	
222	
<i>Using ApplyRange</i>	
.....	
228	

Cancelling a Range	.....
231	.....
A Comment About Ranges	.....
231	.....
Using Filters	.....
232	.....
<i>Basic Filters</i>	.....
233	.....
<i>Special Filter Expressions</i>	..... 234
Null Comparisons	.....
236	.....
String Functions	.....
238	.....
Date/Time Functions	.....
239	.....
Miscellaneous Functions	.....
241	.....
Escaping Wildcard Characters	.....
242	.....
FireDAC Scalar Functions	.....
243	.....
<i>Other Filter-Related Properties</i>	..... 243
<i>Using the OnFilterRecord Event Handler</i>	..... 245

<i>Navigating Using a Filter</i>	246
<i>Using Ranges and Filters Together</i>	249
Master-Detail Views: Dynamically Filtered Detail Tables	
..... 250	
<i>Defining a Range-Based Dynamic Master-Detail Relationship</i>	
..... 252	
<i>Defining a Parameter-Based Dynamic Master-Detail Relationship</i>	
..... 254	
<i>Detail Records and the FetchOptions.DetailDelay Property</i>	
..... 258	
xii Delphi in Depth: FireDAC	
<b>Chapter 10 Creating and Using Virtual Fields</b>	
..... 259	
Aggregate Fields	
..... 260	
<i>Creating Aggregate Fields</i>	
..... 261	
Adding the Aggregate Field	
..... 264	
Defining the Aggregate Expression	
..... 266	
Setting Aggregate Index and Grouping Level	
..... 268	
Making the Aggregate Field Available	
..... 269	
Turning Aggregates On and Off	
..... 272	
<i>Creating Aggregate Collection Items</i>	
..... 272	

Understanding Group State	274
Creating AggregateFields at Runtime	277
Calculated Fields	
280	
<i>InternalCalc Fields and FireDAC Scalar Functions</i>	
283	
Lookup Fields	
285	
Understanding FieldOptions	292
<i>The FieldOptions Property</i>	293
<i>Combine Options and PositionMode</i>	295
<i>The UpdatePersistent FieldOptions Property</i>	295
<i>Field and Fields Properties</i>	296
<b>Chapter 11 Persisting Data</b>	297
Persisting Data to Files	298
<i>Saving to Files</i>	
299	
File formats	
301	
Storing Human-Readable Data	
303	

What to persist	305
Persisted Version Information	308
<i>Loading from Files</i>	309
File Formats	312
Merging Data When Loading	312
<i>Automating Persistence</i>	316
<i>Maintaining Backups</i>	317
Persisting to Streams	321
<i>Saving to a Stream</i>	322
<i>Loading from a Stream</i>	324
Persistence and FDSchemaAdapters	326
<b>Chapter 12 Understanding FDMemTables</b>	<b>329</b>
The Role of FDMemTable in FireDAC Applications	330
Defining an FDMemTable's Structure	331
Table of Contents	xiii
<i>Defining Structure Using FieldDefs</i>	332

Creating FieldDefs at Design Time	.....
332	
Creating FieldDefs at Runtime	.....
336	
<i>Defining an FDMemTable's Structure Using Fields</i>	.....
339	
Creating Fields at Design Time	.....
339	
Creating Fields at Runtime	.....
345	
Loading FDMemTables Directly From DataSets	.....
347	
<i>Loading an FDMemTable Using CopyDataSet</i>	.....
348	
Other FDDataSets and CopyDataSet	.....
359	
<i>Copying Data Using the FDDataSet Data Property</i>	.....
362	
Editing Data at Design Time	.....
364	
<b>Chapter 13 More FDMemTables: Cloned Cursors and Nested DataSets</b>	.....
369	
Cloning and FDMemTables	.....
369	
<i>Master with Detail Clone</i>	.....
371	
<i>Editing with Cloned Cursors and Cached Updates</i>	.....
375	
Creating Nested DataSets	.....
379	

<i>Defining Nested DataSets at Design Time</i>	380
<i>Defining Nested DataSets at Runtime</i>	384
Using FieldDefs at Runtime	
385	
Using Fields at Runtime	
389	
<i>Final Thoughts About Nested DataSets</i>	393
<b>Chapter 14 The SQL Command Preprocessor</b>	397
Macro Substitution	400
Escape Sequences	406
Constant Substitution	407
Identifier Substitution	408
Conditional Substitution	409
FireDAC Scalar Functions	412
FireDAC String/Character Scalar Functions	415
Numeric Scalar Functions	
417	
Date/Time Scalar Functions	
419	
System Scalar Functions	

.....	
421	
The Convert Scalar Function	
.....	
422	
Special Character Processing	
.....	423
<b>Chapter 15 Array DML</b>	
.....	
425	
Using Array DML	
.....	426
Array DML Mode and Errors	
.....	434
xiv Delphi in Depth: FireDAC	
<i>Handling Array DML</i>	
<i>Limits</i> .....	437
<b>Chapter 16 Using Cached Updates</b>	
.....	439
Cached Updates: The Basics	
.....	441
<i>Entering and Exiting Cached Updates</i>	
.....	441
<i>Detecting the Cache State</i>	
.....	442
<i>Record Status and Change Filters</i>	
.....	444
<i>What Has Changed?</i>	
.....	447
<i>Persisting the Cache</i>	
.....	448
Managing the Cache	
.....	449
<i>Canceling the Last Change</i>	
.....	449

<i>Cancelling a Specific Change</i>	450
<i>Cancelling All Updates</i>	450
<i>Clearing the Cache Without Cancelling Changes</i>	451
<i>Using Save Points</i>	452
<i>Applying Updates</i>	453
<i>Apply Updates Using Brute Force</i>	453
<i>Calling the ApplyUpdates Method</i>	453
The ApplyUpdates Method in Automatic Mode	457
ApplyUpdates and OnUpdateRecord	463
<i>ApplyUpdates and Errors</i>	467
Managing Errors Following the Call to ApplyUpdates	467
Managing Errors Using Event Handlers	469
Understanding Centralized Cached Updates	477
<i>Managing Master-Detail Tables in the Centralized Cached Updates Model</i>	481
<b>Chapter 17 Understanding Local SQL</b>	487
Implementing Local SQL	488
<i>The Initial Setup</i>	488

<i>Configuring the SQLite Connection</i>	492
<i>Configuring the Datasets</i>	493
<i>Configuring FDLocalSQL</i>	495
Activating Local SQL Queries at Runtime	
.....	499
<i>Some Comments on FDLocalSQL</i>	
.....	504
<b>Appendix A Code Download, Database Preparation, and Errata</b>	
.....	507
<i>Code Download</i>	
.....	
507	
<i>Database Preparation</i>	
.....	509
<i>Installation</i>	
.....	
509	
<i>The DataPaths Unit</i>	
.....	509
<i>Using SharedDMVcl</i>	
.....	512
<i>The EMPLOYEE Named Connection</i>	
.....	513
Table of Contents	xv
<i>InterBase UDF</i>	
<i>Definitions</i>	
.....	515
Errata	
.....	
517	
<b>Index</b>	
.....	
519	

About the Author xvii

## **About the Author**

### **Cary Jensen**

Cary Jensen is a best-selling author of more than twenty-five books on software development. He is also an award-winning trainer and a popular speaker at software conferences throughout much of the world. An Embarcadero Delphi MVP, Cary has provided training to companies on every version of Delphi and has toured and trained on every version, starting with the original Delphi World Tour in 1995, and continuing through the Delphi 5 World Tour. His other tours include Delphi Development Seminars (2000), Borland Development Seminars (1998-2001), and Delphi Developer Days (2001 to present).

Cary is Chief Technology Officer at Jensen Data Systems, Inc., a company that has been providing training, consulting, and software development services since 1988. He is an active developer, providing clients with assistance in data modeling, software architecture, software development, team development,

mentoring, training, and software migration. Dr. Jensen has a Ph.D. in Engineering Psychology, from Rice University, specializing in human-computer interaction.

Twitter: <http://twitter.com/caryjensen/>

Blog — Let's Get Technical: <http://caryjensen.blogspot.com/>

Company web site: <http://www.JensenDataSystems.com/>

LinkedIn: <https://www.linkedin.com/in/cary-jensen-31a921>

Email: [info@jensendatasystems.com](mailto:info@jensendatasystems.com)

About the Technical Reviewers xix

## **About the Technical**

### **Reviewers**

#### **Dmitry Arefiev**

Dmitry Arefiev is the creator of AnyDAC, the product that eventually became FireDAC. He is currently the FireDAC architect for Embarcadero Technologies, the makers of RAD Studio and Delphi.

Email: darefiev@gmail.com

### **Holger Flick**

Dr. Holger Flick is a well-known member of the Delphi Community, and has worked with Delphi and Borland Pascal before Delphi. While achieving a Degree in Computer Science and a Doctorate of Engineering, he was part of several developer teams at Borland and later CodeGear. This gave him the means to gain first-hand knowledge of the tools and frameworks. He wrote several articles and spoke at many Delphi Road Shows, seminars, and conferences. When developing software with Delphi, his focus is on database-driven applications for both desktop and mobile platforms. Since 2016, Holger heads his new brand “Flix Engineering” and is available for training, development, and consulting services.

URL: <https://flixengineering.com/blog>

Twitter: <https://twitter.com/hflickster>

LinkedIn: <https://de.linkedin.com/in/hflick>

Email: [info@flixengineering.com](mailto:info@flixengineering.com)

xx Delphi in Depth: FireDAC

### **Jens Fudge**

Jens Fudge has been working with Delphi since 1995, when it first came out. He has built mainly database systems for a lot of various customers in different areas like railroad companies, airports, cement factories and even a government application. Jens is an Embarcadero Delphi MVP, and works as a trainer and consultant for many different companies, and is also a frequent speaker at international and national conferences. Apart from being a Delphi developer and consultant, Jens also brews beer, wine and mead, and shoots archery. The latter has inspired Jens to the name of his company, which is Archersoft. Jens won the Gold medal in archery at the Paralympic Games in Barcelona, Spain in 1992.

Email: [Jens.fudge@archersoft.dk](mailto:Jens.fudge@archersoft.dk)

### **Bruce McGee**

Bruce McGee operates a software consulting company named Glooscap Software in Toronto, Ontario, Canada. He has been a user of and advocate for Delphi for many years, and continues to work with it daily. He is also a big

fan of continuous learning, especially in the software development field, and the need to constantly hone our craft.

Blog: <http://www.glooscap.com/>

Acknowledgements xxi

# Acknowledgements

Software development is an ironically solitary task that typically involves a lot of people. In addition to the many hours that we spend in front of our computers, using our skills to create value out of thin air, we rely heavily on the talents of others, from the designers who write the specifications, to the quality assurance folks who ensure that our work is up to the task, from the managers who make sure that we receive the necessary resources to do our jobs, to our client's whose needs make our efforts so worthwhile.

In that respect, writing a book is very similar. In addition to those countless hours of research and writing, each author depends on the support and input of many people in getting a book to press. And this project is no different.

To begin with, I want to thank my wife and business partner, Loy Anderson, who has become a tireless editor and organizer. Without her efforts this book would never see the light of day. Not only does she endlessly read copy for readability, spelling, and grammar, but she also acts as publisher, working with the print and ebook publishers to ensure that the final product is one of which we can all be proud.

Next, I want to thank this book's technical reviewers. To begin with, I want to thank Dmitry Arefiev, the creator of FireDAC. I am so grateful that he was able to review many of the chapters of this book, especially the more complex chapters. His comments and corrections ensured the accuracy of so many technical details. And he did this while working for the release of RAD Studio 10.2 Tokyo. I also want to note that his commitments to Embarcadero Technologies prevented him from reviewing every chapter prior to this book heading to the printers, so you have me to blame for those remaining inaccuracies. I will strive to resolve those using the errata, whose URL appears in Appendix A.

Next I want to thank Dr. Holger Flick. He contacted me via LinkedIn last fall with questions regarding my last book: *Delphi in Depth: ClientDataSets*. His questions led me to choose FireDAC as the topic of this book, and I began working on it as soon as I completed the Delphi Developer Days 2016 course book (which I wrote with Nick Hodges, Director of Engineering at Embarcadero Technologies). Holger even offered to be a technical reviewer, an offer that I could not refuse. In addition to reviewing the book, he is planning on translating this book into German, something that I find very

exciting.

## xxii Delphi in Depth: FireDAC

I also want to thank Jens Fudge of ArcherSoft. Like Holger, Jens offered many suggestions and observations that helped improve the overall quality of the book. I am deeply indebted to both of them. Finally, I want to thank Bruce McGee of Glooscap Software. His comments and suggestions were also quite helpful, and he showed a remarkable talent for catching wording errors that managed to escape the rest of us.

I also want to acknowledge the help, encouragement, and advice of my many friends and colleagues. To begin with, I want to thank Jim McKeeth, Chief Developer Advocate and Engineer at Embarcadero Technologies. In addition to his encouragement, Jim always ensured that I had the latest information on RAD

Studio.

I also want to thank my Delphi Developer Days co-presenters, including Bob (Dr.Bob) Swart of Bob Swart Training and Consultancy, Ray Konopka of Raize Software, and most recently, Nick Hodges of Embarcadero Technologies. It has been a great pleasure to write and present with these pillars of the Delphi community.

I also want to give a great big thanks to my long-time friend, and former Delphi Developer Days co-presenter, Marco Cantù, the RAD Studio Product Manager

at Embarcadero Technologies. I've known Marco for more than 20 years, and we share both a love for writing and a love for Delphi. I am grateful that we have such a thoughtful and dedicated person leading Delphi into the future.

I also want to thank the development teams at Embarcadero Technologies, from Nick Hodges and Dmitry Arefiev on down. Delphi (and C++Builder) are

wonderful products, and I want these people to know that we appreciated their work.

Finally, I want to thank the many members of the Delphi community. We're in this together, and your commitment to this remarkable product makes the arduous task of writing books like this one worth the effort.

Introduction 1

# Introduction

I remember when I first heard about Delphi. It was 1993, and I was serving on the Paradox Advisory Board for the annual Borland International Conference.

After a presentation by Philippe Kahn at the Borland International headquarters in Scotts Valley, California, in which he introduced us to this next generation Pascal compiler and IDE (Integrated Development Environment), one of the

other board members turned to me and said that Delphi was the compiler that we Paradox developers had been waiting for.

What this story reveals is that I entered the Delphi community as a database developer. In fact, I was a columnist for The Delphi Informant Magazine, from the very first issue, and my column was named DBNavigator, reflecting my

emphasis on multiuser databases. I continue to share this enthusiasm for database development to this very day.

As a Delphi database developer, I was keenly aware of the great data-related breakthroughs introduced in Delphi. Though considered an obsolete technology now, when it was introduced, the Borland Database Engine was ahead of its

time. Delphi also introduced the data module (Delphi 2), and the ClientDataSet (Delphi 3 Client/Server).

I also had my share of disappointments, namely dbExpress (Kylix and Delphi 6). While dbExpress was fast, it was more of a C-like, pass-through SQL framework that violated many of the principles of the TDataSet interface. When SQL Links for Windows, and subsequently the BDE itself, were deprecated, it felt like database developers had to turn to third-party solutions in order to continue working with databases using the *Delphi way*.

That changed in the spring of 2013, when Embarcadero released RAD Studio XE3. Embarcadero had acquired AnyDAC, and re-branded it as FireDAC. FireDAC was the perfect replacement for the BDE, supporting both the TDataSet interface as well as a large number of database servers. Finally, Delphi database developers had a flexible, powerful, and easy to use framework for working with data.

Over the intervening years, FireDAC has matured and improved, eventually becoming the unequivocal Delphi data access mechanism of choice. I used it in my preceding book, *Delphi in Depth: ClientDataSets Second Edition*, and I have used it exclusively for database examples for Delphi Developer Days material.

## 2 Delphi in Depth: FireDAC

over the last three years. And now, I have expressed my confidence in FireDAC

by writing this book.

### **Who Is This Book For**

This book is intended for the Delphi database developer. In it you will find information at nearly every level of application development. If you are new to database development in Delphi, you will find basic information about how the TDataSet interface works. For example, how to navigate records, the concept of the current record, accessing fields, and how to edit data.

If you are an advanced database developer, you too will find valuable information. For example, how to define dynamic master-detail relationships, the convenience of nested datasets, and the power of cached updates.

In order to use the examples found in this book you will need to be using Delphi XE6 or later, and ideally Delphi 10 Seattle or later. At a minimum, you will need the professional version of these products, and will also need to install either the InterBase server or the InterBase developer edition. There are a few more requirements, and you will find out more about these in Appendix A, which you should read before continuing to *Chapter 1: Overview of FireDAC*.

### **Conventions**

Most of the examples in this book make use of FDQuery components, which are used to execute SQL (Structured Query Language) statements. In this book, I am pronouncing SQL as “es”-“que”-“el,” and not “sequel.” What this means is that I will say “an SQL statement,” instead of “a SQL statement.”

Another convention that I use is to drop the T in most references to a class. For example, while I will occasionally speak strictly about a class, say TFDQuery, I will most often refer to instances of this class as FDQueries, and then more conversationally as “queries.” My main goal is readability. To me, the constant use of the T in a class name makes the text harder to read.

Another convention relates to the sample projects that accompany this book.

In almost every case, when I show a code segment, it is code that can be found in a sample project from the code download. The first time I refer to a given project in a chapter, I include a note indicating the name of the project as it appears in the code download. I do not repeat this note in subsequent references to that project in the same chapter.

## Introduction 3

Another convention concerns how screenshots are referenced. This book includes both figures and illustrations. All figures are numbered, and include a caption. Illustrations are not numbered, and do not include captions. Illustrations are used for small screenshots that are discussed in the text that immediately precedes the screenshot. By comparison, figures may not appear on the same page from which they are referenced, and may be referred to again later in the chapter. It's a minor point, but one that I want to make in case you start wondering why some screenshots lack a caption.

There is one last thing. This book is about techniques involving FireDAC. And while FireDAC itself is cross-platform, almost every one of the sample projects is a VCL (Visual Component Library) example that runs only on Windows.

Since Delphi is a Windows-based IDE, it is guaranteed that every reader of this book will be running Windows. Writing FireMonkey applications for iOS,

Android, or OSX (Mac) involves additional technologies, and I didn't want to get bogged down with discussions of LiveBindings (which I do cover), the platform assistant, and FireMonkey component configuration. I know that some readers will be unhappy about this decision, but I wanted you to know that I had my reasons.

## Chapter 1: Overview of FireDAC 5

# Chapter 1

## Overview of FireDAC

FireDAC is a comprehensive collection of components that implement Delphi's traditional TDataSet interface. In this respect, it is comparable to the Borland Database Engine (BDE), dbExpress, InterBase Express, and dbGo, RAD

Studio's TDataSet components for ActiveX Data Objects (ADO).

Embarcadero added FireDAC to Delphi in the spring of 2013 after acquiring AnyDAC from DA-Soft Technologies, and it first appeared in Delphi XE3. At that time, the components still used the AD prefix. By Delphi XE5, the component prefix was formally changed to FD, reflecting the FireDAC moniker.

When Delphi first shipped it had one data access framework, the BDE, although at the time it was referred to as ODAPI (Open Database Application Programming Interface). ODAPI was initially released with Quattro Pro for Windows, and later Paradox for Windows. It was later renamed IDAPI (Independent Database API), and then again as IDAPI (Integrated Database API). Even these technologies were an outgrowth of the Paradox Engine, a DOS

set of overlays which, when released back in September of 1991, represented one of the first times that the table and record locking mechanisms of a database (Paradox, of course) was made available to developers using other tools, which in this case were Turbo Pascal and Turbo C++.

While the BDE was a breakthrough technology in its early years, providing a fast, independent data access layer, it was cumbersome to install, used a lot of network bandwidth, and had limited support for remote database servers, this being provided by a set of DLLs (dynamic link libraries) referred to as Borland SQL Links for Windows. Over time, it became increasingly obsolete.

The need for a new data access mechanism for Delphi became even more apparent during the development of Kylix, a Delphi-based compiler and IDE (integrated development environment) for Linux. Porting the BDE to Linux was ruled out, and dbExpress was born. dbExpress is a high-speed

client/server data access framework based largely on pass-through SQL (Structured Query

Language).

## 6 Delphi in Depth: FireDAC

The dbExpress framework had one major drawback, however. In most cases converting a BDE project to dbExpress required a major refactoring of the data access logic, and dbExpress did not support the old-style file server databases such as Paradox, dBase, or MS Access until Delphi XE2. As a result, for many BDE developers, dbExpress was a poor option.

To make matters worse, Borland SQL Links for Windows, the drivers that supported remote database servers such as Oracle, InterBase, SQL Server, and the like, were deprecated. This happened shortly after the release of dbExpress.

Deprecation of the BDE itself followed a few years later, leaving Delphi developers without a good migration path without going outside of the product.

This has changed with the introduction of FireDAC. Conversion from the BDE

to FireDAC is more or less smooth, and Delphi even ships with a tool, named reFind, that helps with much of the conversion process. It is for this reason that I use FireDAC in all of my new projects (and in all of the database projects included in my source code contributions to Delphi Developer Days). In

addition, I think that a good argument can be made for migrating legacy applications to use FireDAC when a major revision is scheduled. Yes, it is that good.

In this chapter, I am providing a high-level overview of FireDAC, and here I will introduce most of FireDAC's more interesting features. Many of these features are discussed in depth in later chapters of this book.

### **FireDAC Features**

There are many reasons to use FireDAC, and a lot of these have to do with the feature set that FireDAC brings to the table. While FireDAC supports capabilities similar to other TDataSet collections available in Delphi, there are a number of features that are unique to FireDAC, and which make FireDAC

extremely attractive.

To begin with, FireDAC does an exceptional job supporting the TDataSet interface, going so far as to introduce some of the advanced features found in only some of the TDataSet implementations (ClientDataSet, for example). I've already made this point, so I won't belabor it further. But there is much more to FireDAC than simple TDataSet conformity. FireDAC supports an exceptional

collection of features and capabilities that make it the clear choice for implementing data access in Delphi applications.

For the remainder of this chapter, I am going to highlight the many features of FireDAC that really make it shine. For those more involved features, I am going

## Chapter 1: Overview of FireDAC 7

to keep these descriptions at a high level, saving the specific details for later chapters where the feature is examined in more detail.

### Cross-Platform Support

FireDAC is supported on all of Delphi's platforms. You can use FireDAC in Windows applications, Mac OS applications, iOS, Android, and Linux applications, and where RAD Studio supports 64-bit compilation, both 32-bit and 64-bit versions.

To what extent a given platform is supported depends to some extent on the availability of an appropriate client library. For example, all databases are supported on the Windows platform, in part because every database vendor creates a client API for Windows. There are few client APIs, however, for the mobile platforms. For example, both InterBase and SQLite are supported on both iOS and Android. However, many databases do not publish a client library for the iOS or Android ARM platforms.

The good news is that even though some platforms have limited client libraries, you can always use DataSnap or RAD Server (previously called Embarcadero

Mobility Services, or EMS) to serve data to your mobile devices.

### Exceptional Support for Databases

FireDAC provides drivers for almost every major relational database, both

commercial and open source. On the commercial side, you find native support for Oracle, IBM DB2, MS SQL Server, InterBase, SAP SQL Anywhere, SAP Advantage Database Server, and Teradata. Open source databases natively supported by FireDAC include Firebird, MongoDB, MySQL, PostgreSQL, and SQLite.

If you don't find your particular database in that list, there's no need to worry. FireDAC supports two bridging drivers, one for ODBC (Open DataBase Connectivity) and another for dbExpress. Using the FireDAC ODBC driver permits you to work with virtually any relational database in existence, as ODBC is about as universal as it gets. Even COBOL data files are supported by ODBC.

What's particularly interesting about FireDAC's native drivers is that many of these provide you with access to features specific to the associated database. For example, FireDAC supports the return of multiple result sets from an SQL

command execution against those databases that support that feature, such as InterBase, MS SQL Server, Oracle, and PostgreSQL. Likewise, FireDAC's native drivers support database alerts (notifications or events).

## 8 Delphi in Depth: FireDAC

It should be noted that full support for the databases listed above requires that you have an Enterprise-level Delphi license or higher, or that you have purchased the FireDAC Client/Server Add-On Pack for Professional-level RAD

Studio products. RAD Studio Professional (as well as Delphi Professional and C++Builder Professional) is licensed only for local file server database access, such as MS Access, Paradox, SQLite, and Advantage Local Server. Fortunately, Delphi Professional (and C++Builder Professional) come with both the

InterBase developer edition and the FireDAC InterBase driver, so you can use those versions with this book without having to first buy the FireDAC Client/Server Add-On Pack

### **Flexible Queries Using the SQL Command Preprocessor**

The SQL command preprocessor, formerly called *Dynamic SQL*, permits you

to write flexible SQL that includes macros, conditional substitution, identifier substitution, and ODBC-like escape functions. The SQL command preprocessor can perform a variety of manipulations on your SQL before it sends that SQL to the underlying database.

One of the more useful applications of the SQL command preprocessor is to permit you to write one set of queries that can be executed against a variety of databases, even when those databases support different dialects of SQL. For example, the {`CONVERT(...)`} escape function will be expanded into the `TO_CHAR` keyword when executing against an Oracle database, but into `CONVERT` when connected to Microsoft SQL Server.

Statements that use conditional substitution look similar to the `{$IF}` conditional compilation compiler directives in the Delphi language, though the substitution is not performed at compile time. Instead, this operation occurs at runtime, and is performed by the SQL command pre-processor, as part of SQL command

preparation and right before the query is submitted to the underlying database. The SQL command preprocessor has utility beyond supporting multiple databases. For example, the use of macro substitution in your SQL statements permits you to write queries whose tables, fields, or WHERE clause predicates are not known until runtime. (A predicate is a Boolean expression used in SQL, such as in WHERE clauses, HAVING clauses, and joins). All you need to do is ensure that you have bound valid values to those macros before executing the query, and the command preprocessor will take care of updating the resulting SQL.

## **Blazing Performance with Array DML**

Array DML (Data Manipulation Language, a category of SQL operations), provides a mechanism that supports high-speed data manipulation using Chapter 1: Overview of FireDAC 9

parameterized query and stored procedure execution. This feature is useful when you have a single parameterized query or stored procedure that needs to be executed repeatedly, with each execution using different values in the parameters.

To use Array DML, you create a parameterized SQL statement, along with an array of parameters, with one row of the array of parameters for each instance of the query or stored procedure execution. When executing, FireDAC sends

both the parameterized query and the array of parameters to the database engine, which is then responsible for performing the parameter binding and query

executions one time for each element in the array. The alternative is to execute your queries one at a time, manually binding the next set of parameters prior to each subsequent execution.

Array DML can produce unparalleled performance, benefiting from a reduction in network traffic, database roundtrips, and delegating the parameter binding to the server. The advantages of Array DML are particularly beneficial for extract, transform, load (ETL) operations, such as those often associated with data warehousing.

Performance varies by database, as some databases natively support the operations performed by Array DML. For the others, FireDAC emulates the operations performed by Array DML, which might produce only marginal performance improvements. Databases that benefit most from array DML include, but may not be limited to, InterBase, Firebird, IBM DB2, MS SQL Server, Oracle, and SQLite.

### **Support for a Variety of Query Execution Modes**

FireDAC provides support for a number of different query execution modes, both blocking and non-blocking. FireDAC can execute a single non-blocking query asynchronously, and provides a mechanism for canceling time-consuming blocking queries.

If you need to execute more than one FireDAC query asynchronously, you can execute those queries in worker threads (using Delphi's TThread class), providing each thread with its own FDConnection and query (or stored procedure) object. When used with a FireDAC-maintained pool of connections, connection overhead can be minimized.

### **Powerful Monitoring Capabilities**

FireDAC includes components and utilities that make it easy to trace FireDAC's interaction with the underlying databases. Using the FDMoniFlatFileClientLink, you can write trace information to a file, or you can implement your own

monitor mechanism using the FDMoniCustomClientLink, which triggers an event handler with informative parameters. You implement your custom tracing operations from within this event handler.

But the real gem in this collection is the FDMoniRemoteClientLink. Using this component, you can monitor trace information at runtime using the FireDAC

Monitor utility that ships with FireDAC. (The FDMonitor.exe utility is located in RAD Studio's bin directory.) You can monitor local applications, but since FDMoniRemoteClientLink and FDMonitor communicate via TCP/IP

(Transmission Control Protocol/Internet Protocol), you can monitor the application from almost anywhere on the Internet.

## Cached Updates

All TDataSets in FireDAC support cached updates. When enabled, changes made to records are cached in memory. That cache can then be used to review the changes, selectively revert specific changes, cancel all changes, or apply those changes still in cache to the underlying database.

FireDAC provides you with two modes for cached updates: the centralized mode and the decentralized mode. In the decentralized mode updates are cached individually for each dataset. When you are ready to apply the changes in the cache, you do so by calling the individual dataset's ApplyUpdates method.

The centralized mode provides a single change cache for one or more datasets.

In order to use the centralized mode, you use an FDSchemaAdapter component, which you then assign to the SchemaAdapter property of each FireDAC dataset that will participate in the shared change cache. As changes are made to

individual datasets, those changes are logged in a chronological order across all associated datasets in a single cache that is implemented by the FDSchemaAdapter. When you are ready to apply the changes in the cache, you do so by calling the FDSchemaAdapter's ApplyUpdates method, and the changes will be applied in the same chronological order as they were logged.

## Result Set Persistence

The data in FireDAC datasets and FDSchemaAdapters can be written to disk or to a stream to persist that data from one session to the next without reading from a database. This feature, which is nearly identical to the SaveToFile and SaveToStream methods of the ClientDataSet, gives you added flexibility when designing your applications.

In addition, if you are employing cached updates in your FireDAC datasets, the change cache is also persisted to the file or stream. This permits you to maintain

## Chapter 1: Overview of FireDAC 11

information about changes to your data over multiple sessions and an extended period of time.

What's interesting about LoadFromFile is that it allows you to work with data offline. For example, after loading data from a database into an FDQuery, you can save the query results to a local file, after which the data can be loaded again without the presence of the database connection. Here again, FireDAC goes beyond the traditional usage of dataset persistence. Combining offline mode, dataset persistence, and cached updates mode, your application can work disconnected from a database and persist the data between application run

sessions. Later, when the database is once again available, any changes to the data can be applied to the underlying tables.

## Data Type Mapping

Just as FireDAC gives you a lot of control over connection options, it also permits you to customize the mapping of data types. You control the mapping of data types using the FormatOptions property of your FireDAC object (FDConnection, FDQuery, or another FireDAC entity). If you set FormatOptions.OwnMapRules to True, you can define one or more FDMapRule

instances in the FormatOptions.MapRules property. Each FDMapRule describes to FireDAC a source data type, database data type name and/or a field name, and the data type to which it should convert the data in the destination.

Data type mapping is especially valuable in applications that support two or more different database types, permitting you to map incompatibilities to a common set of data types. It can also be invaluable when you are migrating from one database (for example, a file server database) to another (a remote

database, for instance), or from one set of data access components (for example, BDE) to FireDAC. Data type mapping in this case permits you to maintain a

consistent user interface despite changes in the underlying data types.

## **Local SQL**

Local SQL permits you to execute SQL SELECT, INSERT, and UPDATE statements against any dataset. (Other DML statements may be allowed, but the ones I listed are the obvious operations that most developers will be interested in.) For example, you can perform a query against an FDTable to gather simple aggregate statistics like SUM and AVG from the data it contains. Similarly, you can query an FDQuery and perform a left outer join to an FDStoredProc

component (in which case the stored procedure must return a result set).

Similarly, you can load a text file into an FDMemTable and execute an SQL SELECT query against it.

## **12 Delphi in Depth: FireDAC**

Importantly, this ability to query datasets is not limited to FireDAC datasets. For instance, you can create a query that performs a join between an FDQuery, an SQLDataSet, and a ClientDataSet.

FireDAC performs this SQL slight-of-hand by using the TDataSet API and the SQL engine from the SQLite open source project. It's a very clever technique, and one that enables a whole range of interesting data-related solutions that would otherwise be difficult or impossible to implement. Your joins don't have to be as complicated as I've described, but the benefits are obvious.

## **Additional Features**

I've outlined many of the major features of FireDAC, but there's more to FireDAC than what I've described so far. Here are some additional features of FireDAC that might interest you.

## **CONNECTION RECOVERY**

FireDAC provides transparent connection recovery. When an FDConnection's ResourceOptions.AutoReconnect property is set to True, FireDAC will attempt to re-connect to a database when the connection is dropped. This feature is especially valuable in environments where the network connection is unstable or is not reliable.

## **ADVANCED TRANSACTION SUPPORT**

FireDAC supports a range of transaction options, including simultaneous and nested transactions. True simultaneous transactions are supported on InterBase and Firebird. Nested transactions are emulated on those databases that support save points.

## **BUILT-IN DIALOG SUPPORT**

One of the lesser, though certainly welcome, features of FireDAC is its predefined dialog boxes. FireDAC includes a handful of useful, specific purpose dialog boxes that can be added to your application simply by dropping a

component onto your form or data module. For example, the FDGUIxLoginDialog can be used to automatically display a dialog box to capture a user's database username and password. Similarly, the FDGUIxErrorDialog can be used to display exceptions raised by FireDAC.

## **SUPPORT FOR DATABASE-SPECIFIC SERVICES**

FireDAC makes it easy to use some of the supported database services, such as backup, restore, validation, SQLite custom functions, and more. These services are accessed using the components from the FireDAC Services tab of the Tool Palette. Here you will find components that expose the underlying services for many of FireDAC's supported databases.

Chapter 1: Overview of FireDAC 13

## **CUSTOMIZABLE DATA ACCESS**

FireDAC gives you tremendous control over how your data is accessed. FDManager, FDConnection, and FDCommand components have a collection of

option-related properties (such as FetchOptions, ResourceOptions, and so forth) that each has many customizable properties. While the default options are

normally well suited for most data access scenarios, you have exceptional control over how FireDAC operates, permitting you to accommodate special cases. Furthermore, these options can be inherited, making it easy to customize these options on an application-by-application basis. For example, you can define the base options for an FDManager, and these will be inherited by any FDConnections that appear within your application.

Having said that, the down-stream classes, for example, FDQuery and FDTable, can override the settings inherited from higher order components, such as

FDConnections and the FDManager. As a result, you have complete control over the configuration of your data access components.

## **BATCH MOVE SUPPORT**

FireDAC provides you with classes for moving data from one database to another, as well as to and from text files and datasets. The TFDBatchMove component makes use of readers and writers to perform this feat.

Readers include the TFDBatchMoveTextReader,

TFDBatchMoveDataSetReader, and TFDBatchMoveSQLReader. Writers include the TFDBatchMoveTextWriter, TFDBatchMoveDataSetWriter, and TFDBatchMoveSQLWriter. You use the appropriate reader and writer instances to move data between text files, DataSets, and database tables (SQL).

## **WRITTEN IN OBJECT PASCAL**

FireDAC is written in Object Pascal. The Professional version of Delphi includes some of the FireDAC source code, and the Enterprise, Ultimate, and Architect editions include nearly all of the source code. This source code can help you understand how FireDAC works, and may even inspire you to create your own FireDAC database drivers.

Now that you've learned what FireDAC can do for you, I am certain that you will want to begin using it, and the following chapter is designed to help you do just that. There you will learn the basics of creating a connection and configuring it, executing a query through the created connection, and a few extra steps necessary to get everything compiled and running. Using these steps, you are ready to create new applications using FireDAC.

Chapter 2: Connecting to Data 15

# Chapter 2

## Connecting to Data

You connect to a database using a connection component (FDConnection), and then typically wire one or more FireDAC datasets to that connection. At this high level, these steps are no different than from any other data access framework based on the TDataSet interface. At a lower level, however, the specific steps you take are uniquely FireDAC.

FireDAC supports three distinct options for connecting to your database. These are:

- Temporary connections
- Persistent connections
- Private connections

Which type of connection you use affects how the connection can be used. For example, a persistent connection, which is a named connection, relies on an externally defined connection definition file, and therefore, can be shared by two or more applications. In addition, a persistent connection can be pooled, given that the connections are associated with a common FDManager, which is to say that connections can be pooled within a single application.

Private connections are also named connections, and can be pooled. Unlike persistent connections, however, they are defined at runtime, exist only for the duration of the application session, and cannot be shared between applications.

On the other hand, they do not rely on an external connection definition file, and they can be pooled.

Finally, temporary connections, which are not named, cannot be pooled and cannot be shared. They are, however, very easy to define at either runtime or design time, and do not rely on an external connection definition file.

### Creating Unnamed Connections

Unnamed connections are easy to define in that they do not require you to configure an FDManager (FDManagers are discussed later in this chapter). In

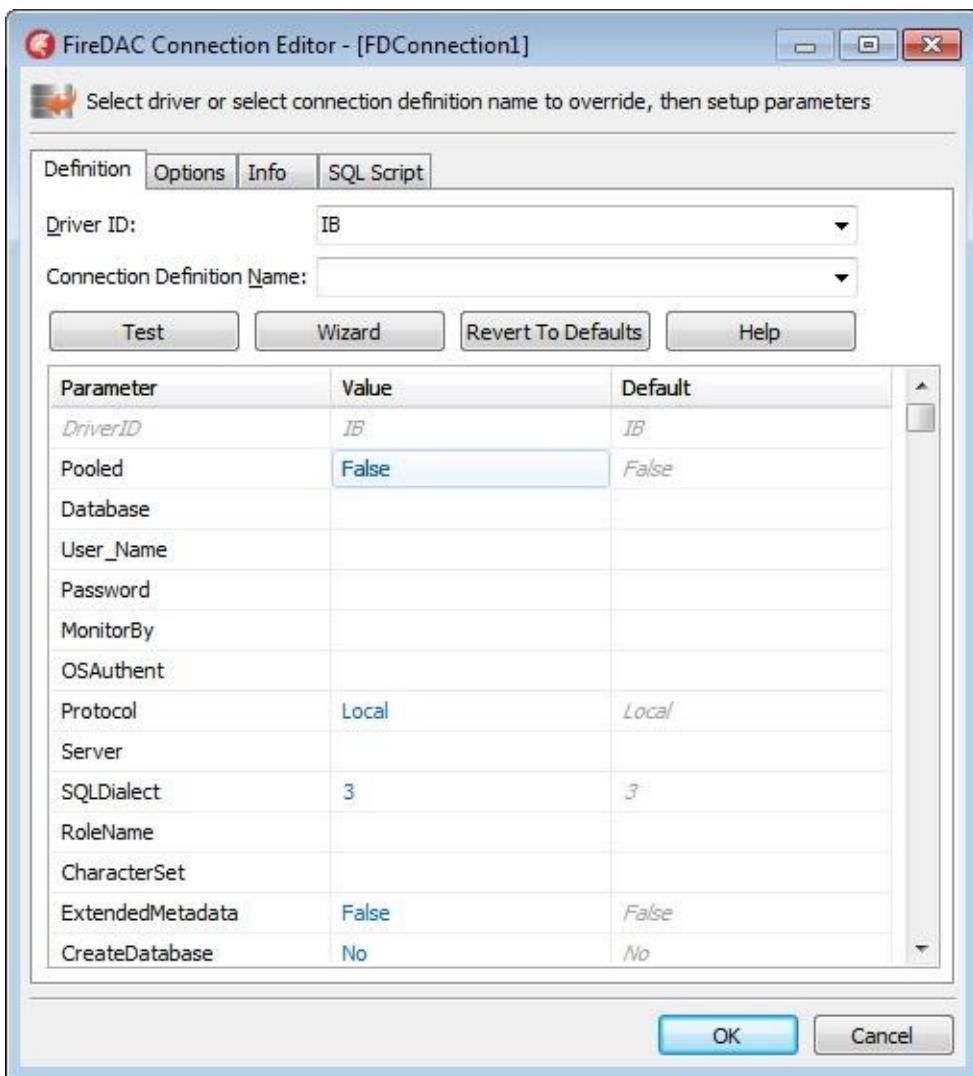
addition, they are easy to deploy given that they do not require an external file definition. The following sections demonstrate how to create unnamed connections, referred to as *temporary connections*.

## **Creating Temporary Connections**

You create a temporary connection using the FDConnection component. The following steps demonstrate how to create a temporary connection to InterBase (a database that was probably installed for you when you installed Delphi. If not, you can download the Developer Edition of InterBase for free).

With InterBase installed and running as a service locally, use the following steps to connect to the sample employee.gdb database that is part of Delphi's sample databases:

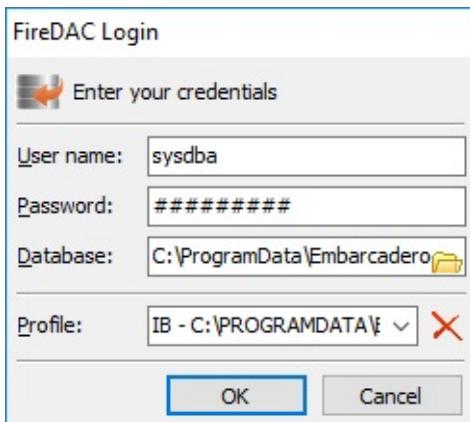
1. Select File | New | VCL Forms Application from Delphi's main menu to create a new project.
2. Add to this project a data module by selecting File | New | Other, and then selecting the Data Module template from the Delphi Files page of the Object Repository.
3. Add an FDConnection component from the FireDAC page of the Tool Palette to the data module.
4. Double-click the FDConnection component (or right-click it and select Connection Editor...) to view the FireDAC Connection Editor. This FireDAC Connection Editor, shown in the following Figure 2-1, has already had its Driver ID set to IB, which is why the connection parameters for the InterBase driver are shown.



## Chapter 2: Connecting to Data 17

**Figure 2-1: The FireDAC Connection Editor**

5. Set Driver ID to **IB**. Once you set the value of Driver ID, the properties associated with the FireDAC InterBase driver are displayed in the configuration pane, as seen in the preceding figure. Set Database to the employee.gdb database, which in the most recent versions of Delphi (Delphi 10.2 Tokyo at the time of this writing) is located in one of the following directories (with no line feeds):



18 Delphi in Depth: FireDAC

C:\ProgramData\Embarcadero\InterBase\gds\_db\examples\database\  
or

C:\Users\Public\Documents\Embarcadero\Studio\19.0\Samples\Data

Also, set User\_Name to **sysdba**, and Server to **127.0.0.1** (localhost works as well).

6. Next, click Test to test your connection. FireDAC will respond by displaying the Login dialog box shown in the following illustration. Enter the password **masterkey** and click OK.

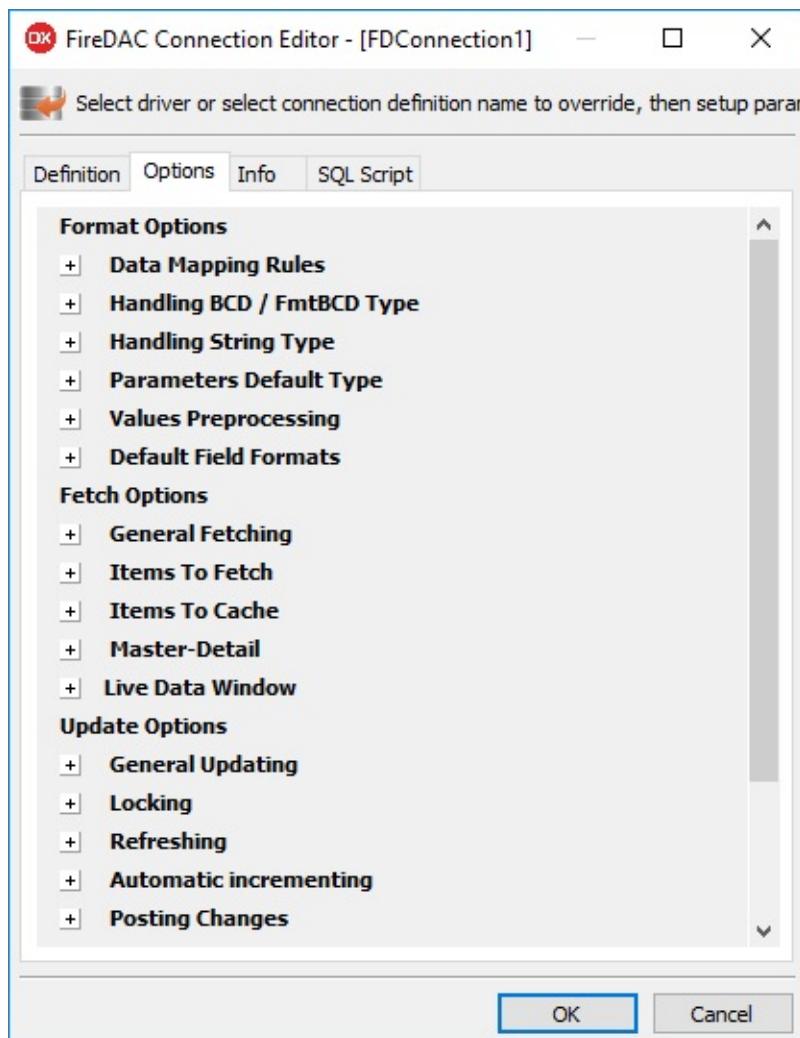
FireDAC should respond by confirming that it connected successfully.

*Note: If you are using the developer editor of InterBase, the service might not be running. In that case, you may have to start it using the Services applet*

from *Administrative Tools*.

This configuration was particularly easy since the InterBase server is running on the local machine. Depending on the database that you are connecting to, where it is located, and details about your configuration, you will likely enter more parameters than we did here.

In addition to connection parameters, the FireDAC Connection Editor permits you to configure many of the FDConnection properties. These are found on the



## Chapter 2: Connecting to Data 19

Options tab of the FireDAC Connection Editor, shown in the Figure 2-2.

### Figure 2-2: The Options tab of the FireDAC Connection Editor

Unlike the connection parameters, which depend on which FireDAC database driver you are using, the FDConnection properties shown on the Options tab are the same for all connections and control a wide variety of aspects of the connection.

#### 20 Delphi in Depth: FireDAC

Each of these options corresponds to an instance property of the FDConnection.

Those properties have names such as FetchOptions, FormatOptions, ResourceOptions, and the like. Instead of configuring these properties using the Connection Editor dialog box, you can select the FDConnection component and set the corresponding options using the Object Inspector. Using the Connection Editor to configure your options is much easier than using the Object Inspector, however, since the Connection Editor provides additional assistance in assigning values to these properties.

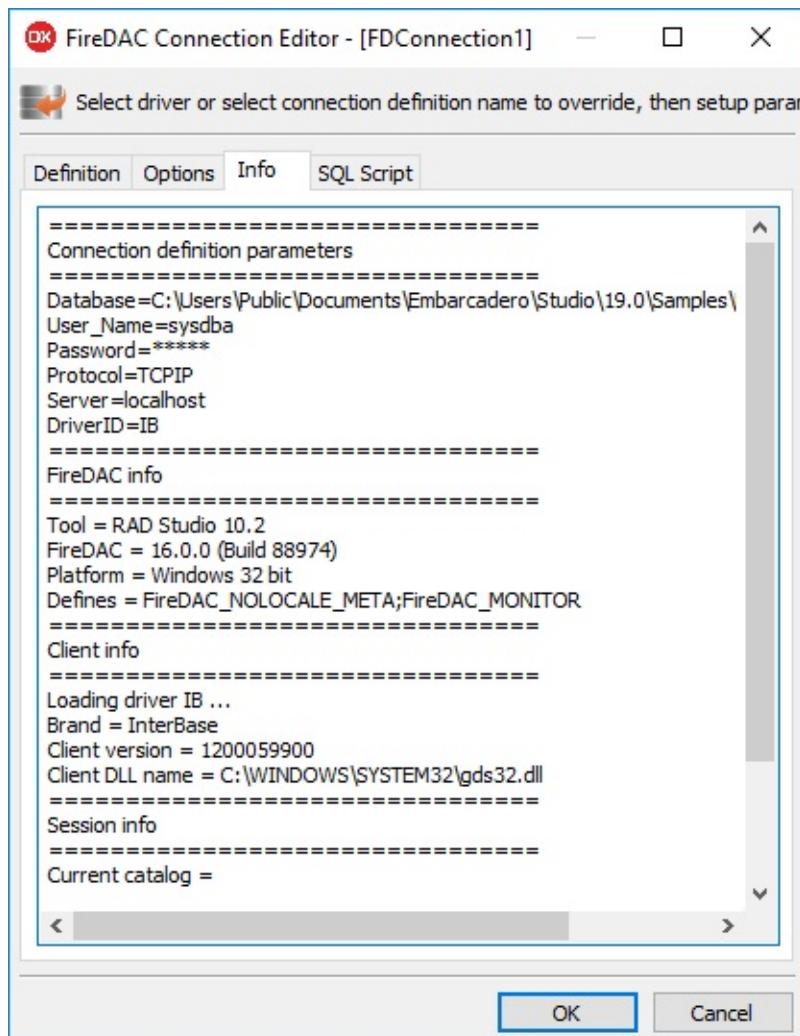
Use FormatOptions subproperties to define how the database data types are mapped to FireDAC fields. FetchOptions control how FireDAC retrieves data from the underlying database. The UpdateOptions property defines how FireDAC writes data back to the database, while ResourceOptions control how FireDAC allocates its resources. Finally, use TxOptions to configure how transactions are handled.

These properties are very important for your use of FireDAC. For that reason, these are discussed in greater detail in *Chapter 3, Configuring FireDAC*.

If you have successfully tested your connection, the Info page contains details about the connection you have established. This information is especially important if you are having problems with your connection. For example, suppose that get an error when you attempt to connect to your database, and the error message indicates that you are using the wrong client driver. Display the Info tab of the FireDAC Connection Editor and examine the Client info

section.

There you can see which client DLL FireDAC is trying to use, as shown in Figure 2-3.



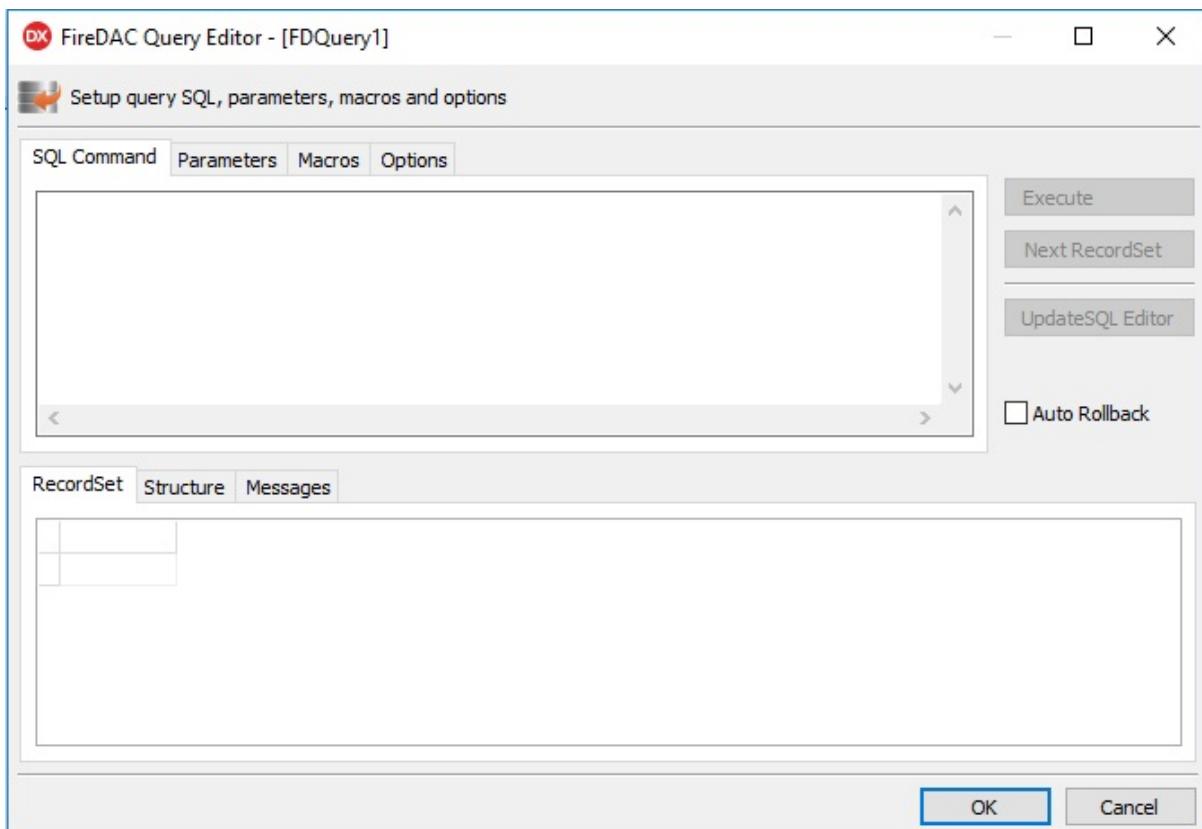
## Chapter 2: Connecting to Data 21

**Figure 2-3: The Info tab of the FireDAC Connection Editor**

Finally, the SQL Script tab permits you to enter SQL statements that you want to execute ad hoc against this connection. This can be very useful if you want to perform a quick operation such as viewing some data or creating a new table.

When you are done setting your connection parameters and options, close the Connection Editor by clicking the OK button.

We are now ready to finish this simple example.



### 22 Delphi in Depth: FireDAC

7. Add an FDQuery to your data module. (If you have an FDConnection component on the module to which you add an FDQuery, the query sets its Connection property automatically. If that does not happen, set the FDQuery's Connection property to FDConnection1 before continuing.)
8. Right click this FDQuery and select FireDAC Query Editor..., or simply double-click on the FDQuery. Delphi will respond by displaying the

Query Editor, shown in Figure 2-4.

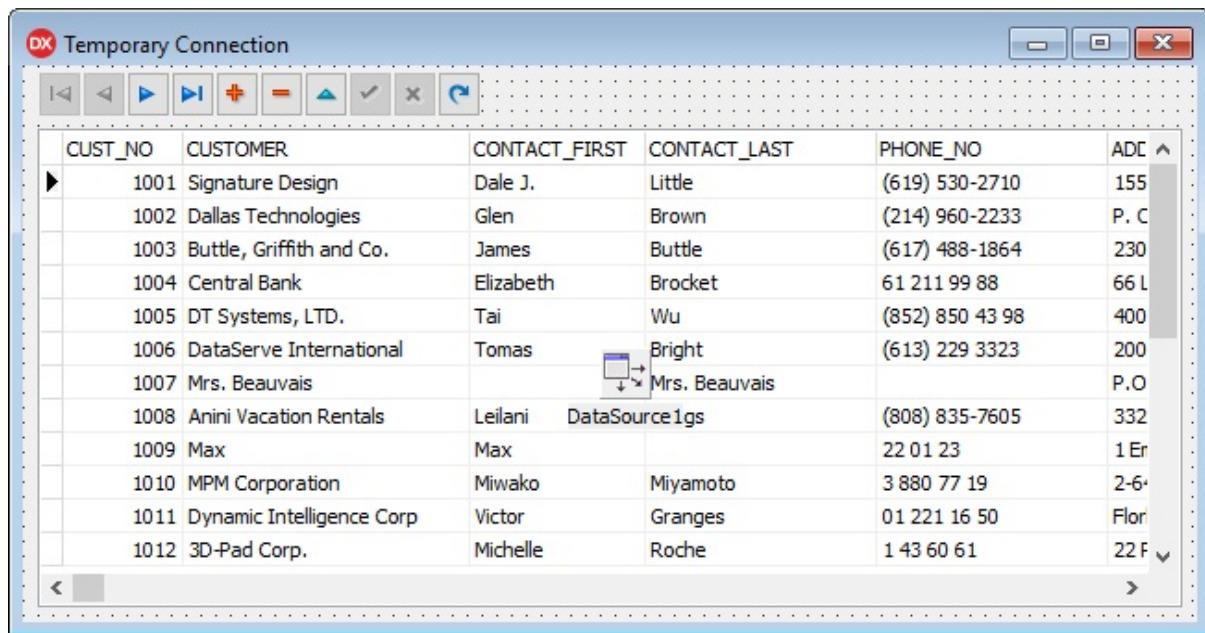
**Figure 2-4: The SQL Command tab of the FireDAC Query Editor**

9. In the SQL Command pane (the upper part) enter the following query:

SELECT \* FROM Customer

You can now test the query, if you like, by clicking the Execute button.

When you are done, click OK to return to the form. Clicking OK saves the query you entered into the FireDAC query's SQL property. If you instead click Cancel, the query text is lost.



The screenshot shows the FireDAC Query Editor window titled "Temporary Connection". The main area is a grid displaying data from a table named "Customer". The columns are labeled: CUST\_NO, CUSTOMER, CONTACT\_FIRST, CONTACT\_LAST, PHONE\_NO, and ADC. The data grid contains 12 rows of customer information. Row 7, which contains the name "Mrs. Beauvais", has a tooltip "Mrs. Beauvais" displayed over the "CONTACT\_LAST" cell. The "ADC" column for this row also has a tooltip "P.O". The "PHONE\_NO" column for this row has a tooltip "(613) 229 3323". The "CUST\_NO" column for this row has a tooltip "1007". The "CUSTOMER" column for this row has a tooltip "Mrs. Beauvais". The "CONTACT\_FIRST" column for this row has a tooltip "Tomas". The "CONTACT\_LAST" column for this row has a tooltip "Bright". The "PHONE\_NO" column for this row has a tooltip "(613) 229 3323". The "ADC" column for this row has a tooltip "200". The "CUST\_NO" column for this row has a tooltip "1007". The "CUSTOMER" column for this row has a tooltip "Mrs. Beauvais". The "CONTACT\_FIRST" column for this row has a tooltip "Leilani". The "CONTACT\_LAST" column for this row has a tooltip "DataSource1gs". The "PHONE\_NO" column for this row has a tooltip "(808) 835-7605". The "ADC" column for this row has a tooltip "332". The "CUST\_NO" column for this row has a tooltip "1008". The "CUSTOMER" column for this row has a tooltip "Anini Vacation Rentals". The "CONTACT\_FIRST" column for this row has a tooltip "Max". The "CONTACT\_LAST" column for this row has a tooltip "Max". The "PHONE\_NO" column for this row has a tooltip "22 01 23". The "ADC" column for this row has a tooltip "1 Er". The "CUST\_NO" column for this row has a tooltip "1009". The "CUSTOMER" column for this row has a tooltip "Max". The "CONTACT\_FIRST" column for this row has a tooltip "Miwako". The "CONTACT\_LAST" column for this row has a tooltip "Miyamoto". The "PHONE\_NO" column for this row has a tooltip "3 880 77 19". The "ADC" column for this row has a tooltip "2-6". The "CUST\_NO" column for this row has a tooltip "1010". The "CUSTOMER" column for this row has a tooltip "MPM Corporation". The "CONTACT\_FIRST" column for this row has a tooltip "Victor". The "CONTACT\_LAST" column for this row has a tooltip "Granges". The "PHONE\_NO" column for this row has a tooltip "01 221 16 50". The "ADC" column for this row has a tooltip "Flor". The "CUST\_NO" column for this row has a tooltip "1011". The "CUSTOMER" column for this row has a tooltip "Dynamic Intelligence Corp". The "CONTACT\_FIRST" column for this row has a tooltip "Michelle". The "CONTACT\_LAST" column for this row has a tooltip "Roche". The "PHONE\_NO" column for this row has a tooltip "1 43 60 61". The "ADC" column for this row has a tooltip "22 F".

CUST_NO	CUSTOMER	CONTACT_FIRST	CONTACT_LAST	PHONE_NO	ADC
1001	Signature Design	Dale J.	Little	(619) 530-2710	155
1002	Dallas Technologies	Glen	Brown	(214) 960-2233	P. C
1003	Buttle, Griffith and Co.	James	Buttle	(617) 488-1864	230
1004	Central Bank	Elizabeth	Brocket	61 211 99 88	66 L
1005	DT Systems, LTD.	Tai	Wu	(852) 850 43 98	400
1006	DataServe International	Tomas	Bright	(613) 229 3323	200
1007	Mrs. Beauvais		Mrs. Beauvais		P.O
1008	Anini Vacation Rentals	Leilani	DataSource1gs	(808) 835-7605	332
1009	Max	Max		22 01 23	1 Er
1010	MPM Corporation	Miwako	Miyamoto	3 880 77 19	2-6
1011	Dynamic Intelligence Corp	Victor	Granges	01 221 16 50	Flor
1012	3D-Pad Corp.	Michelle	Roche	1 43 60 61	22 F

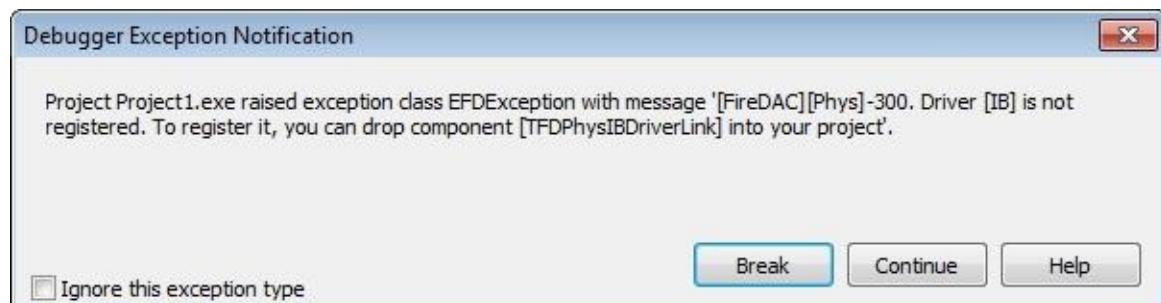
## Chapter 2: Connecting to Data 23

10. Now, set the FDQuery's Active property to True. At this point you will be once again challenged for the password associated with sysdba. Enter masterkey when prompted and click OK. We had to re-enter the password because we left the LoginPrompt property of the FDConnection to True so that the user will be challenged for a username and password when they first run the program. In order to supply that dialog box, add an FDGUIxLoginDialog component to the data module from the FireDAC UI tab of the Tool Palette. Now set the FDConnection's LoginDialog property to FDGUIxLoginDialog1.

11. Return to your main form and use your data module unit by selecting File | Use Unit, and select your data module's unit name from the displayed list.

12. Now add a DBGrid, a DBNavigator, and a DataSource component to your main form. Set the DataSource property of both the DBGrid and DBNavigator to DataSource1, and the DataSet property of the DataSource to DataModule1.FDQuery1 (or whatever name is appropriate for your data module and data source). Your form should now look something like that shown in Figure 2-5.

**Figure 2-5: The data returned by a FireDAC Query is shown in the designer**



### 24 Delphi in Depth: FireDAC

At this point everything looks great, and if we were using one of the other

data access frameworks (BDE, dbExpress, etc.) we could hit Run (F9) and the application would just run. But there is a little quirk with FireDAC, and that is that it requires some resources that are in units that do not necessarily appear in your uses clause by default. Depending on the version of Delphi you are running, if you hit Run (F9) now you might get a runtime error like the one shown in Figure 2-6.

**Figure 2-6: An exception notes that required resources are needed**

Later versions of FireDAC will ensure that the proper resources are automatically added, beginning with Delphi XE8. Nonetheless, if you get this error, there are two components that you can add to your project that will add the necessary units to your uses clause. The first component is the physical driver link component associated with the database driver you are using. For InterBase, this component has the name FDPhysIBDriverLink, and there are similarly named components for the other supported FireDAC drivers. The second component is the FDGUIxWaitCursor component.

You will find the physical driver link components on the FireDAC Links page of the Tool Palette, and the FDGUIxWaitCursor component on the FireDAC UI

page of the Tool Palette.

13. If necessary, add the two required components (FDPhysIBDriverLink and FDGUIxWaitCursor).

Here is the interesting part. FireDAC doesn't really need these components, and you do not need to set any of their properties either. What FireDAC really wants is the units associated with these components in your uses clause so that they get initialized. The FDPhysIBDriverLink component causes the insertion of the

FireDAC.Stan.Intf, FireDAC.Phys, FireDAC.Phys.IBBase, and FireDAC.Phys.IB units to your uses clause (at least in XE5 and later. A distinctly different set of units is added in versions prior to XE5). Similarly, the FDGUIxWaitCursor component results in the addition of the following units:

Temporary Connection

The screenshot shows a software interface titled "Temporary Connection". At the top is a toolbar with various icons: back, forward, search, add, delete, and others. Below the toolbar is a table with four columns: CUSTNO, COMPANY, ADDR1, and ADDR2. The table contains 15 rows of data. Row 1221 is highlighted with a blue selection bar. The data is as follows:

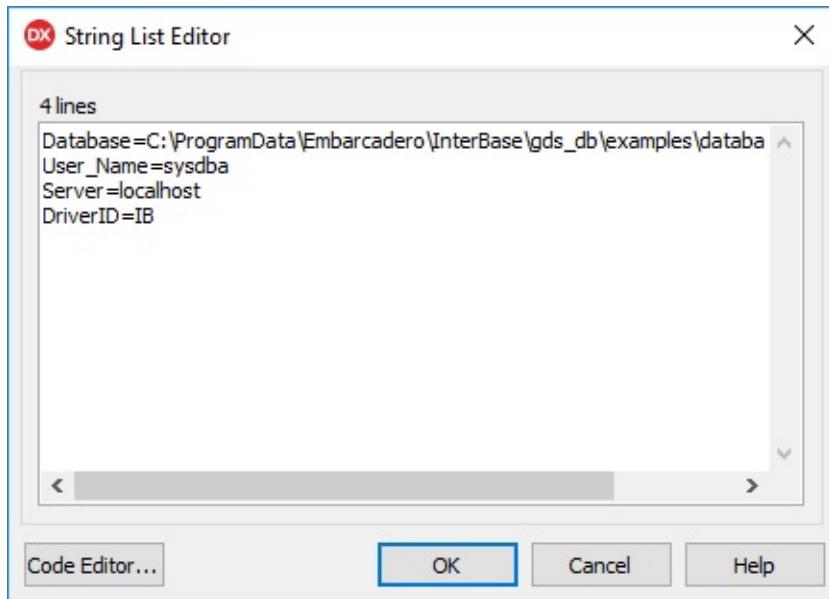
CUSTNO	COMPANY	ADDR1	ADDR2
1221	Kauai Dive Shoppe	4-976 Sugarloaf Hwy	Suite 103
1231	Unisco	PO Box Z-547	
1351	Sight Diver	1 Neptune Lane	
1354	Cayman Divers World Unlimited	PO Box 541	
1356	Tom Sawyer Diving Centre	632-1 Third Frydenhoj	
1380	Blue Jack Aqua Center	23-738 Paddington Lane	Suite 310
1384	VIP Divers Club	32 Main St.	
1510	Ocean Paradise	PO Box 8745	
1513	Fantastique Aquatica	Z32 999 #12A-77 A.A.	
1551	Marmot Divers Club	872 Queen St.	
1560	The Depth Charge	15243 Underwater Fwy.	
1563	Blue Sports	203 12th Ave. Box 746	

## Chapter 2: Connecting to Data 25

FireDAC.UI.Intf, FireDAC.VCLUI.Wait, and FireDAC.Comp.UI (again, in XE5 and later). If your project is a cross-platform project, the FireDAC.FMXUI.Wait unit will be inserted instead of FireDAC.VCLUI.Wait. With respect to these final two components that we've added to the project, since it is the underlying units that FireDAC really wants, you have two options. The first is to place these components on your form (or the data module, it really doesn't matter, so long as these units appear in at least one unit in your project source) and simply leave them there. The other is to place these components, save or compile your application to get the required units added to the uses clause, after which you can harmlessly delete the components. Personally, I prefer to leave them there as a reminder that they caused the addition of units to the uses clause.

14. Having added the necessary components, you can now press F9 and the application will run. Since we left the FDConnection.LoginPrompt property set to True, we are first challenged for our password (and would have been challenged for our user name as well had we not entered that into the FireDAC Connection Editor). After entering masterkey as the password, the main form is displayed, looking something like that shown in Figure 2-7.

**Figure 2-7: A DBGrid is displaying data from a FireDAC query at runtime**



## 26 Delphi in Depth: FireDAC

*Code: This code can be found in the FDTemporaryConnection project of the code download. See Appendix A for details.*

This data can now be viewed, navigated, and edited in a manner similar to that supported by the BDE's TTable component, even though this data is associated with an SQL query. In most of Delphi's data access mechanisms, queries are not directly editable, at least not by default. But in FireDAC, query results are editable, for the most part.

### Defining a Temporary Connection Using FDConnection.Params

By default, the configuration you define in the FireDAC Connection Editor is stored in the Params property of the TFDConnection. You can easily see this in the project you just created by using the following steps:

1. With your data module displayed in the designer, select the FDConnection component.
2. Select the Params property in the Object Inspector and click the ellipsis that appears in order to open the String List Editor, shown in Figure 2-8.

**Figure 2-8: The FireDAC Connection Editor saves the connection properties in the Params collection of the FDConnection component**

## Chapter 2: Connecting to Data 27

In this case, the String List Editor lists only four parameters: Database, User Name, Server, and DriverID. Remember, however, that this was a very simple configuration. We are connecting to a database server running on the local machine, and we did not leverage any additional capabilities of the

Connection Editor, such as pooling, OS authentication, role name, character set, or any of a number of available configurable parameters. In most cases, a successful

connection requires more parameters than did this application.

Nonetheless, the bottom line is that those parameters that you set using the Definition tab of the Connection Editor are written to the Params TStrings property, and those can be easily viewed by viewing the property editor of the Params property. Similarly, those properties that you set using the Options tab of the FireDAC Connection Editor are written to the corresponding instance properties of the FDConnection.

If you are familiar with the connection parameters of your database driver, you can enter these values into the Params property editor manually at design time to achieve the same effect as that accomplished using the Connection Editor.

### **Defining a Temporary Connection at Runtime**

If you want to define your database connection at runtime, you can also use the Params property. In fact, you can simply copy the same strings that you find in the Params property at design time following a successful connection to a

database, and use these values to execute the corresponding assignments at runtime.

For example, consider the values displayed in the String List Editor in the preceding figure. The following OnCreate event handler uses these values to configure the FDConnection at runtime

```
procedure TDataModule2.DataModuleCreate(Sender: TObject);  
begin  
  FDConnection1.Params.Add('Database=C:\ProgramData\Embarcadero' +  
    'InterBase\gds_db\examples\database\employee.gdb');  
  FDConnection1.Params.Add('User_Name=sysdba');  
  FDConnection1.Params.Add('Server=127.0.0.1');  
  FDConnection1.Params.Add('DriverID=IB');  
  FDQuery1.Open;  
end;
```

You can also set the Params property using the FDConnection's

ConnectionString property. In that case, you assign a semicolon-separated list of

## 28 Delphi in Depth: FireDAC

the name-value pairs that define your connection. For example, the following code achieves the same results as did the preceding code sample:

```
procedure TDataModule2.DataModuleCreate(Sender: TObject);  
begin  
  FDConnection1.ConnectionString := 'Database=' +  
    'C:\ProgramData\Embarcadero' +  
    'InterBase\gds_db\examples\database\employee.gdb;' +  
    'User_Name=sysdba;Server=127.0.0.1;DriverID=IB';  
  FDQuery1.Open;  
end;
```

Again, these connections are temporary connections, which are unnamed. As a result, they cannot be shared and cannot be pooled.

### **Creating Named Connections**

As the moniker suggests, named connections are connection definitions that have a name, or more specifically, a *connection definition name*. In the case of persistent connections, this name is defined in a file external to your application.

In the case of private connections, this name is associated with a connection definition defined at runtime, in which case no external file exists.

Named connections can be pooled. This is done by setting the Pooled parameter of the named connection definition to True. Pooled connections are particularly valuable in a multithreaded environment where each thread must create and

connect its own FDConnection (and use only FireDAC datasets associated with that connection). If these connections are configured using a named connection whose Pooled parameter is set to True, a newly connected FDConnection may

use an existing, connected connection from the FireDAC connection pool, instead of having to create an entirely new connection to the database. Since connecting to a database is a relatively expensive operation, connection pooling can provide a significant performance benefit in multithreaded

applications.

## Chapter 2: Connecting to Data 29

*Note: When working with a database from a worker thread, it is important that the thread connects to the database using an FDConnection distinct from any other thread. From the perspective of the database, each connection is a separate user, thereby allowing the multi-user capabilities of the database to resolve competition for data access by individual threads.*

The following sections demonstrate how to create named connection definitions.

Later in this section you will learn how to make public and private connections using these connection definitions.

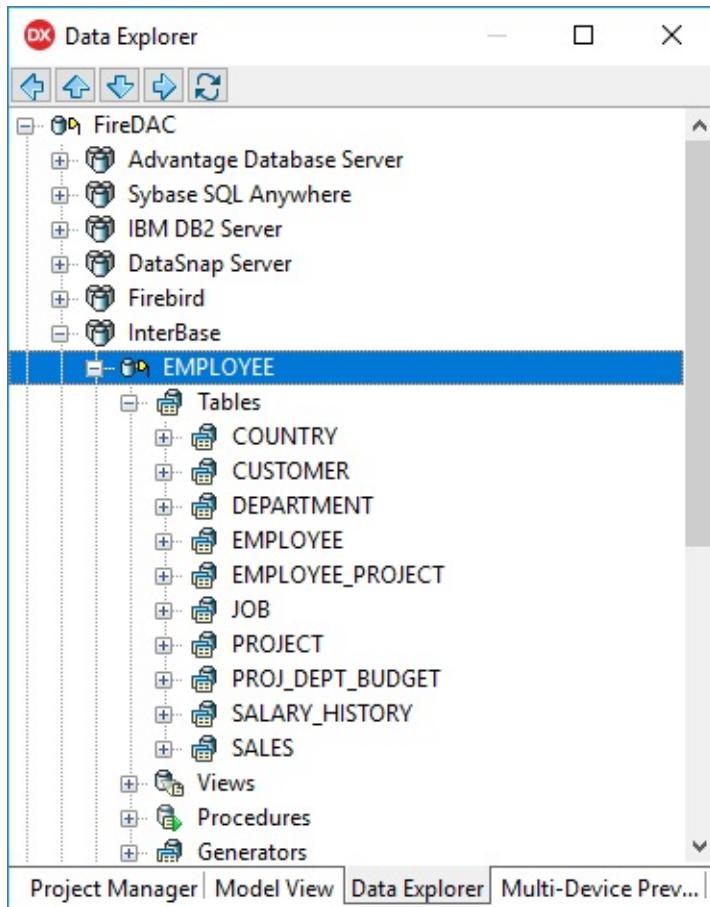
### **Creating Named Connections Definition Using the Database Explorer**

The Database Explorer in Delphi's IDE directly supports FireDAC beginning with Delphi XE6. Specifically, you can create, modify, and test FireDAC named connections directly from the Database Explorer. Existing named connection definitions appear as nodes beneath their associated driver node under the FireDAC section of the Database Explorer.

Figure 2-9 shows the named connection to the InterBase employee.gdb database defined in the FDConnectionDefs.ini file. As you can see, the node associated with this named connection shares the name of the connection in the connection definition file. The connection definition file is located, by default, in the following location:

C:\Users\Public\Documents\Embarcadero\Studio\

FireDAC\FDConnectionDefs.ini

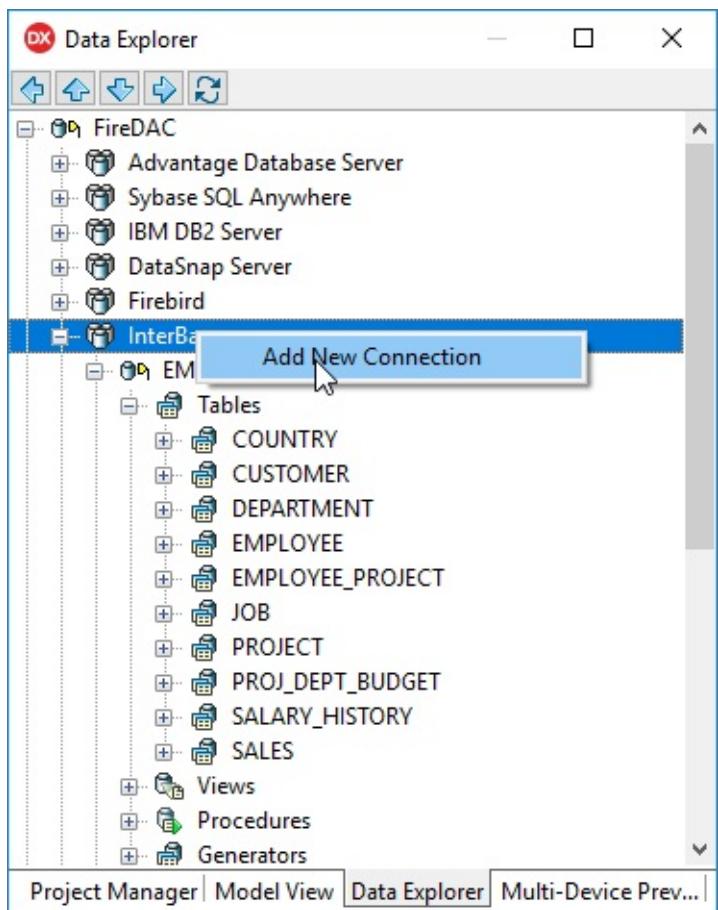


## 30 Delphi in Depth: FireDAC

**Figure 2-9: A connection named EMPLOYEE is selected in the Data Explorer in the Delphi IDE**

If you right click a node associated with a FireDAC named connection, you can choose to modify its definition, rename it, delete it, refresh it, or close it (if you have connected to it). Furthermore, once you connect to it, you can inspect its metadata, including tables, columns, indexes, views, and stored procedures, as shown in Figure 2-9.

To create a new named connection using the Database Explorer, right-click the node associated with the FireDAC database driver you want to use, and select Add New Connection, as shown in Figure 2-10.

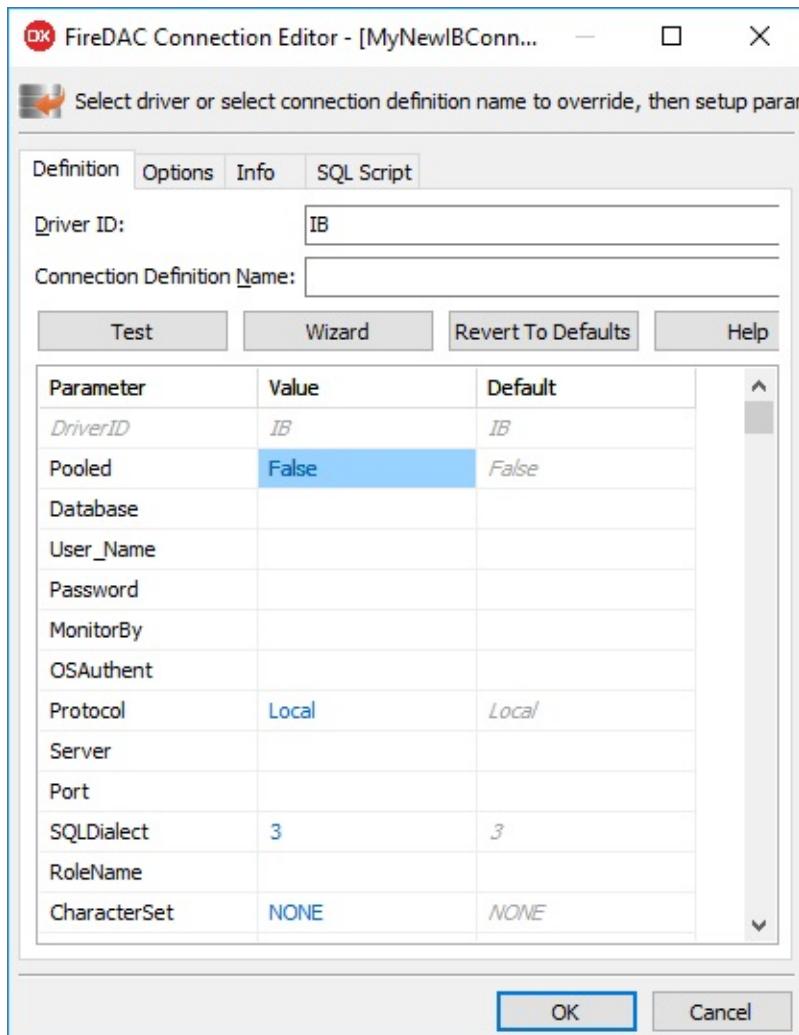


## Chapter 2: Connecting to Data 31

**Figure 2-10: Adding a new FireDAC named connection in the Data Explorer**

Next, you'll be asked to provide a name for this named connection, as shown here.

After entering a connection name, for example, MyNewIBConnection, and clicking OK, Delphi will display the FireDAC Connection Editor shown in



### 32 Delphi in Depth: FireDAC

Figure 2-11, where you define the parameters of this named connection. You complete the connection definition just as you did earlier in this chapter when you created the temporary connection using the FireDAC Connection Editor.

The primary difference is that once you save this configuration, it will be

associated with the connection definition name, and that name can be used to create a persistent connection.

### **Figure 2-11: Named connections are also configured using the FireDAC Connection Editor**

Chapter 2: Connecting to Data 33

Clicking OK saves these connection parameters into the FDConnectionDefs.ini file. Later, when you select to modify a named connection from the Data

Explorer, the Connection Definition Name field near the top of the FireDAC Connection Editor will contain the name of the connection you are editing, which in this case would be MyNewIBConnection.

### **Creating a Named Connection Definition Using the FireDAC Explorer**

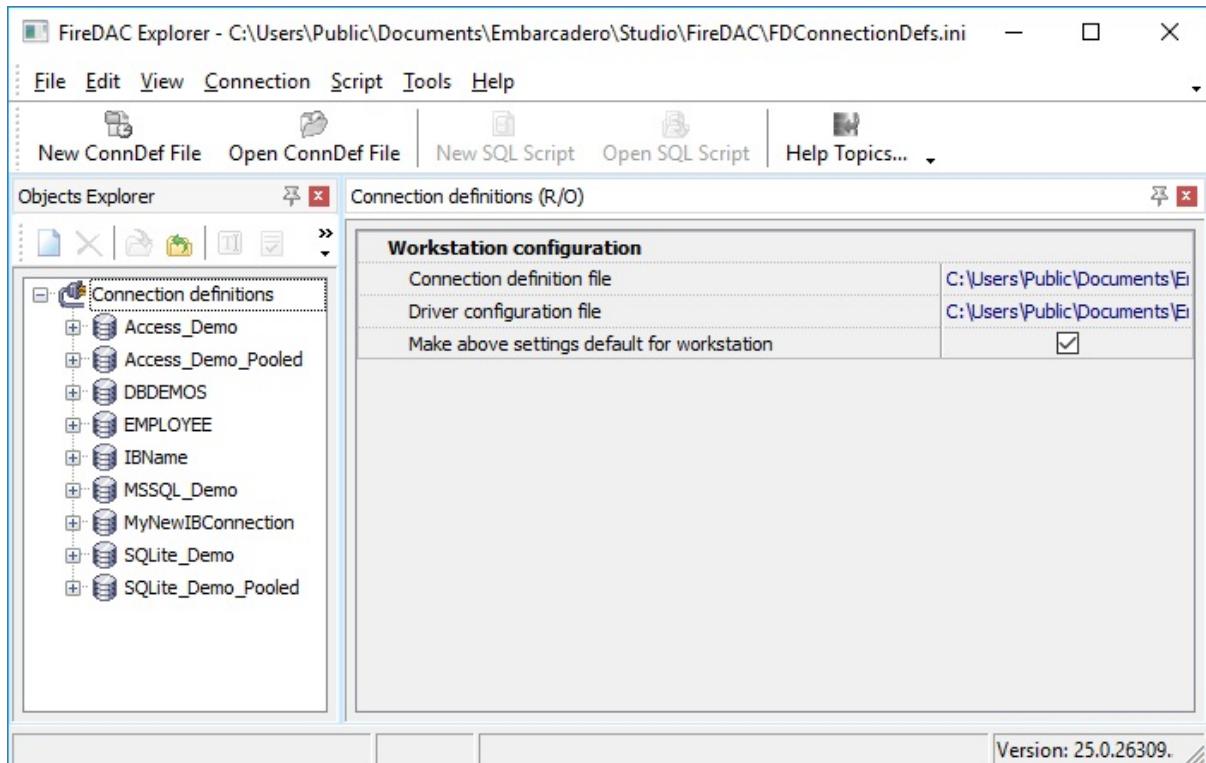
Before Delphi added support for creating and managing named connections in the Data Explorer, your go-to interface for defining named connections was the FireDAC Explorer. Since Delphi XE6, you can use either the Data Explorer or the FireDAC Explorer. In this section you will learn how to define a named connection definition using the FireDAC Explorer.

How you access the FireDAC Explorer depends on which version of Delphi you are using. In the most recent versions of Delphi, the FireDAC Explorer can be accessed from the Tools menu by selecting Tools | FireDAC Explorer.

The FireDAC Explorer is named FDEexplorer.exe, and it is located in Delphi's bin directory (which is C:\Program Files (x86)\Embarcadero\Studio\19.0\bin by default in Delphi 10.2 Tokyo). As a result, if you are using an older version of Delphi and the FireDAC Explorer is not available from Delphi's menu system, you can use the Tools | Configure Tools menu item to add a Tools menu item for the FireDAC Explorer, or simply navigate to Delphi's bin directory using the command prompt or the Windows Explorer and execute FDEexplorer.exe

manually.

The FireDAC Explorer is shown in the Figure 2-12.



## 34 Delphi in Depth: FireDAC

**Figure 2-12: You can use the FireDAC Explorer to create named connections**

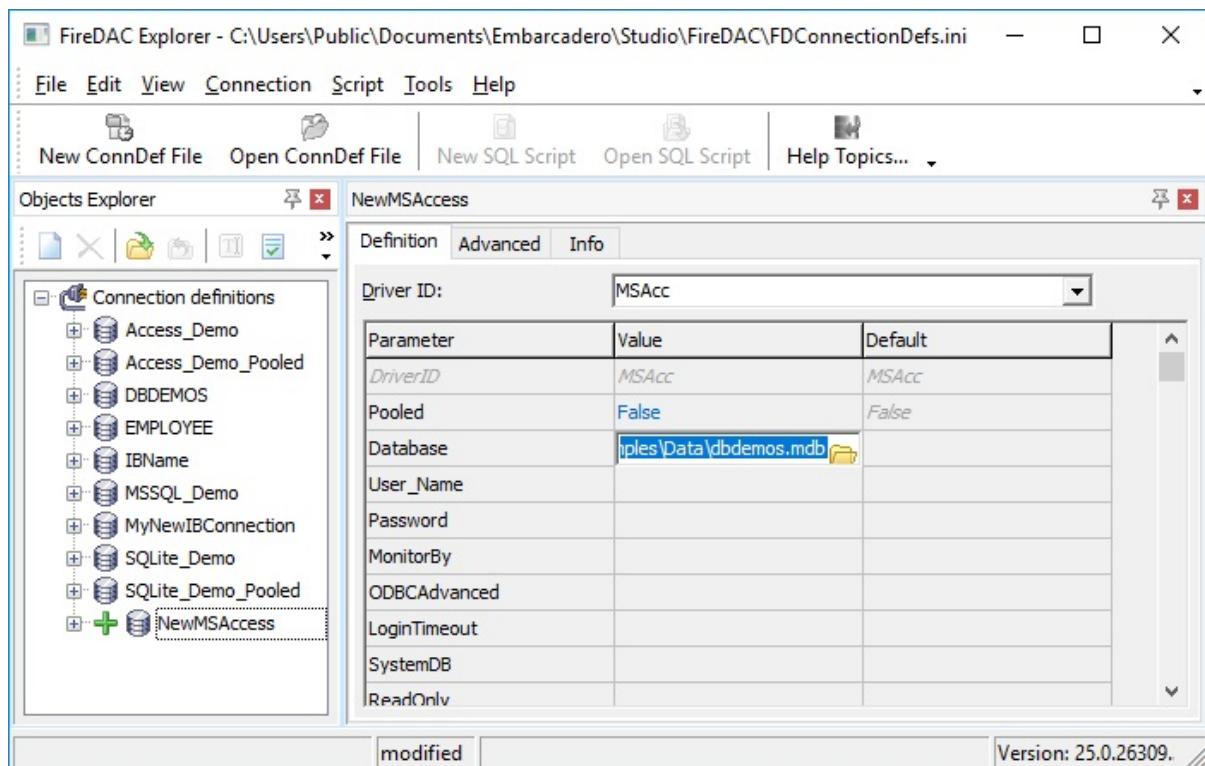
When the root node of the Objects Explorer is selected, the locations of the connection definition file (the same file that you update when you use the Data Explorer) and the driver configuration file are shown in the right-hand pane.

These files are simple INI files that hold the connection definition name/value pairs in sections whose names are associated with a named connection. These files are located in the following directory:

C:\Users\Public\Documents\Embarcadero\Studio\FireDAC

If you select one of the connection definition names, the saved parameters for that connection are shown in the right-hand pane. Furthermore, you can expand that connection definition to examine the underlying database's tables, stored procedures, and other similar features.

The following steps walk you through the process of creating a named connection for the dbdemos.mdb MS Access database that gets installed when you install Delphi:



## Chapter 2: Connecting to Data 35

1. Right-click the root node of the FireDAC Explorer and select Connection Definition. A new, named, but not yet configured definition will appear in the Objects Explorer tree. Select that name and rename it to NewMSAccess.
2. You now need to select the type of FireDAC driver you want to use for this database. Using the right-hand pane, set Driver ID to MSAcc. In response, the parameters for an Access connection will appear in the right-hand pane.
3. Set Database to the dbdemos.mdb database. This database will likely be in a directory similar to the following, where 19 represents the version of Delphi (and 19 is associated with Delphi 10.2 Tokyo):

C:\Users\Public\Documents\Embarcadero\Studio\  
19.0\Samples\Data

At this point, the FireDAC Explorer should look something like that shown in Figure 2-13.

**Figure 2-13: Creating a named connection in the FireDAC Explorer**

### 36 Delphi in Depth: FireDAC

4. You can now save this definition and open the connection. To begin with, right-click the connection named NewMSAccess and select Apply from the displayed context menu. Next, test opening the connection by clicking the plus sign (+) that appears to the left of the connection. You will be asked to supply a user name and password, but this database is not encrypted, so you can simply click the OK button without entering a username or password.

By saving the connection, you are creating a named connection, which is saved to the FDConnectionDefs.ini file. When you open the connection in the FireDAC Explorer you can use it to view your available tables, views, and stored procedures. You can also execute SQL statements against the

connection.

Since both the FireDAC Explorer and the Data Explorer save connection definition names in the same file, you can continue to work with your new connection using either of these tools. (Although you might have to close and then re-open Delphi before a connection that you have newly created in the FireDAC Explorer becomes available in the Data Explorer.)

### **Creating a Persistent Connection**

A persistent connection employs a named connection whose definition exists in an external ini file. This named connection may be created at either design time or at runtime.

When created at design time, the connection definition file must be in a location specified by a Windows registry setting, which is found in the following key:

HKEY\_CURRENT\_USER\Software\Embarcadero\FireDAC\ConnectionDefF

If there is no entry in the Windows registry, or your application is running on a different operating system, FireDAC will look in the current directory of the application for the connection definition file.

When created at runtime, you can specify the connection definition file and its location programmatically. In this case, you must use the FDManager, a singleton object used by all FireDAC connections. You can either use the default FDManager that FireDAC creates, or add one into your application at design time from the Tool Palette. If you do not create one at design time, FireDAC creates one at runtime, and you can refer to it at runtime using the FDManager global variable (this variable can also be used to refer to your design time-placed FDManager).

The main difference between using the automatically created FDManager and a manually placed FDManager is that you can configure properties of the

### **Chapter 2: Connecting to Data 37**

manually placed FDManager at design time. By comparison, any configuration of the automatically created FDManager must be performed at runtime, prior to activating any of the FDConnections in your application.

Regardless of whether you use the automatically created FDManager, or a manually placed one, the value of the ConnectionDefFileName property of the FDManager impacts the name and placement of your external connection definition file. If the FDManager.ConnectionDefFileName property is

assigned a value, FireDAC will use the connection definitions found in that file at runtime. If the ConnectionDefFileName property is blank at runtime, FireDAC

will attempt to load the file whose name appears in the Windows registry or in the application's directory.

Another relevant property of the FDManager is ConnectionDefFileAutoLoad, which is True by default. When set to True, the FDManager attempts to open the appropriate connection definition file before permitting any FDConnections to connect. If ConnectionDefFileAutoLoad is set to False, you must explicitly call FDManager.LoadConnectionDefFile prior to permitting any FDConnections to connect.

If you are defining a custom location or file name for the connection definition file, and you don't want to deploy your local connection definition ini file with your applications, you will want to create a named connection in the default (local) connection definition file, and then create a second connection definition file using the name and directory location you define in the

ConnectionDefFileName property. At design time, the default connection definition file will be used, and at runtime the specified connection definition file (the version you intend to deploy) will be used.

Creating a named connection definition was discussed earlier in this chapter, where steps were provided to create a named connection that we called NewMSAccess. That named connection is stored in the default connection definition file, and a portion of this file is shown here (the carriage return was added due to the format requirements of this book):

[FDConnectionDefs.ini]

Encoding=UTF8

REM Connections not related to this discussion appear here

[NewMSAccess]

DriverID=MSAcc

38 Delphi in Depth: FireDAC

Database=C:\Users\Public\Documents\Embarcadero\Studio\18.0\Samples\Data\dbdemos.mdb

Each named connection appears as a section name in the ini file, and the name/value pairs in this section correspond to the same name/value pairs that would otherwise appear in the FDConnection.Params property following the definition of a temporary connection.

If you want to use a connection definition file using a name other than FDConnectionDefs.ini, or in a location other than the application's current directory, you simply copy the section or sections of the default ini file and paste them into your custom ini file. In addition, you either set the ConnectionDefFileName property of a manually placed FDManager component,

or you set the ConnectionDefFileName property of the automatically created FDManager at runtime, prior to attempting to connect any of your FDConnection components.

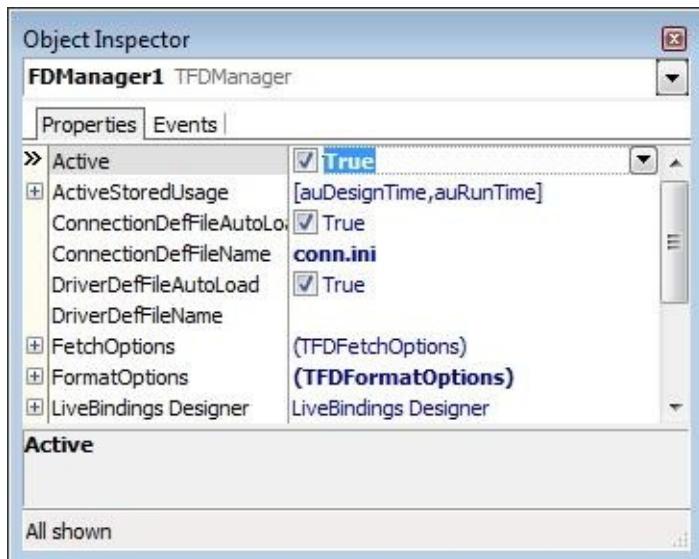
Assuming that we have copied the NewMSAccess section from the default connection definition file, and pasted it into a file named conn.ini in the application's working directory, the following runtime code will configure the automatically created FDManager prior to connecting an FDConnection:

```
procedure TDataModule1.DataModuleCreate(Sender: TObject);  
begin  
  FDManager.ConnectionDefFileName :=  
    ExtractFilePath(ParamStr(0)) + 'conn.ini';  
  //This next line is not necessary if the ConnectionName  
  //property of FDConnection1 was set at design time.  
  FDConnection1.ConnectionName := 'NewMSAccess';  
  FDConnection1.Open;  
end;
```

*Note: We could have set the FDConnection's ConnectionName property at design time as well, since this is a published property of the FDConnection class.*

Alternatively, if you have manually placed an FDManager component on your form, and set its ConnectionDefFileName property to conn.ini (this is a relative path, defaulting to the application's executable directory), your

application will automatically use the named connection in the connection definition file at runtime. The following figure shows the Object Inspector where a manually



## Chapter 2: Connecting to Data 39

placed FDManager has been selected. In this case, the ConnectionDefFileName property is set to conn.ini.

Of course, if you want to use a fully-qualified filename, you can do that as well.

In some cases, you might want to determine the correct path to the connection definition file at runtime. In those cases, you must have code that defines the value of the ConnectionDefFileName property of the FDManager prior to connecting the first FDConnection.

*Code: This code can be found in the FDPersistentConnection project of the code download.*

### Creating a Private Connection

Private connections possess all of the features of a persistent connection, except one. The connection definition is not defined in a connection definition file.

Instead, the connection information must be defined in code at runtime, and added to the FDManager. As you might expect, this operation must be completed prior to permitting any of your FDConnection objects to attempt to connect using the named connection.

The following code demonstrates one example of defining a private connection:

**procedure** TDataModule1.DataModuleCreate(Sender: TObject);

40 Delphi in Depth: FireDAC

**var**

Params: TStringList;

**begin**

Params := TStringList.Create;

**try**

Params.Add('Database=' +

'C:\Users\Public\Documents\Embarcadero' +

'Studio\19.0\Samples\Data\EMPLOYEE.GDB');

Params.Add('User\_Name=sysdba');

```

Params.Add('Server=127.0.0.1');
Params.Add('Pooled=True');
Params.Add('DriverID=IB');
FDManager.AddConnectionDef('IB_EMPLOYEE', 'IB', Params);
finally
Params.Free;
end;
FDConnection1.ConnectionDefName := 'IB_EMPLOYEE';
FDConnection1.Connected := True;
end;

```

This code connects to the same employee.gdb database as was demonstrated in the earlier part of this chapter during the discussion of temporary connections.

Note that the Params TStrings object is defined the same as was done in the earlier code sample that defined the temporary connection programmatically, meaning that you can use a temporary connection definition to determine which name/value pairs you need to define for the named connection that you add to the FDManager.

*Code: This code can be found in the FDPrivateConnection project of the code download.*

The name of this named connection is assigned to the ConnectionDefName property of the FDConnection prior to opening that FDConnection. As you can see, the name is defined within the FDManager, and the individual FDConnections obtain their connection strings simply by referring to this name in their ConnectionName properties. Since all connections refer to the same set of connection parameters, by name, they therefore share the same exact

connection parameters. It is through this mechanism that you can request, create, and sustain a pool of like connections, so long as the connection definition includes the Pooled parameter set to True.

Chapter 2: Connecting to Data 41

## Connecting to Any Database Using ODBC

FireDAC includes drivers for many of the most popular databases. But what if

your database is not one of them? Fortunately, FireDAC includes an ODBC driver, which you can use to connect to any database for which there is an ODBC driver. For example, if you plan on migrating a Paradox application from using the Borland Database Engine (BDE) to FireDAC, you will want to use the ODBC driver for Paradox tables that ships with Windows.

*Note: While FireDAC's drivers are dynamically linked into your compiled code, the use of an ODBC driver assumes that the ODBC driver has been deployed to the machines on which you will run your applications. In the case of the Paradox (and dBase) ODBC drivers, those ship with Windows by default. If the ODBC driver that you intend to use is not part of the default Windows*

*installation, you must take steps to ensure that the ODBC driver is properly installed either before you install your application, or as part of installing your application.*

*Also, note that nearly all ODBC drivers assume the installation of the local client library, where necessary. For example, if you are connecting to SQL Anywhere using an ODBC driver, you need to have both the SQL Anywhere ODBC driver as well as the SQL Anywhere client library installed on the machine through which you intend to connect to the database. On the other hand, most of the Microsoft ODBC drivers for file server databases, such as Paradox, do not require a client library, and can therefore be connected directly to those supported files.*

Implementing FireDAC connectivity to your database using ODBC involves several steps. First, you configure your FDConnection to use the FireDAC bridging ODBC driver (this can be done using a temporary, private, or persistent connection). Next, you define the parameters of this driver either by providing a reference to an existing ODBC data source name (DSN), or you provide all of the necessary connection information (typically associated with an ODBC

connection string) to the ODBCAdvanced parameter of the FDConnection.

There are a number of requirements before you can connect to your database using ODBC. Here is what is required:

- ▀ An installed ODBC driver for that database
- 42 Delphi in Depth: FireDAC
- ▀ A FireDAC connection that uses the FireDAC ODBC bridging driver to

connect to the ODBC driver

- ▀ The connection string parameters necessary to configure your ODBC driver to connect to your database

The following section demonstrates this process by implementing ODBC access to the sample Paradox tables that ships with Delphi. I am choosing Paradox for several reasons. For one, Microsoft Windows ships with a Paradox ODBC

driver, so it is something that should be available on your development machine.

Second, Delphi ships with sample Paradox tables that you can use. Finally, Paradox table support is one of the reasons why there are so many Delphi developers still using the BDE. Since the BDE has been deprecated, I want to show you how easy it is to access Paradox tables from FireDAC. Here are some of the connection string parameters for the Windows Paradox ODBC driver,

taken from the following URL (which at the time of this writing is still a valid URL):

```
http://www.connectionstrings.com/microsoft-paradox-driver-odbc/  
Driver={Microsoft Paradox Driver (*.db)};  
DriverID=538;Fil=Paradox 5.X;  
DefaultDir=c:\pathToDb\;Dbq=c:\pathToDb\;  
CollatingSequence=ASCII;
```

But before we start, I want to point out that the application that I am creating uses the 32-bit Windows ODBC driver for Paradox, which means that my application will be a 32-bit Windows application. If you need to build a 64-bit application, you will also need a 64-bit ODBC driver for your database.

Use the following steps to create a connection and execute a query against a sample Paradox table that ships with Delphi:

1. Place an FDConnection and an FDQuery onto a data module.
2. Right-click the FDConnection and select Connection Editor.

Alternatively, double-click the FDConnection to display the Connection Editor.

3. Set the Driver ID drop down to ODBC. The FireDAC ODBC driver

parameters appear in the Connection Editor.

4. Use the available dropdown list to set the ODBC Driver parameter to {Microsoft Paradox Driver (\*.db )}. Note: You include the matching Chapter 2: Connecting to Data 43

curly braces, and there is exactly one space between the ‘\*.db’ and the ‘)’}. If you omit this one blank space, the driver will not work.

5. Set ODBC Advanced to the following string. This string assumes that you are using Delphi 10.2 Tokyo, in which the version is 19.0. If you are using a different version of Delphi, replace the strings in both the DefaultDir and Dbq parameters with the corresponding sample directory path. Note, this string cannot include carriage returns or line feeds, unlike the following string which is formatted for the pages of this book: DriverID=538;Fil=Paradox 5.X;

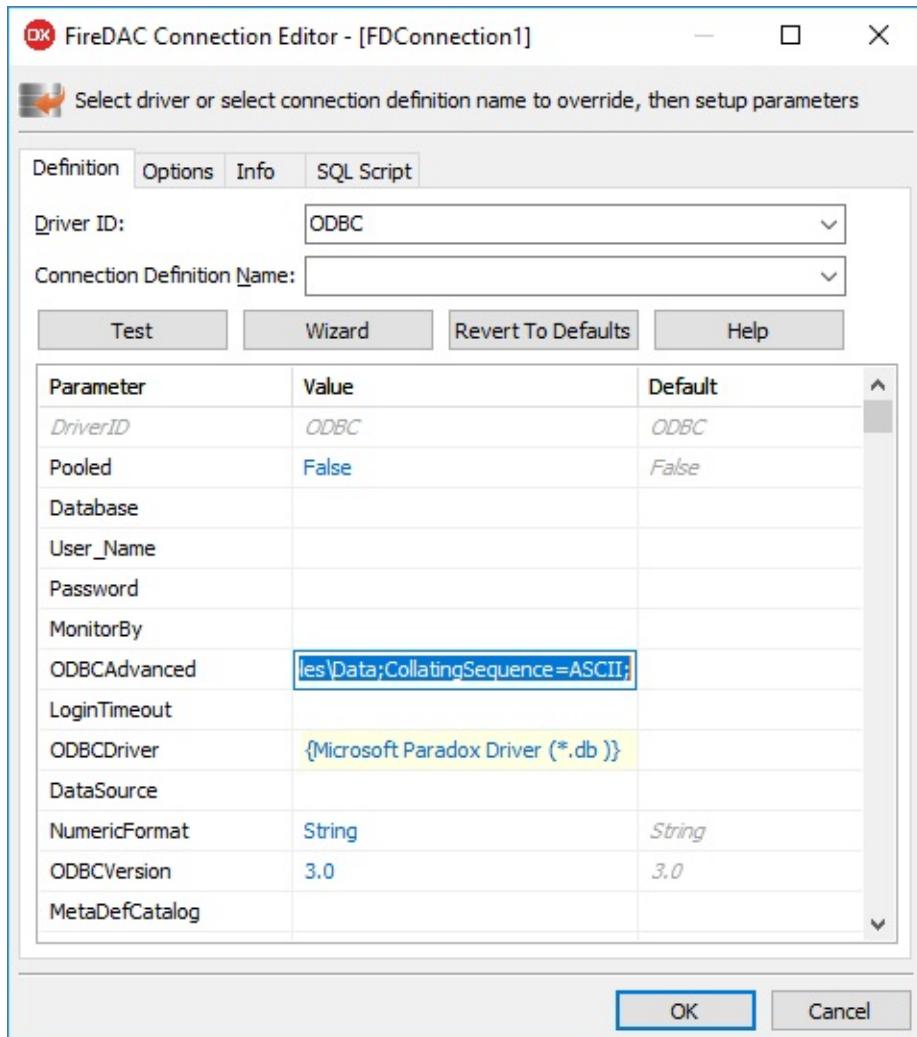
DefaultDir=C:\Users\Public\Documents\Embarcadero\Studio\19.0\Samples\Data;

Dbq=C:\Users\Public\Documents\Embarcadero\Studio\19.0\Samples\Data;

CollatingSequence=ASCII;

6. Your Connection Editor should now look something like that shown in Figure 2-14.

7. Click Test. FireDAC will ask for a username, password, and other information. The sample database that ships with RAD Studio is not encrypted, so you can skip entering any information and click OK. A dialog box should confirm that you are connected. Close this dialog box and then save the Connection Editor by clicking OK.



## 44 Delphi in Depth: FireDAC

**Figure 2-14: An ODBC connection is being configured in the Connection Editor**

8. Select the FDQuery and make sure that its Connection property is set to the FDConnection you just configured.
9. Set the FDQuery's SQL property to SELECT \* FROM Customer.
10. Next, set the FDQuery's Active property to True.
11. Finally, place one FDPhysODBCDriverLink and one FDGUIxWaitCursor onto your data module.

Paraodox ODBC with no Data Source Name

The screenshot shows a window titled "Paraodox ODBC with no Data Source Name". At the top is a toolbar with various icons for navigation and operations. Below the toolbar is a table with four columns: "CustNo", "Company", "Addr1", and "Addr2". The table contains 11 rows of data. Row 1 (highlighted with a blue border) is selected, showing CustNo 1221 and Company Kauai Dive Shoppe. Rows 2 through 11 show other companies like Unisco, Sight Diver, Cayman Divers World Unlimited, etc., with their respective addresses. The table has scroll bars on the right and bottom.

CustNo	Company	Addr1	Addr2
1221	Kauai Dive Shoppe	4-976 Sugarloaf Hwy	Suite 103
1231	Unisco	PO Box Z-547	
1351	Sight Diver	1 Neptune Lane	
1354	Cayman Divers World Unlimited	PO Box 541	
1356	Tom Sawyer Diving Centre	632-1 Third Frydenhoj	
1380	Blue Jack Aqua Center	23-738 Paddington Lane	Suite 310
1384	VIP Divers Club	32 Main St.	
1510	Ocean Paradise	PO Box 8745	
1513	Fantastique Aquatica	Z32 999 #12A-77 A.A.	
1551	Marmot Divers Club	872 Queen St.	
1560	The Depth Charge	15243 Underwater Fwy.	

## Chapter 2: Connecting to Data 45

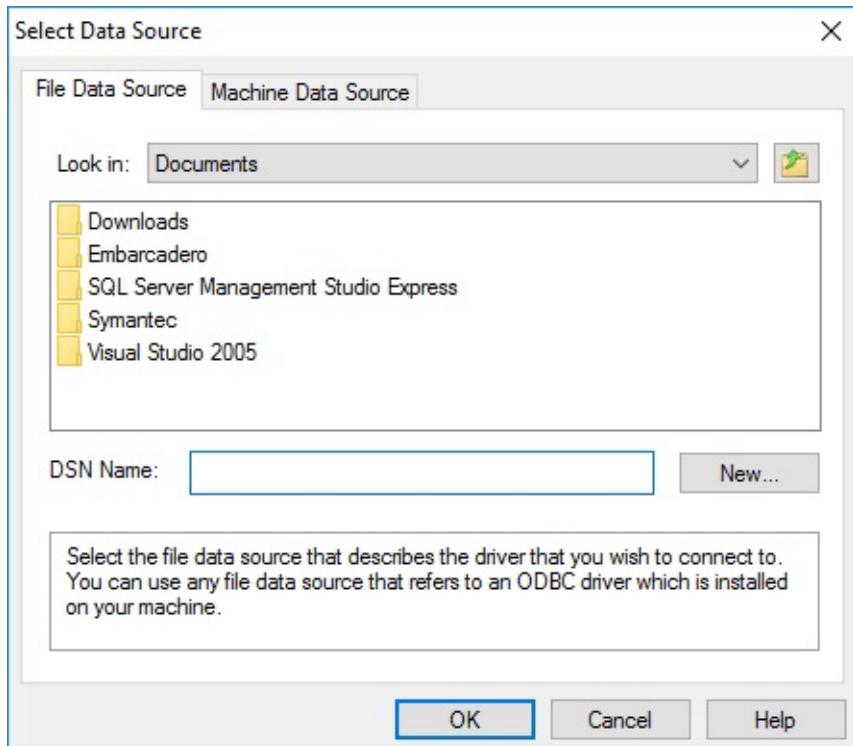
There, you are now connected to Paradox using FireDAC and the Windows Paradox ODBC driver. If you now use this data module from a form, and have a properly configured DBGrid, DBNavigator, and DataSource on this form, and

the DataSource's DataSet property is set to the FDQuery on the data module, your form should look something like that shown in Figure 2-15.

**Figure 2-15: A Delphi application uses a Paradox table through the FireDAC ODBC driver**

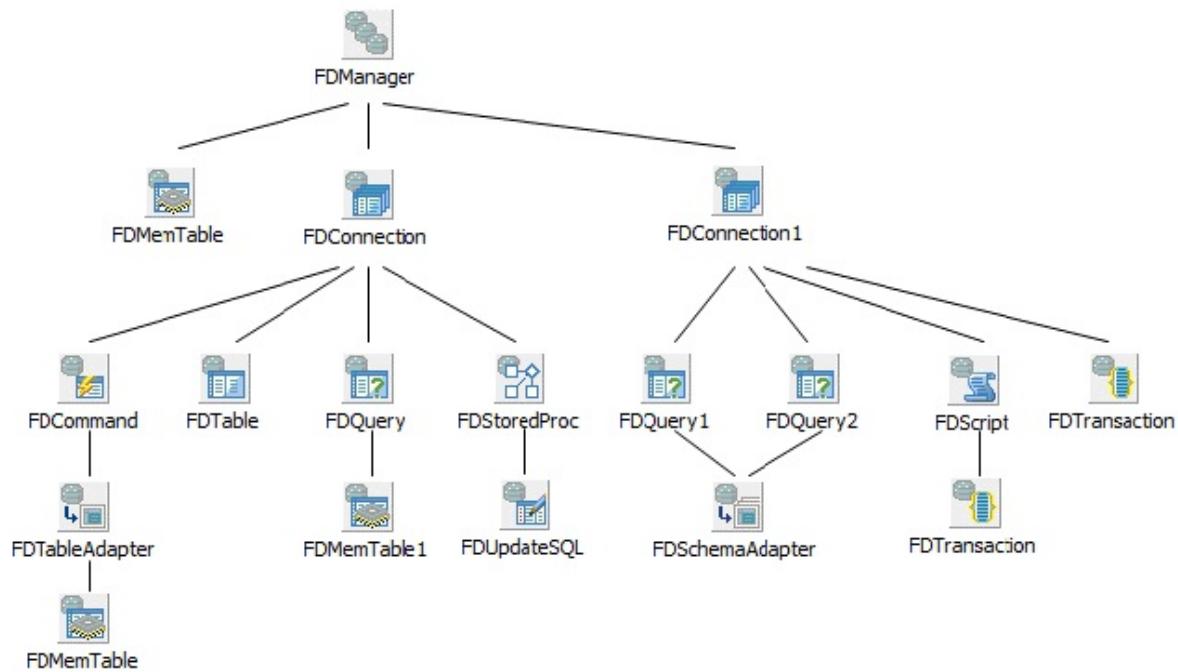
*Code: The FDParadoxODBCDynamic project can be found in the code download.*

There is another alternative. From the FireDAC Connection Editor, after you have set Driver ID to ODBC, you can click the Wizard button to display the ODBC Select Data Source wizard. The features of this wizard assist you in selecting or creating an ODBC data source. The ODBC Select Data Source wizard is shown in Figure 2-16.



**Figure 2-16: The ODBC Select Data Source wizard**

In the next chapter you will learn how to configure your FireDAC data access components.



# Chapter 3

## Configuring FireDAC

FireDAC consists of a collection of components and classes that implement a loosely-coupled, multi-layer architecture. You will find evidence of this structure in many places in FireDAC, and one of the more obvious is in the design of the connection and dataset classes that provide for the connection to, and interaction with, underlying databases. This basic hierarchy can be seen in Figure 3-1.

### Figure 3-1: The connection and dataset hierarchy

The FDManager is a singleton instance that exists in any FireDAC application, and it is either created at runtime by FireDAC, or you can place one in a project manually in order to configure it at design time. The actual connection to the database is created by FDConnection instances, and you can have one or more

48 Delphi in Depth: FireDAC

of these in any given application. All FDConnection objects within an application interact with the one FDManager.

FDCommand, FDTable, FDQuery, FDStoredProc, FDCommand, and FDScript

instances are used to work with the database to which its associated FDConnection is connected. FDMemTables can either work independently, or in conjunction with other datasets to work with data in memory. And FDTransaction instances are used to manage transactions against the database to which its associated FDConnection is connected.

### Share Configuration Properties

What's not obvious in Figure 3-1, but becomes apparent as soon as you start configuring these objects, is that they share many properties, and these properties are instance properties. Specifically, these properties hold a reference to an object that influences how the containing object performs its various tasks.

For example, nearly every component in Figure 3-1 has a published

FetchOptions property, which holds an instance of the TFDFetchOptions class.

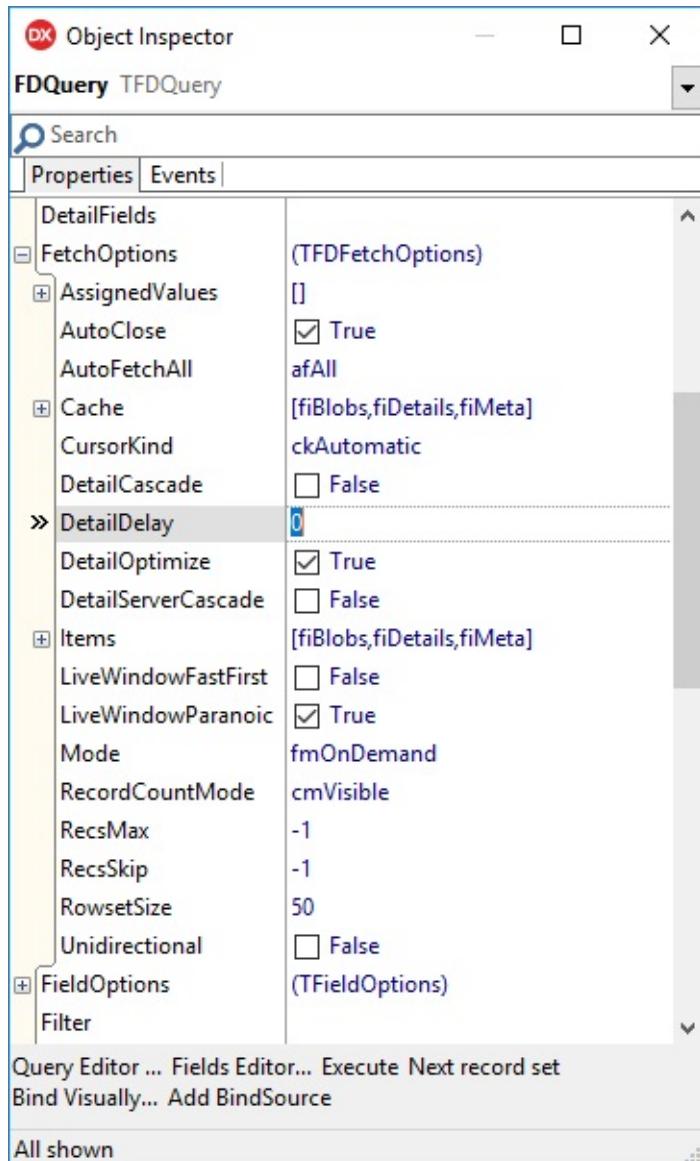
You can configure how the associated object retrieves data from the underlying database by setting the properties of this FDFetchOptions instance. Figure 3-2

shows the expanded FetchOptions property of an FDQuery in the Object Inspector.

### **Configuration Property Inheritance**

Yet another feature of these properties that is not immediately obvious is that the components in Figure 3-1 provide for the structured configuration of these properties. Specifically, when one of these components is configured to use another component higher up in the hierarchy, it will inherit the configuration of the corresponding instance property from the object to which it is connected.

For example, an FDQuery that is configured to use a particular FDConnection will initialize its FetchOptions property to match that of the FDConnection to which it is connected. Likewise, an FDConnection that is placed into a project that contains a configured FDManager will assume the FetchOptions settings of the FDManager.



# Chapter 3: Configuring FireDAC 49

## Figure 3-2: The FetchOptions property of an FDQuery in the Object Inspector

This cascading inheritance of corresponding instance properties can greatly reduce the amount of overall configuration that you need to perform. For example, if you want to change the UpdateMode of the UpdateOptions property from upWhereKeyOnly to upWhereAll on all of your datasets, all you need to do, before ever touching the UpdateOptions.UpdateMode for any of the

### 50 Delphi in Depth: FireDAC

FireDAC datasets, is to place an FDManager into your project, presumably on a data module, and set its UpdateOptions.UpdateMode to upWhereAll. Any FDConnections in the project will adopt this setting, as well any datasets assigned to these FDConnections.

*Note: There are very good reasons for wanting to set UpdateOptions.UpdateMode to upWhereAll, and I describe these reasons in detail later in this chapter.*

### Overriding Individual Configuration Properties

While FireDAC components inherit their configuration properties from the object to which they are connected, they can also override individual settings, providing you with complete control over the actions of the associated component. For example, an FDQuery connected to an FDConnection whose FetchOptions.Unidirectional property is False (the default) will inherit that value from the FDConnection. However, if you need this particular FDQuery to return a unidirectional cursor (a cursor type that uses much less memory than a bidirectional cursor), all you need to do is set the FDQuery's FetchOptions.Unidirectional property to True.

When you do this, the AssignedValues property of the FDQuery's FetchOptions property will include the evUnidirectional flag. This flag indicates to FireDAC

that you want this FDQuery to use its value of FetchOptions.Unidirectional, instead of the object to which it is configured. From that point on, even if you

change the `FetchOptions.Unidirectional` property of the `FDConnection` to which this `FDQuery` connects, your preference for the query's unidirectional cursor is respected.

## **Restoring Configuration Property Inheritance**

While you can override an inherited property from a connected object, you can also restore the inheritance, if you want. Take, for instance, the preceding example of an `FDQuery` overriding the inherited `FetchOptions.Unidirectional` property from its connected `FDConnection`. If you want to once again restore the inheritance between the `FDConnection`'s `FetchOptions.Unidirectional` property and that of the `FDQuery`, simply remove the `evUnidirectional` flag from the `FDQuery`'s `AssignedValues` property, and voilà, it's done. The `FDQuery` will once again use the value of the `FetchOptions.Unidirectional` property of the connection to which it is connected.

Chapter 3: Configuring FireDAC 51

## **Configuring the Shared Properties**

I've already mentioned that in order to benefit from property inheritance you must be aware of how `AssignedValues` affects this operation. For example, if you have previously overridden one or more values in an `FDQuery`, and now you want to assign it to an `FDConnection` from which you want to adopt all properties, you must ensure that all `AssignedValues` flags are removed from each of the shared property objects.

In the following sections, I am going provide you with a general overview of each of the shared property classes, along with a brief description of each of the properties of these instance classes. These properties permit you to configure almost every aspect of how FireDAC interacts with your database. In other

words, these properties give you a great deal of flexibility, and permit you to maximize your database's performance.

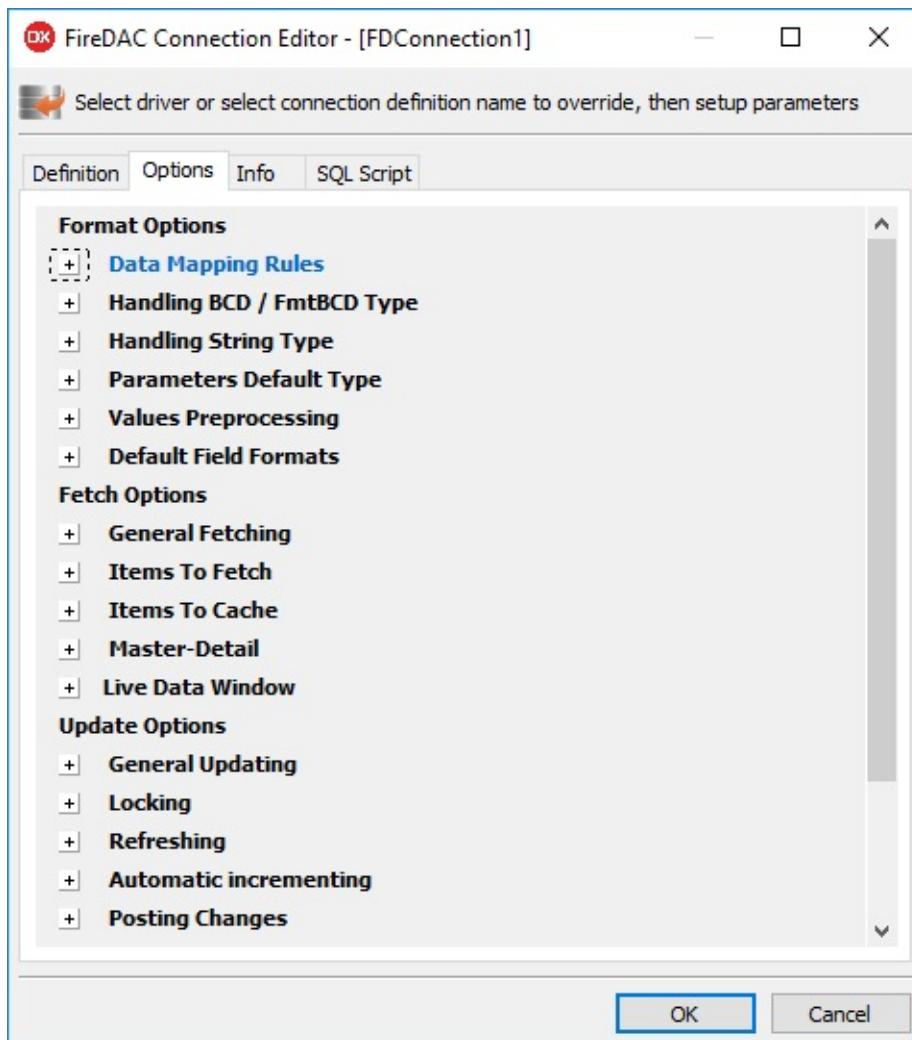
Doing this, however, requires that you know your database, and can figure out which of FireDAC's properties you use to customize your database's behavior. If you do decide to make changes, it is a very good idea to test your changes in a real world setting. Just as some changes might improve performance or add

desirable features, some changes can seriously hurt performance or introduce unwanted behaviors.

There are two ways to configure these shared properties. The obvious

mechanism is the Object Inspector. As shown back in Figure 3-2, these property categories can be expanded to display the individual properties of the implementing class, and those properties are displayed in alphabetical order.

The second way to configure these properties, at least with respect to the FDConnection class, is to use the Options tab of the Connection Editor, shown in Figure 3-3 (there is also an Options tab on the FireDAC Query Editor that serves this same purpose). As you can see, the sections on this tab are organized by the four value adopting classes, and then further organized by operation within that class.



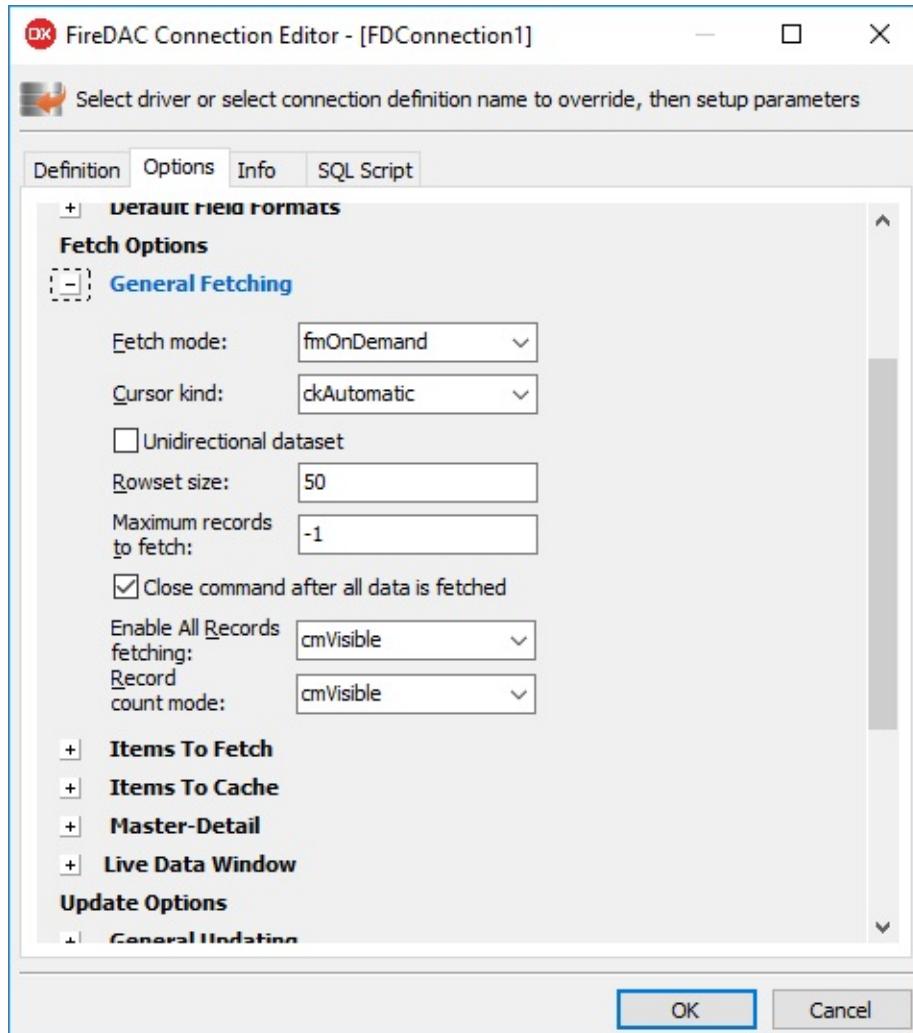
## 52 Delphi in Depth: FireDAC

**Figure 3-3: The Options tab of the Connection Editor dialog box**

Not only does this dialog box organize the various properties of these shared objects, but it provides further assistance in assigning values to the properties.

For example, in Figure 3-4 you can see that the FDFetchOptions.Mode and FDFetchOptions.CursorKind properties are set using a combobox, from

which  
you can select valid settings.



# Chapter 3: Configuring FireDAC 53

**Figure 3-4: The Connection Editor dialog box provides support for setting options**

The shared class properties are rich and complex, and it is outside the scope of this book to provide a complete description of each of them. Instead, these classes, how their properties are organized, and the individual properties, are provided here with brief descriptions, which are intended to provide you with a general orientation to the available options. The help system, as well as the RAD Studio Wiki, includes more extensive details concerning almost all of the

## 54 Delphi in Depth: FireDAC

properties listed here. As a result, if you are considering overriding the default value of a particular property, you should consider consulting the help in order to get a more complete description of that property.

Before I continue, I have a confession to make. I have not worked with every database that FireDAC supports. In fact, I have worked with only about a third of them. Furthermore, as I inspect the wealth of properties supported by these shared class properties I see some that I am not familiar with or that do not apply to the databases I have used. As a result, in writing this section I have had to, at times, rely heavily on Delphi's documentation for descriptions of these classes and their properties.

*Note: The AssignedValues property of the shared property classes, which identifies properties of a class that will not be adopted from another class, was described earlier in this section. As a result, this description will not be repeated in the following sections.*

### Fetch Options

As its name suggests, FetchOptions affects how FireDAC datasets retrieve information from a database. This property affects a number aspects of data retrieval, including how many records are returned in a result set, what type of cursor FireDAC creates on the database, and some of these properties only apply to particular components, such as an FDSchemaAdapter or an FDTable.

Figure 3-5 shows the expanded FetchOptions property for an FDConnection in the Object Inspector.

Object Inspector

FDConnection1 TFDConnection

Properties Events

Search

FetchOptions	(TFDFetchOptions)
AssignedValues	[]
AutoClose	<input checked="" type="checkbox"/> True
AutoFetchAll	afAll
Cache	[fiBlobs,fiDetails,fiMeta]
CursorKind	ckAutomatic
DetailCascade	<input type="checkbox"/> False
DetailDelay	0
DetailOptimize	<input checked="" type="checkbox"/> True
DetailServerCas	<input type="checkbox"/> False
Items	[fiBlobs,fiDetails,fiMeta]
LiveWindowFast	<input type="checkbox"/> False
LiveWindowPar	<input checked="" type="checkbox"/> True
Mode	fmOnDemand
RecordCountMax	cmVisible
RecsMax	-1
RecsSkip	-1
RowsetSize	50
Unidirectional	<input type="checkbox"/> False
FormatOptions	(TFDFormatOptions)

Connection Editor ...

All shown

# Chapter 3: Configuring FireDAC 55

## Figure 3-5: The FetchOptions property for an FDConnection in the Object Inspector

56 Delphi in Depth: FireDAC

### GENERAL FETCHING

These properties affect how records are retrieved from the underlying database.

#### Property

#### Description

AutoClose

When True, the dataset's cursor is closed after fetching records. Default is True. Set AutoClose to False when an SQL command produces several cursors.

AutoFetchAll

Defines which records are retrieved before a command is disconnected. Default is afAll.

CursorKind

Defines the type of database cursor that you want FireDAC to return. Possible values include ckAutomatic, ckDefault, ckDynamic, ckStatic, and ckForwardOnly. Default is ckAutomatic.

Mode

Controls how FireDAC will fetch records from the result set into internal storage. Possible values include fmManual, fmOnDemand, fmAll, and fmExactRecsMax. Default is fmOnDemand.

RecordCountMode Controls whether record count returns a count of all records, only fetched records, or only those still in

internal storage. Default is cmVisible.

#### RecsMax

Defines the maximum number of records to retrieve into memory. Default is -1 (no limit).

#### RecsSkip

Defines how many records to skip (not fetch) on the first request for result set records. Default is -1.

#### RowsetSize

Defines how many records are included in each fetch.

Default is 50.

#### Unidirectional

When True, returns a unidirectional cursor. The default is False.

Chapter 3: Configuring FireDAC 57

## ITEMS TO FETCH

This Items property controls what types of fields to include in a given fetch operation.

### Property Description

#### Items

Contains flags that specify whether to include blob fields, nested fields, and metadata in a fetch operation. The default is [fiBlobs, fiDetails, fiMeta].

## ITEMS TO CACHE

This Cache property controls what items you want FireDAC to cache.

### Property Description

#### Cache

Contains flags that specify whether to cache blob fields, nested fields, and meta data in internal storage. The default is [fiBlobs, fiDetails, fiMeta].

## MASTER-DETAIL

These properties affect how FireDAC handles master-detail records.

### **Property**

#### **Description**

DetailCascade

When using centralized cached updates, controls whether changes to the master record will be cascaded to the associated fields on the detail tables. Default is False.

DetailDelay

Defines how many milliseconds FireDAC will wait after a scroll on the master record before fetching details. Default is 0.

DetailOptimize

When set to True, detail records are refreshed only when the master record fields used to join to the detail records are changed. Default is True.

58 Delphi in Depth: FireDAC

DetailServerCascade Controls whether locally cascaded changes will be written to the underlying server upon update. Default is False.

## **LIVE DATA WINDOW**

Use these properties to configure how FDTables generate their underlying queries when Live Data Window is enabled (the default). Live Data Window, which applies to FDTables only, is described in greater detail in *Chapter 4, Basic Data Access*.

### **Property**

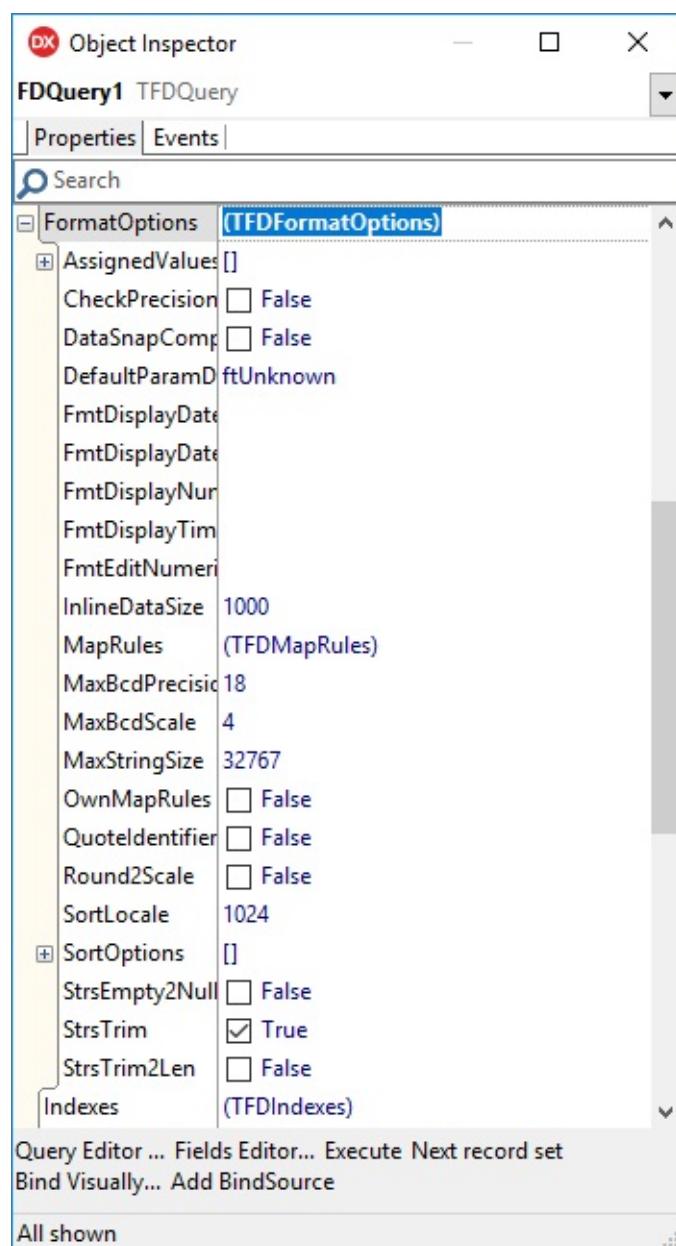
#### **Description**

LiveWindowFastFirst When set to False, fetch all record operations are optimized. Otherwise, the time to fetch the first record is optimized. The default is False.

LiveWindowParanoic When False, some query generation is optimized, but results may not be accurate (record counts, for example). Default is True in Delphi 10 Seattle and later, and False in earlier versions.

## Format Options

The TFDFormatOptions class is the base class for all FireDAC FormatOptions properties. The properties of this class define how the data obtained from your FireDAC datasets appears to your applications. Figure 3-6 shows the expanded FormatOptions property for an FDQuery in the Object Inspector.



# Chapter 3: Configuring FireDAC 59

## Figure 3-6: The FormatOptions property for an FDQuery in the Object Inspector

60 Delphi in Depth: FireDAC

### DATA MAPPING RULES

FireDAC's support for data mapping permits you to specify how a given data type from a database is managed by FireDAC. For example, a given field in a database may be stored as a floating-point value, but in your application you may want to treat it as a field of type ftCurrency.

Data mapping is especially useful in a variety of situations. For example, when migrating from one database vendor to another, where the new vendor does not support a data type that you have relied upon in your applications, you can map the associated type from the new vendor to the data type that your application is expecting.

Another valuable use for data mapping can be found in applications that must support two or more database vendors (say, InterBase and MS SQL Server). In those tables where incompatibilities exist between table structures, data mapping can be used to map those underlying fields to a common type that will be used in your applications.

Finally, data mapping is valuable when you are migrating an application from one data access mechanism, for example, the BDE, to FireDAC.

*Note: Data mapping is typically only required when you employ persistent fields. It is rarely used with dynamic fields.*

Basic data mapping is managed through two properties of the TFDFormatOptions class.

#### Property

#### Description

MapRules

MapRules is a collection of FDMapRule objects, each of which provide FireDAC with information about conversions that it must perform between source data types and destination data types. By default, this collection is empty.

Items cannot be added to the MapRules collection unless the OwnMapRules property is True.

**OwnMapRules** When True, the object to which this FDFormatOptions property belongs will employ mapping rules to provide data type compatibility. Otherwise mapping rules will be inherited. The default value is False.

## Chapter 3: Configuring FireDAC 61

Individual mapping rules are instances of the TFDMapRule class. The following table contains the names and descriptions of the published properties of this class.

### **Property**

#### **Name**

##### NameMask

A string that can include both literal characters, as well as the SQL wildcard characters % and \_, to define the column name(s) to which this mapping will apply. Useful when applying the same mask to columns that include common character combinations. The default is an empty string.

##### PrecMax

Defines the upper limit for a range of data type precision.

##### PrecMin

Defines the lower limit for a range of data type precision.

##### ScaleMax

Defines the upper limit for a range of data type scale.

##### ScaleMin

Defines the lower limit for a range of data type scale.

##### SizeMax

Defines the upper limit for a range of data type size.

##### SizeMin

Defines the lower limit for a range of data type size.

**SourceDataType** Specifies the data type returned by the associated database driver.

**TargetDataType** Specifies the data type FireDAC will expose.

#### TypeMask

A string that can include both literal characters, as well as the SQL wildcard characters of % and \_, to define the column DB type name(s) to which this mapping will apply. Useful when applying the same mask to fields of a specific data type. The default is an empty string.

62 Delphi in Depth: FireDAC

### **HANDLING BCD/FMTBCD TYPE AND DATASNAP COMPATIBILITY**

Use these properties to configure which numeric field type FireDAC will default to, given the underlying scale and precision, as well as to configure FireDAC to be more compatible with dbExpress and DataSnap.

#### **Property**

##### **Name**

**DataSnapCompatibility** Set this to True when FireDAC is used in DataSnap server applications to make FireDAC compatibility with the DataSnap framework. The default is False.

##### **MaxBcdPrecision**

When working with binary coded decimal values, FireDAC will define fields as dtBDC if precision is less than or equal to the column precision, otherwise it will define the field as dtFmtBDC. The default is 18.

##### **MaxBcdScale**

When working with binary coded decimal values, FireDAC will define fields as dtBDC if scale is less than or equal to the column scale, otherwise it will

define the field as dtFmtBDC. The default is 4.

## HANDLING STRING TYPE

Use these properties to configure how FireDAC retrieves string fields.

### Property

#### Name

InlineDataSize

String fields whose length is less than this value are stored directly in the record buffer. Memory is allocated for longer fields and a pointer to the memory is stored in the record buffer. The default value is 1000 bytes.

MaxStringSize

Defines the maximum length for strings. Longer fields are returned as BLOB (Binary Large OBject) fields. The default is 32767.

StrsEmpty2Null Set to True to have zero-length strings returned as nulls in the result set.

Chapter 3: Configuring FireDAC 63

StrsTrim

Set to True to remove trailing blanks from fixed length strings and trailing zero bytes from fixed length binary fields. The default is True.

StrsTrim2Len

Set to True to have FireDAC trim excess characters when a field value exceeds the configured field length. The default is False.

## PARAMETERS DEFAULT TYPE

Use this DefaultParamDataType property to define the default data type of parameters.

### Property

#### Description

**DefaultParamDataType** The default data type of query, stored procedure, and script parameters. The default is `ftUnknown`.

## VALUE PREPROCESSING

These properties control how FireDAC retrieves numeric and date/time values.

### Property

#### Description

**CheckPrecision** Compare the precision and scale of values prior to reading and writing operations when `True`. The default is `False`.

### Round2Scale

Set to `True` to have FireDAC round numeric and date/time values to the scale associated with the underlying database fields. The default is `False`.

64 Delphi in Depth: FireDAC

## DATASET SORTING

Use these properties to control default sorting options.

### Property

#### Description

**SortLocale** Define the locale ID to configure local sorting. The default is `LOCALE.USER.DEFAULT`.

**SortOptions** Use the sort options flags to define the default local sorting.

Available flags include `soNoCase`, `soNullFirstwise`, `soDescNullLast`, `soDescending`, `soUnique`, `soPrimary`, and `soNoSymbols`. The default is `[]`.

## QUOTATION IDENTIFIER

Use this `QuoteIdentifiers` property to control whether identifiers are enclosed in quotes in generated SQL statements.

### Property

#### Description

**QuoteIdentifiers** Set to `True` to enclose identifiers (for example, field names and table names) in quote characters. The default is `False`.

## Chapter 3: Configuring FireDAC 65

### DEFAULT FIELD FORMATS

These properties permit you to change the default format masks for native types.

#### Property

##### Name

FmtDisplayDate

Use FmtDisplayDate to override the default display format for Date fields. The default is an empty string.

FmtDisplayDateTime Use FmtDisplayDateTime to override the default display format for DateTime fields. The default is an empty string.

FmtDisplayNumeric Use FmtDisplayNumeric to override the default display format for numeric fields. The default is an empty string.

FmtDisplayTime

Use FmtDisplayTime to override the default display format for Time values. The default is an empty string.

FmtEditNumeric

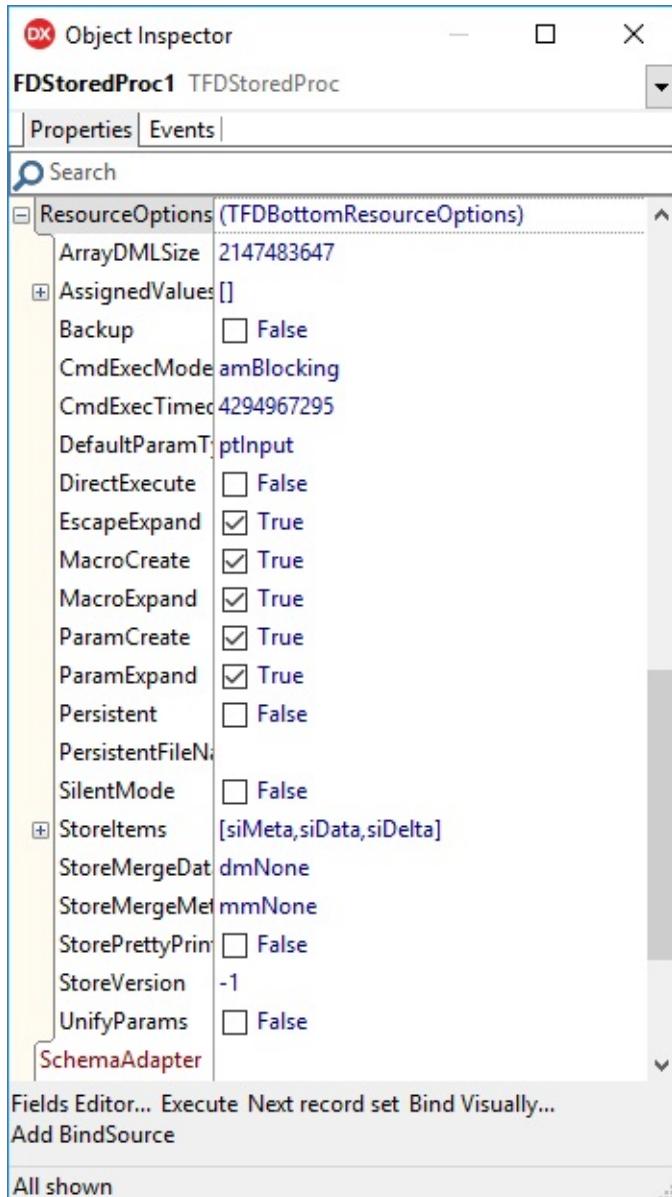
Use FmtEditNumeric to override the default editing format for numeric fields. The default is an empty string.

### Resource Options

ResourceOptions defines how FireDAC allocates and manages resources. The ResourceOptions property of FDManager and FDConnection is implemented by

a TFDTopResourceOptions instance, a TFDResourceOptions descendant. FDCommand and FireDAC's datasets implement ResourceOptions as a TFDBottomResourceOptions instance, which descends from

TFDTopResourceOptions. Figure 3-7 shows the expanded ResourceOptions property for an FDStoredProc in the Object Inspector.



## 66 Delphi in Depth: FireDAC

**Figure 3-7: The ResourceOptions property for an FDStoredProc in the Object Inspector**

Chapter 3: Configuring FireDAC 67

## COMMAND TEXT PROCESSING

These properties let you control the FireDAC command preprocessor.

### Property

#### Name

**DefaultParamType** Defines the default parameter direction. Valid values are ptUnknown, ptInput, ptOutput, ptInputOutput, and ptResult. The default is ptInput.

**DirectExecute**

When False, FireDAC automatically prepares queries prior to execution. Introduced mostly for MS SQL Server. The default is False.

**EscapeExpand**

When True, FireDAC will expand escape sequences during SQL preprocessing. The default is True.

**MacroCreate**

When True, FireDAC populates the macro collection automatically. The default is True.

**MacroExpand**

When True, the SQL preprocessor replaces macros with their associated values. The default is True.

**ParamCreate**

When True, FireDAC populates the Params collection automatically. The default is True.

**ParamExpand**

When True, FireDAC replaces FireDAC parameter markers with the DB native markers. The default is True.

**UnifyParams**

When True, special symbols are removed from stored procedure parameter names, and special parameters are removed. The default is False, and should be set to True only under special circumstances, such as when an application supports several databases.

## PERSISTENCE MODE

Use these properties to enable and control dataset persistence.

### Property

#### Name

Backup

When set to True, FireDAC will create a backup of the data file before overwriting it during a save operation.

The default is False.

BackupExt

The default extension to use for a backup file. The default is .bak.

BackupFolder

The default folder to which the backup file is saved.

The default is the current folder.

DefaultStoreExt

The default extension to use when closing a dataset and Persistent is True. The default is an empty string.

DefaultStoreFolder The default folder to use when closing a dataset and Persistent is True. The default is the current folder.

DefaultStoreFormat The format, XML, JSON, or binary, to use when storing a dataset. The default is sfAuto.

Persistent

When set to True, datasets will load and save their data to a file, rather than the underlying database. The default is False.

PersistentFileName The name of the file from which the data will be loaded or to which it will be saved. The default is an empty string.

StoreItems

A set of flags that define what will be stored. The

default value is [siData, siDelta, siMeta].

#### StoreMergeData

Controls whether data loading into an active dataset from a file or stream will be merged or overwritten.

The default is dmNone.

#### Chapter 3: Configuring FireDAC 69

#### StoreMetaData

Controls whether meta data loading into an active dataset from a file or stream will be merged or overwritten. The default is mmNone.

#### StorePrettyPrint

Data persisted in the XML or JSON format is formatted in a human-friendly format.

#### StoreVersion

Used to define the version number to use when writing data. The default is -1.

### **COMMAND EXECUTION**

These properties influence query and stored procedure execution.

#### **Property**

##### **Name**

#### ArrayDMLSize

The maximum number of records to update in a single slice of a batch operation using array DML. The default is 2,147,483,647.

#### CmdExecMode

The default mode for executing queries. Valid values include amBlocking, amNonBlocking, amCancelDialog, and amAsync. The default is amBlocking.

CmdExecTimeout The default period, in milliseconds, after which

FireDAC will abort executing a query. The default value is 2,147,483,647.

#### SilentMode

Set to False to suppress the display of the wait cursor during query execution, and any other FireDAC dialog, like the Login Dialog. The default value is False.

### CONNECTION RESOURCES

Use these properties to configure how FireDAC manages connection resources.

#### Property

##### Name

##### AutoConnect

Set to True to cause a connection to automatically connect when a dataset needs a connection. The default is True.

##### 70 Delphi in Depth: FireDAC

##### AutoReconnect

Set to True to request that an FDConnection attempt to reconnect following the loss of a connection. The default is False.

KeepConnection When True, connections continue to be connected even when there are no open datasets using the connection. The default is True.

##### MaxCursors

Use to define the maximum number of concurrently open cursors. If great than 0, and the maximum cursors are active, and one more cursor is needed, the least used cursor will be closed. The default is -1 (no limit).

##### ServerOutput

Set to True to ensure that FireDAC receives all server-

generated messages, even when they do not represent an error condition. These messages may be found in the FDConnection.Messages property. The default is False.

ServerOutputSize Use to define the server buffer size. Applies only to Oracle. The default is 20000.

## **Update Options**

Update options permit you to configure various aspects of FireDAC's ability to edit the data and write data back to an underlying database. The UpdateOptions property of both FDManager and FDConnection is a TFDUpdateOptions

instance, while the UpdateOptions properties of FDCommand and other FireDAC datasets is a TFDBottomUpdateOptions instance, a descendant of TFDUpdateOptions. Figure 3-8 shows the expanded UpdateOptions property for an FDCommand in the Object Inspector.

Object Inspector

FDCommand1 TFDCommand

Properties | Events |

Search

	(TFDBottomUpdateOptions)
AssignedValues	[]
AutoCommitUpdates	<input type="checkbox"/> False
AutoIncFields	
CheckReadOnly	<input checked="" type="checkbox"/> True
CheckRequired	<input checked="" type="checkbox"/> True
CheckUpdatable	<input checked="" type="checkbox"/> True
CountUpdatedRecords	<input checked="" type="checkbox"/> True
EnableDelete	<input checked="" type="checkbox"/> True
EnableInsert	<input checked="" type="checkbox"/> True
EnableUpdate	<input checked="" type="checkbox"/> True
FastUpdates	<input type="checkbox"/> False
FetchGeneratorsPoint	gpDeferred
GeneratorName	
KeyFields	
LockMode	lmNone
LockPoint	lpDeferred
LockWait	<input type="checkbox"/> False
ReadOnly	<input type="checkbox"/> False
RefreshDelete	<input checked="" type="checkbox"/> True
RefreshMode	rmOnDemand
RequestLive	<input checked="" type="checkbox"/> True
UpdateChangedFields	<input checked="" type="checkbox"/> True
UpdateMode	upWhereKeyOnly
UpdateNonBaseFields	<input type="checkbox"/> False
UpdateTableName	

Query Editor ... Execute

All shown

# Chapter 3: Configuring FireDAC 71

**Figure 3-8: The UpdateOptions property for an FDCommand in the Object Inspector**

## Inspector

72 Delphi in Depth: FireDAC

### GENERAL UPDATING

Use these properties to configure which operations are acceptable for a dataset.

#### Property

##### Name

EnableDelete Set to True to permit records to be deleted from datasets. The default is True.

EnableInsert

Set to True to allow records to be added to datasets. The default is True.

EnableUpdate Set to True to allow editing of datasets. The default is True.

ReadOnly

Set to True to prohibit changes to datasets. The default is False.

RequestLive

Set to True to request an editable result set. This property is a shortcut for many of the other general updating properties.

The default is True.

### LOCKING

Use these properties to configure how FireDAC locks records.

#### Property Name

LockMode Defines how FireDAC locks records. Possible values are lmNone, lmPessimistic (if supported), and lmOptimistic. The default is lmNone.

**LockPoint** Controls the point at which locking (when locking) occurs, either upon edit or upon post. Possible values are lpImmediate and lpDeferred. The default is lpDeferred.

**LockWait** When pessimistic locking is being used, set to True to have FireDAC wait until the lock is available. Otherwise an exception will be raised. The default is False.

## Chapter 3: Configuring FireDAC 73

### **REFRESHING**

Use these properties to configure how FireDAC refreshes data after posting changes.

#### **Property**

##### **Name**

**RefreshMode** Defines how FireDAC refreshes records after updates or inserts. Possible values are rmManual, rmOnDemand, and rmAll. The default is rmOnDemand.

**RefreshDelete** Set to True to have FireDAC remove from local storage any records not found following a refresh. The default is True.

### **AUTOMATIC INCREMENTING**

Use these properties to control how FireDAC works with auto-increment fields and generators.

#### **Property**

##### **Name**

##### **AutoIncFields**

A comma-separated list of auto-increment fields. Note that FireDAC may automatically recognize those fields when fiMeta is in FetchOptions.Items and for some DB's connection parameters, such as

ExtendedMetadata=True. The default is any empty string.

**FetchGeneratorsPoint** Controls the point at which a new value is obtained

from a generator. Possible values are gpNone, gpImmediate, and gpDeferred. The default is gpDeferred.

#### GeneratorName

The name of the generator. The default is an empty string.

### 74 Delphi in Depth: FireDAC

#### **POSTING CHANGES**

Use these properties to configure how FireDAC applies changes to the database.

The properties whose names begin with Check affect the associated properties of the dataset's TFields. The remaining properties affect the associated properties of the datasets themselves.

#### **Property**

##### **Name**

##### AutoCommitUpdates

When set to True, automatically commit updates after posting updates when cached updates is True.

Otherwise a manual call to CommitUpdates call is required. The default is False. This property was introduced in Delphi 10 Seattle.

CountUpdatedRecords When set to True, FireDAC raises an exception if an update to a single record affects more or less than one record. The default value is True.

##### CheckReadOnly

Set to True to flag read-only fields (fields which cannot be modified) as read-only. The default is True

##### CheckRequired

Set to True to flag required fields (fields which cannot accept a null value) as required. The default is False

for FDMemTables and True for all other FireDAC datasets.

#### CheckUpdatable

Set to True to have FireDAC raise an exception if an attempt is made to modify a field whose properties do not permit changes or when the dataset is not in an editing state. The default is True.

#### FastUpdates

Set to True to optimize update speed. This property is a shortcut to setting other properties of the TFDUpdateOptions class, and when set to True may result in the overwriting of other user's posts. The default is False.

#### KeyFields

A comma-separated list of key fields. Use to add pfInKey to TField.ProviderFlags when automatic assignment fails. The default is an empty string.

### Chapter 3: Configuring FireDAC 75

**UpdateChangedFields** Set to True to include non-null fields in generated INSERT statements, and changed fields in UPDATE statements. The default is True.

**UpdateNonBaseFields** Set to True to include non-base fields (fields not associated with the update table) in UPDATE queries, which is appropriate only under specific conditions.

The default is False.

#### UpdateMode

UpdateMode defines how FireDAC generates update, insert, and delete queries. Possible values are upWhereKeyOnly, upWhereChanged, and

upWhereAll. The default value is upWhereKeyOnly.

See the section Understanding

UpdateOptions.UpdateMode later in this chapter for a detailed discussion of this property.

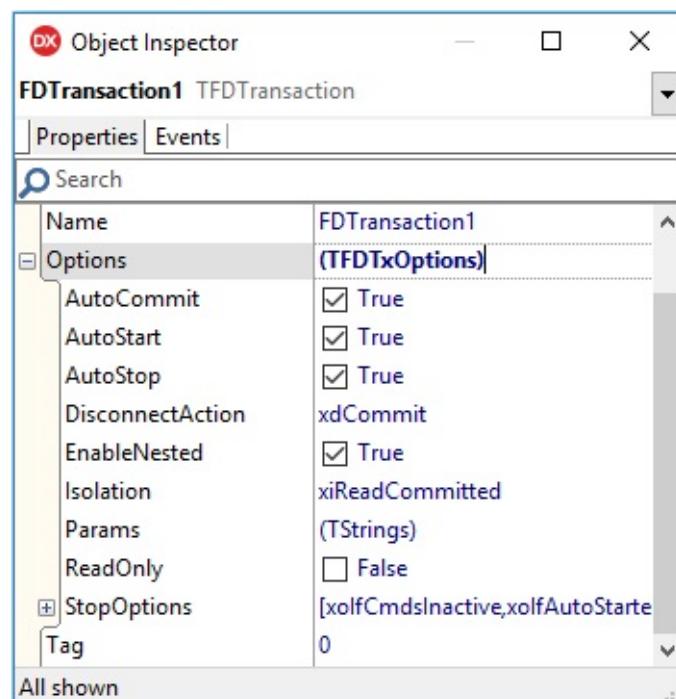
UpdateTableName

The name of the table to update. Use when automatic detection of the update table fails. The default is an empty string.

## Transaction Options

Use the properties of the TFDTxOptions class to configure how FireDAC manages transactions. Only the FDConnection and FDTransaction classes have a transaction object property, named TxOptions and Options, respectively.

Figure 3-9 shows the expanded Options property for an FDTransaction in the Object Inspector.



76 Delphi in Depth: FireDAC

**Figure 3-9: The Options property for an FDTransaction in the Object Inspector**

## ISOLATION LEVEL

Use this Isolation property to control transaction isolation.

### **Property Name**

**Isolation** Set the transaction isolation level, which dictates how changes made during a transaction are viewed by other users (connections). Possible values are xiUnspecified, xiDirtyRead, xiReadCommitted, and xiRepeatableRead. The default is xiReadCommitted.

### **UPDATE ABILITY**

Use this ReadOnly property to define a read-only transaction.

### **Property Name**

**ReadOnly** Set to True to indicate to the database that the transaction will only read data from the database. The default is False.

Chapter 3: Configuring FireDAC 77

### **AUTOMATIC COMMITTING**

Use these properties to configure the automatic committing of transactions.

AutoStart, AutoStop, and StopOptions are specifically intended for InterBase and FireBird, both of which do not support automatic transaction management.

#### **Property**

##### **Name**

**AutoCommit** When set to True, a transaction is automatically started before executing an SQL statement, and upon completion is automatically committed or rolled back. The default is True.

##### **AutoStart**

When AutoCommit is False, set to True to automatically start a transaction before executing an SQL command. The default value is True.

##### **AutoStop**

When AutoCommit is False, set to True to automatically finish a transaction by calling commit or rollback when SQL

execution completes. The default is True.

### StopOptions

Use the flags of this property to configure under which circumstances FireDAC automatically terminates a started transaction. Possible flags are xoIfCmdsInactive, xoIfAutoStarted, and xoFinishRetaining. The default value is [xoIfAutoStarted, xoIfCmdsInactive].

## DBMS-SPECIFIC PARAMETERS

Use this Params property to define parameters specific to a particular database.

### Property Name

#### Params

A collection of name/value pairs and names used to define additional transaction parameters. The default is an empty collection.

78 Delphi in Depth: FireDAC

## DISCONNECTION ACTION

This DisconnectAction property defines the default action to take when a transaction is complete.

### Property

#### Name

DisconnectAction Set to define the default action to take when a connection is being disconnected and a transaction is still active.

Possible values include xdNone, xdCommit, and xdRollback. The default is xdCommit.

## NESTING

Use this EnableNested property to enable or disable nested transactions.

### Property

#### Name

EnableNested Set to True to enable nested transactions. When enabled,

calling StartTransaction from within an active transaction context initiates (or emulates) a nested transaction. The default is True.

## **Understanding UpdateOptions.UpdateMode**

In the opening chapter of this book, I stressed the similarities between the FireDAC and the BDE. Specifically, that you can replace a TTable or TQuery with a TFDQuery and most of your code will work nicely without adjustment.

There is one difference, however, that is significant, and it's related to the default value of a property in FireDAC whose value is different from a similar property found in the BDE. And because the effects of this property are so profound it is important that you know about this difference when you decide to start using FireDAC. The property is *UpdateMode*.

UpdateMode is a property of TFDUpdateOptions, which is the type associated with the UpdateOptions property found in a number of classes in the FireDAC

framework. These include FDManager, FDConnection, FDQuery, FDMemTable, FDTable, FDCommand, and FDSchemaAdapter.

The UpdateMode property in the BDE and the UpdateOptions.UpdateMode property are both of the type TUpdateMode. The following is the declaration of the TUpdateMode type:

Chapter 3: Configuring FireDAC 79

TUpdateMode = (upWhereAll, upWhereChanged, upWhereKeyOnly);

When UpdateMode is set to upWhereAll, all non-BLOB fields are included in the WHERE clause of UPDATE and DELETE queries. This results in update failures if any of the non-BLOB fields of the underlying record were modified since the time that the table was opened, the query or stored procedure was executed, or the FDMemTable was loaded.

This approach is known as optimistic locking, and when two or more users are trying to apply changes at about the same time to the same record, only the first user to apply changes will succeed. All others will fail.

When UpdateMode is set to upWhereChanged, only the primary key fields and

the fields that have been modified are included in the WHERE clause of

UPDATE queries. (Again, INSERT queries are not affected. DELETE queries continue to use an exact match criteria since there are no changed fields in a deleted record.) As long as none of the primary key fields of an updated record are affected, and all non-primary key fields that have been modified have also not been updated in the underlying table since the time the data was loaded into the dataset, these queries should succeed.

Using upWhereChanged permits two or more users to apply their changes to the underlying database so long as none of them have made changes to common

fields. For example, if one user changed a field called Address, and another changed a field called PhoneNumber, and neither of these fields are part of the primary key, both users will successfully apply their changes. This type of update permits merges.

The final UpdateMode value is upWhereKeyOnly. With UpdateMode set to upWhereKeyOnly, the WHERE clause of UPDATE queries only includes the values of the primary key fields. (INSERT and DELETE queries continue to act as they do with upWhereChanged.) Using this mode, so long as the key fields of the underlying record have not been changed, the updates are applied, replacing any updates that other users may have applied.

Assuming that key fields are not touched (this is a pretty safe assumption in most database architectures), the use of upWhereKeyOnly permits everyone to succeed in posting their changes. As a result, the last user to post is the user whose data appears in the underlying database.

The default value of UpdateOptions.UpdateMode is upWhereKeyOnly, and this

is a really big deal. It means that if you use FireDAC datasets without changing

## 80 Delphi in Depth: FireDAC

this default value, there is a good chance that some users may report that their changes have *disappeared*.

Here is how this can happen. If two or more users read the same record (by way of a query, stored procedure call, or by opening an FDTable), and two or more users post a change to that record, the last user to post their record will replace those changes posted before them. What's problematic about this is that the users who posted before the last user will have no idea that their changes have been overwritten.

By comparison, most databases use either pessimistic locking (the first user to start editing the record prevents any other user from editing the record until changes have been posted), or optimistic locking (once the first user posts a change to a record, subsequent attempts to post to that same record will fail, since the original record no longer can be found, based on the WHERE clause predicates, the Boolean expressions in an SQL WHERE clause). In these scenarios, the first user to post wins, and other users must first re-read the record, after which they can decide whether or not to update the newly posted contents.

FireDAC defaults to an UpdateMode of upWhereKeyOnly, since the queries required to update the database tend to execute faster. It is up to you, however, to decide whether or not the performance improvement is more important than the possible loss of data.

The DataSetProvider class, the class which ClientDataSets use to resolve changes back to the underlying database, and BDE datasets (TTable, TQuery, and TStoredProc) also have an UpdateMode property. For these objects, the default value of UpdateMode is upWhereAll, the conservative setting that prevents a user from overwriting another user's edits.

So, the bottom line is this. You need to understand how FireDAC's UpdateOptions.UpdateMode affects how records are updated in the underlying database, and set this property to the value that meets the needs of your application.

In the next chapter, I cover the basics of data access with FireDAC.

# Chapter 4

## Basic Data Access

In Chapter 1, you learned about the many features that make FireDAC the data access mechanism of choice. In Chapter 2, you learned how to make connection to your database, and in Chapter 3, you discovered that FireDAC is highly

configurable, offering a wealth of options for fine-tuning your data access. Now it's time to get serious about working with the actual data in your database.

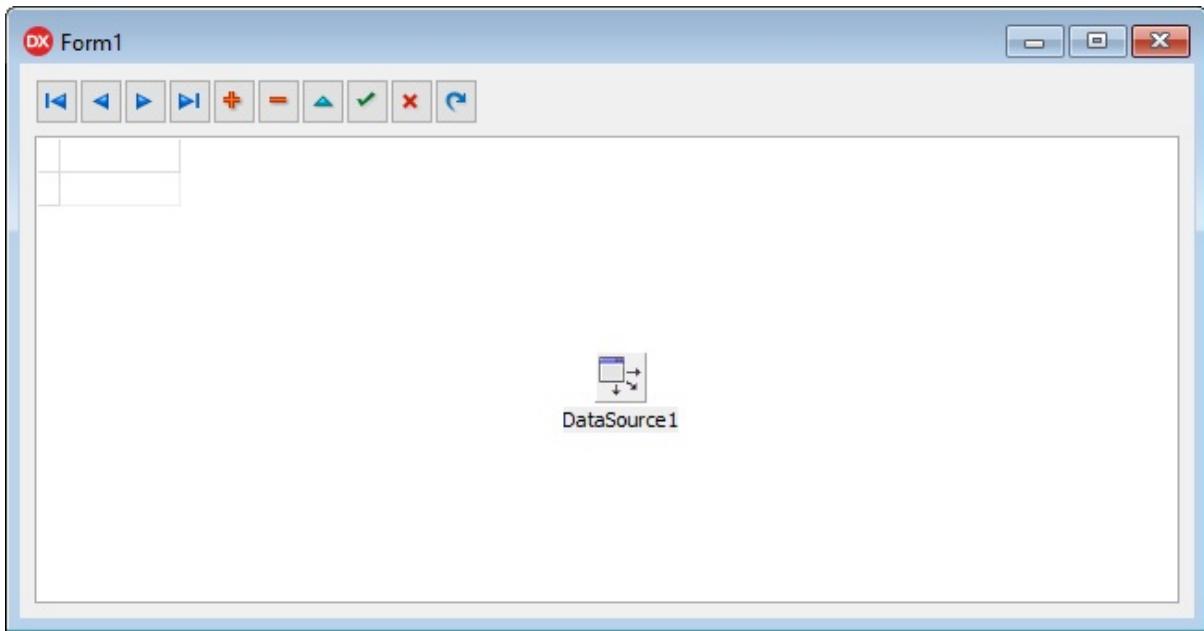
This chapter is designed to introduce you to the basics of data access with FireDAC. I will begin by showing you how to retrieve and navigate the data in your database using your user interface. I will also introduce the use of the three primary FireDAC datasets — FDTables, FDQueries, and FDStoredProcs.

Let's begin by creating a very simple example, one where we can display the contents of a table in a grid. These steps, and nearly all of the remaining examples in this book assume that you have InterBase installed, which should have happened automatically when you installed Delphi. If you do not have InterBase installed, you should download the free developer's edition and install it before you continue.

*Note: If you cannot open the EMPLOYEE name connection from the Data Explorer, as described in the following steps, please visit Appendix A for more information about connecting to InterBase. You may simply need to start the InterBase server from the Services applet.*

Use the following steps to easily create an application that permits you to view and edit the employee table in the employee.gdb database that is installed along side with InterBase:

1. Create a new project in Delphi by selecting File | New | VCL Forms Application from Delphi's main menu.
2. Using the Tool Palette, place a DBNavigator, a DBGrid, and a DataSource onto the design surface of your form. If you are unfamiliar with the Tool Palette, you can start typing into the field at the top (next to the magnifying glass icon) the name of a component you want to

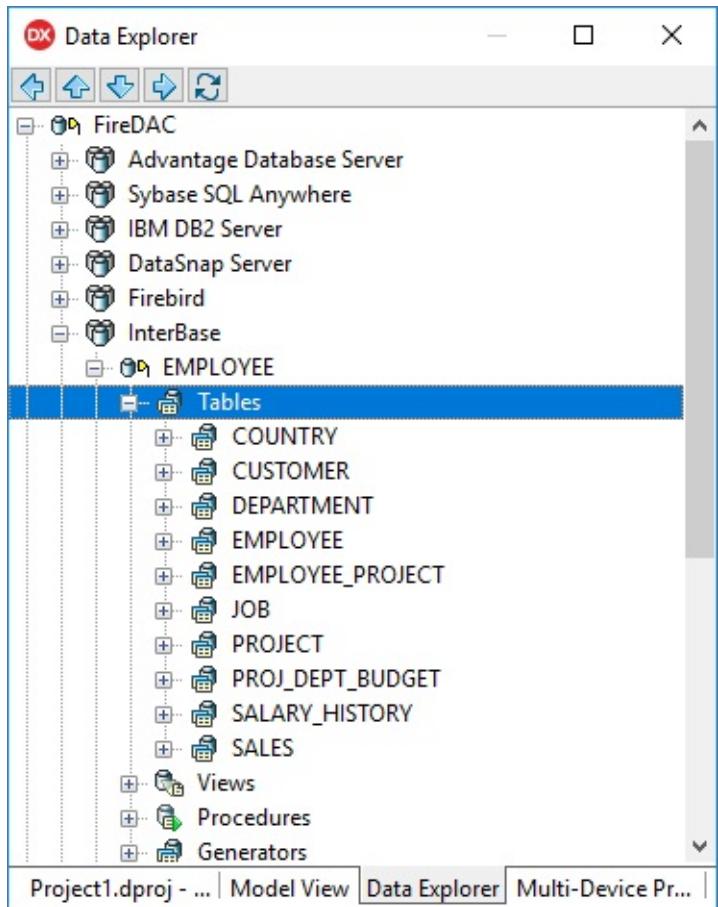


## 82 Delphi in Depth: FireDAC

place. For example, start typing DataSource, and the Tool Palette will progressively filter the available components until just the DataSource is displayed. Once visible, drag the DataSource onto your form's design surface and release it. When you are done placing components your form should look something like that shown in Figure 4-1.

**Figure 4-1: A newly designed user interface**

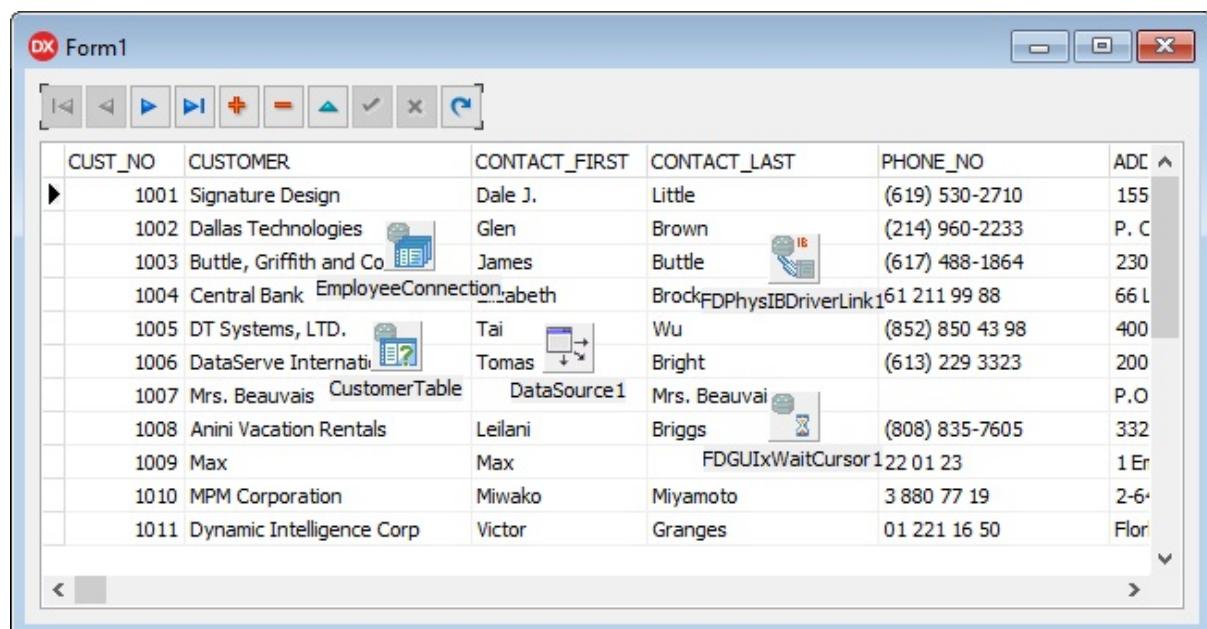
3. From Delphi's main menu select View | Data Explorer to display the Data Explorer pane. This pane should appear in the upper-right corner of your designer, by default.
4. From the Data Explorer, expand the FireDAC node, and then expand the InterBase node, and then the EMPLOYEE node under the InterBase node. Finally, expand the Tables node. Your Data Explorer should now look something like that shown in Figure 4-2.
5. Now click on the CUSTOMER node under Tables, and hold down your left mouse button while dragging the CUSTOMER table to your form and release the button. Delphi responds by placing an FDConnection named EmployeeConnection along with an FDQuery named CustomerTable onto your design surface.



## Chapter 4: Basic Data Access 83

**Figure 4-2: The expanded Table node of the EMPLOYEE database in the Data Explorer**

6. Connect your DataSource to your FDQuery by selecting the DataSource and then using the Object Inspector to set the DataSet property to CustomerTable. Next, select the DBNavigator and set its DataSource property to DataSource1. Similarly, set the DBGrid's DataSource property to point to DataSource1.
7. You are almost done, but you'll need a couple of additional components if you want to run this application. Using the Tool Palette, place both an FDPhysIBDriverLink and an FDGUIxWaitCursor component onto the form (this step is unnecessary in some of the more recent versions of Delphi).
8. Finally, select the FDQuery, and using the Object Inspector, set its Active property to True. Your form should now look something like that shown in Figure 4-3.



The screenshot shows a Delphi application window titled "Form1". At the top is a toolbar with various icons for navigating and managing data. Below the toolbar is a DBGrid control displaying a dataset. The grid has six columns: "CUST\_NO", "CUSTOMER", "CONTACT\_FIRST", "CONTACT\_LAST", "PHONE\_NO", and "ADC". The data consists of 11 records, each representing a customer. The first record, with CUST\_NO 1001 and CUSTOMER "Signature Design", is currently selected, indicated by a blue border around its row.

CUST_NO	CUSTOMER	CONTACT_FIRST	CONTACT_LAST	PHONE_NO	ADC
1001	Signature Design	Dale J.	Little	(619) 530-2710	155
1002	Dallas Technologies	Glen	Brown	(214) 960-2233	P. C
1003	Buttle, Griffith and Co.	James	Buttle	(617) 488-1864	230
1004	Central Bank	Elizabeth	Brocket	61 211 99 88	66 L
1005	DT Systems, LTD.	Tai	Wu	(852) 850 43 98	400
1006	DataServe International	Tomas	Bright	(613) 229 3323	200
1007	Mrs. Beauvais		Mrs. Beauvais		P.O
1008	Anini Vacation Rentals	Leilani	Briggs	(808) 835-7605	332
1009	Max	Max		22 01 23	1 Er
1010	MPM Corporation	Miwako	Miyamoto	3 880 77 19	2-6
1011	Dynamic Intelligence Corp	Victor	Granges	01 221 16 50	Flor

## 84 Delphi in Depth: FireDAC

**Figure 4-3: The newly created form is the designer**

You can now run this form by pressing F9, or by clicking the Run button on Delphi's Toolbar, or by select Run | Run from Delphi's main menu. The running form is shown in Figure 4-4.

**Figure 4-4: The new form running as a Windows application**

## Chapter 4: Basic Data Access 85

You can now click on the buttons of the DBNavigator to navigate and edit the records shown in the grid, and can also scroll the grid using the scroll bar or your navigation keys on your keyboard. Furthermore, if you edit a cell in the grid, giving it a valid value, you can post that record to the underlying database either by clicking the Post button on the DBNavigator (the button with the check mark icon displayed on it), or by navigating off of the edited record.

*Note: There are a number of constraints defined in the InterBase employee database, and these may limit the data you can apply to some of its tables. I mention this in case one or more of your edits are rejected when you attempt to post an edited or inserted record.*

Most of your applications will be more complicated than this one, but this example demonstrates how easy it is to create an application that can read and write data from your database using FireDAC. And in this preceding example, all of our interactions with the data, including editing and navigation, was performed through the provided user interface. In the following section I will take a more in depth look at interacting with data through the user interface.

Later in this chapter, I will discuss the three primary FireDAC datasets in greater detail. For information on programmatic navigation and editing, refer to *Chapter 6, Navigating and Editing Data*.

## The User Interface and Data Binding

Most of Delphi's visual components can be associated with a dataset (albeit indirectly), after which those components can be used to display data-related information (either the actual data, or information about the data, such as the relative position of the current record within the dataset), and in some cases, enable the user to navigate the dataset.

How this data binding is implemented depends on which of Delphi's two component libraries you are using. The VCL (visual component library) supports data-awareness with two distinct mechanisms, one that is limited to a small set of components and another, called LiveBindings that can be applied to almost any visual control. By comparison, FireMonkey controls support only LiveBindings. Note that there are also third-party components that support both VCL-like data awareness as well as LiveBindings.



### 86 Delphi in Depth: FireDAC

This section looks at dataset navigation by way of data-awareness in visual controls. It begins with a look at the VCL-specific framework for data-awareness, a mature mechanism for navigation only available in Windows-based applications. This discussion is followed by a look at navigation as supported through LiveBindings.

#### Navigation and VCL Data Links

Delphi's VCL provides two types of controls that provide data navigation. The first type is a navigation-specific control, and Delphi provides you with one control in this category, the DBNavigator. The second type includes controls that permit navigation as well as data display and editing.

Whether used for navigation or the navigable display of data, these VCL controls have one thing in common — they implement their behavior through an encapsulated class that descends from TDataLink. These DataLink classes hook their owning VCL control to a DataSource, which in turn needs to point to a DataSet. The interaction between the DataLink and the DataSource is what

produces the data-awareness that these VCL controls support. This behavior, however, is transparent to the developer using these VCL controls, in that the

developer never actually interacts with the internal DataLink. As a result, no more needs to be said about DataLinks in the context of VCL navigation.

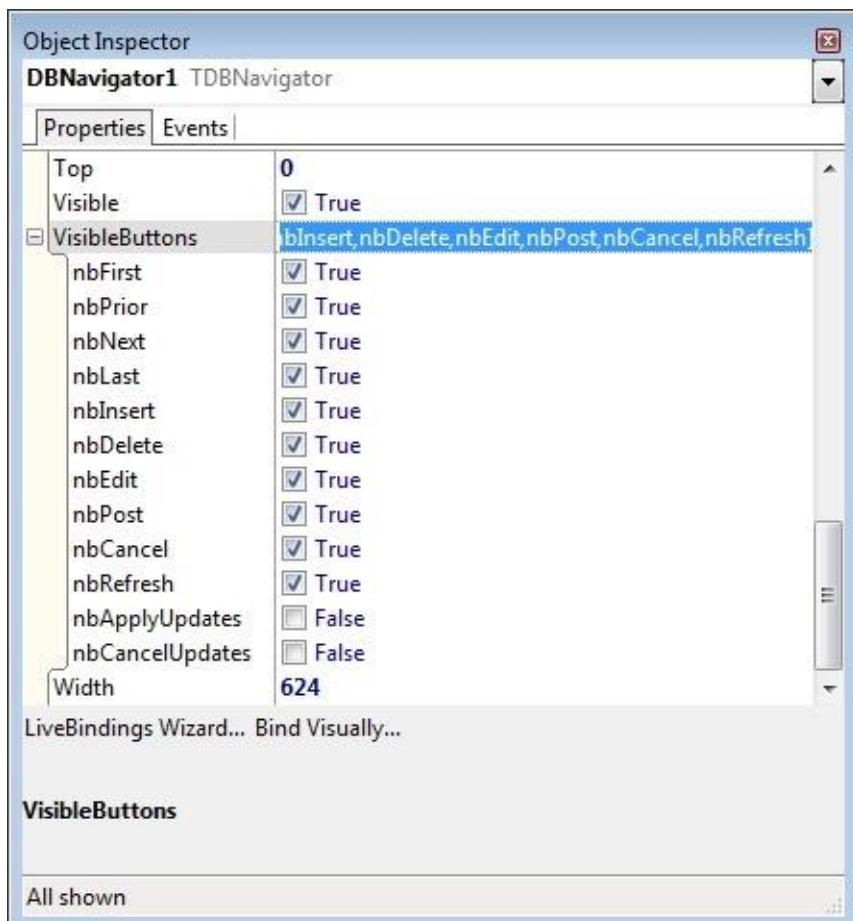
Let's begin this discussion with a look at the VCL DBNavigator.

## THE DBNAVIGATOR

The DBNavigator, shown here, provides a video player-like interface for navigating data and managing records. Record navigation is provided by the first four buttons, which correspond to First, Prior, Next, and Last. The last six buttons provide record management, and correspond to Insert, Delete, Edit, Post, Cancel, and Refresh.

As you can see in this illustration, some of the buttons are not enabled. This DBNavigator is associated with an active DataSet, and the enabled properties of the buttons are context sensitive. For example, you can tell by this image that the current record is the first record in the dataset, since the First and Prior buttons are not enabled. Furthermore, the dataset is in the Browse state, since the Edit button is active, and the Cancel and Post buttons are not active.

You can control which buttons are displayed by a DBNavigator through its `VisibleButtons` property. For example, if you are using the DBNavigator in conjunction with a FireDAC FDMemTable that reads and writes its data from a



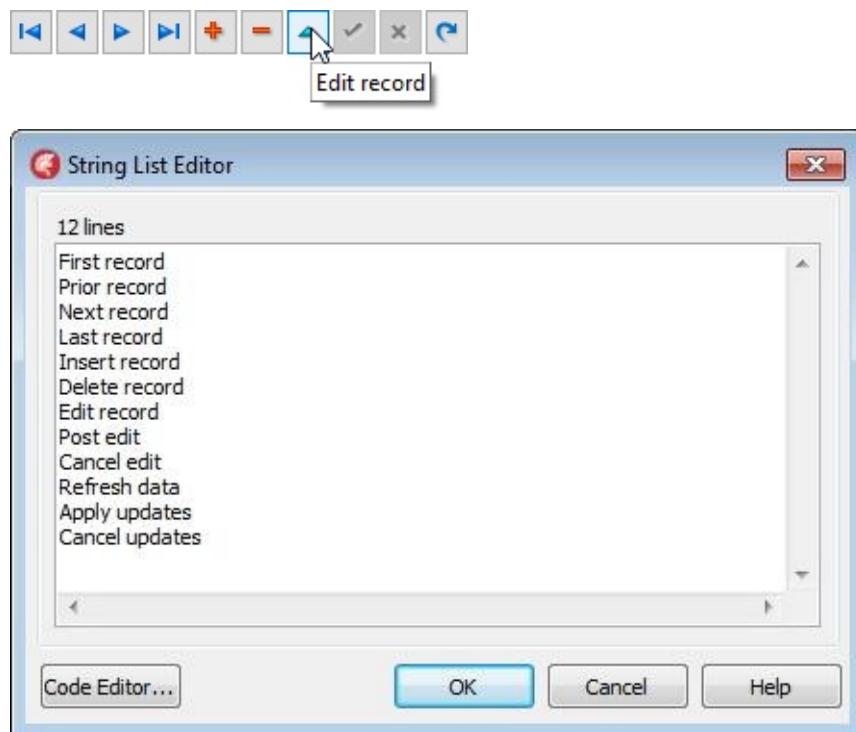
## Chapter 4: Basic Data Access 87

local file, you will want to suppress the display of the Refresh button, since attempting to refresh an FDMemTable that uses local files makes no sense.

Figure 4-5 shows this property editor expanded, permitting you to toggle which buttons you want to be available.

**Figure 4-5: Use the Object Inspector at design time to control which buttons are visible in the DBNavigator**

Another DBNavigator property whose default value you may want to change is ShowHint. The glyphs used for the various buttons of the DBNavigator are not necessarily obvious to all (take the Edit button, for example). To improve this situation, setting ShowHint to True supplements the glyphs with popup help hints, as shown in the following illustration:



### 88 Delphi in Depth: FireDAC

You don't even have to accept the default hints offered by the DBNavigator.

Using the Hints property editor, which is a StringList editor, you can supply whatever text you want for each of the buttons in your DBNavigator. To do this, you change the text associated with the corresponding button in the DBNavigator based on the position of the button in the DBNavigator. The

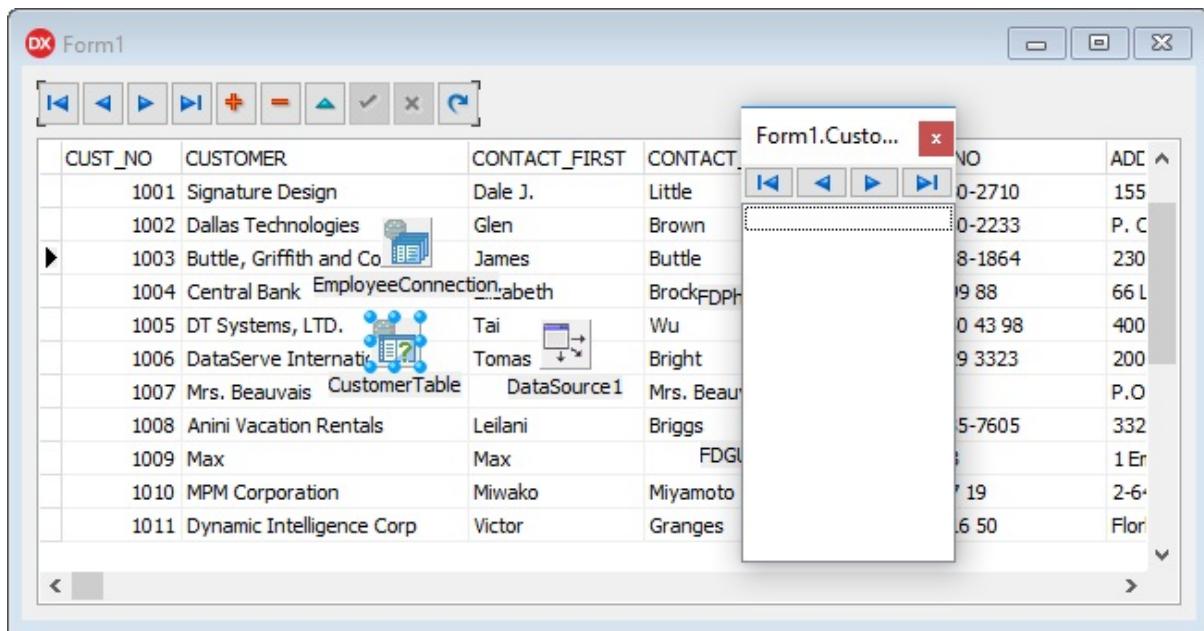
StringList editor, with alternative DBNavigator button hints, is shown in Figure 4-6.

**Figure 4-6: Use the Hints property editor, in conjunction with the ShowHint property, to provide custom hints for a DBNavigator**

Before leaving this topic, I want to mention an often overlooked gem in Delphi.

The Fields Editor, which will be discussed in more detail in later chapters in this book , includes a small navigator control that you can use to navigate an active dataset at design time.

This can be seen in Figure 4-7, where a form that includes a DBNavigator and a DBGrid are associated with an active Table. Notice that the small navigator at the top of the Fields Editor and the DBNavigator are both indicating that the dataset is neither on the first record nor the last record (since First, Next, Last,



## **Chapter 4: Basic Data Access 89**

and Prior are all enabled), which is confirmed by the small arrow indicator in the DBGrid. Before this figure was captured, the Fields Editor navigator had been used to advance to the third record in the dataset.

**Figure 4-7: The Fields Editor has a handy little navigator that you can use to navigate records of an active dataset at design time**

### **MULTI-RECORD VCL CONTROLS AND NAVIGATION: DBGRID AND**

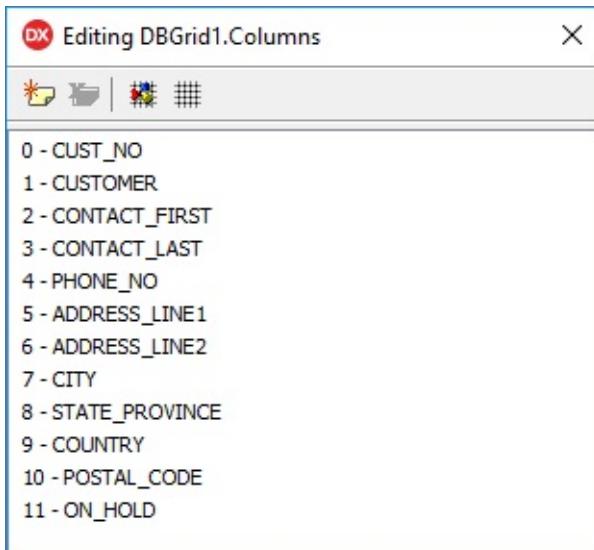
#### **DBCTRLGRID**

The second category of VCL controls that provides navigation is the multi-record controls. Delphi includes two of these controls: DBGrid and DBCtrlGrid.

(Other VCL controls, available from third-party vendors, such as Raize Software, TMS Software, and Woll2Woll Software, to name a few, also provide navigation-supporting multi-record controls.)

A DBGrid displays data in a row/column format, as shown in Figure 4-7. By default, all fields of the dataset are displayed in the DBGrid.

You can control which fields are displayed, as well as specific column characteristics, such as color, typeface, and caption by editing the DBGrid's Columns property using the Columns property editor. To display the Columns property editor, either double-click the DBGrid, right-click the DBGrid and select Columns Editor, or click the ellipsis that appears when the Columns property of a DBGrid is selected in the Object Inspector.



## 90 Delphi in Depth: FireDAC

In Figure 4-8, the Add All Fields button of the Columns property editor has been clicked. These columns can then be selectively deleted, or their properties adjusted by selecting a specific Column in the Columns Editor and then using the Object Inspector to change its properties:

**Figure 4-8: The Columns Editor for a DBGrid**

A DBCtrlGrid, by comparison, is a limited, multi-record container. It is limited in that it can only hold certain Delphi components, such as Labels, DBEdits, DBLabels, DBMemos, DBImages, DBComboBoxes, DBCheckboxes,

DBLookupComboBoxes, and DBCharts. The DBCtrlGrid is useful when you want to create a specialized interface, such as the one shown in Figure 4-9. In this example, records from the EMPLOYEE table of the employee.gdb database is displayed in a 3 by 3 grid.

DBCtrlGrid Example

<table border="1"> <tr><td>EMP_NO</td><td>DEPT_NO</td><td>FULL_NAME</td></tr> <tr><td>2</td><td>600</td><td>Nelson, Robert</td></tr> <tr><td>PHONE_EXT</td><td>JOB_CODE</td><td>JOB_GRADE</td></tr> <tr><td>250</td><td>VP</td><td>2</td></tr> <tr><td colspan="3">HIRE_DATE</td></tr> <tr><td colspan="3">12/29/2007</td></tr> </table>	EMP_NO	DEPT_NO	FULL_NAME	2	600	Nelson, Robert	PHONE_EXT	JOB_CODE	JOB_GRADE	250	VP	2	HIRE_DATE			12/29/2007			<table border="1"> <tr><td>EMP_NO</td><td>DEPT_NO</td><td>FULL_NAME</td></tr> <tr><td>4</td><td>621</td><td>Young, Bruce</td></tr> <tr><td>PHONE_EXT</td><td>JOB_CODE</td><td>JOB_GRADE</td></tr> <tr><td>233</td><td>Eng</td><td>2</td></tr> <tr><td colspan="3">HIRE_DATE</td></tr> <tr><td colspan="3">12/29/2007</td></tr> </table>	EMP_NO	DEPT_NO	FULL_NAME	4	621	Young, Bruce	PHONE_EXT	JOB_CODE	JOB_GRADE	233	Eng	2	HIRE_DATE			12/29/2007			<table border="1"> <tr><td>EMP_NO</td><td>DEPT_NO</td><td>FULL_NAME</td></tr> <tr><td>5</td><td>130</td><td>Lambert, Kim</td></tr> <tr><td>PHONE_EXT</td><td>JOB_CODE</td><td>JOB_GRADE</td></tr> <tr><td>22</td><td>Eng</td><td>2</td></tr> <tr><td colspan="3">HIRE_DATE</td></tr> <tr><td colspan="3">2/7/2008</td></tr> </table>	EMP_NO	DEPT_NO	FULL_NAME	5	130	Lambert, Kim	PHONE_EXT	JOB_CODE	JOB_GRADE	22	Eng	2	HIRE_DATE			2/7/2008		
EMP_NO	DEPT_NO	FULL_NAME																																																						
2	600	Nelson, Robert																																																						
PHONE_EXT	JOB_CODE	JOB_GRADE																																																						
250	VP	2																																																						
HIRE_DATE																																																								
12/29/2007																																																								
EMP_NO	DEPT_NO	FULL_NAME																																																						
4	621	Young, Bruce																																																						
PHONE_EXT	JOB_CODE	JOB_GRADE																																																						
233	Eng	2																																																						
HIRE_DATE																																																								
12/29/2007																																																								
EMP_NO	DEPT_NO	FULL_NAME																																																						
5	130	Lambert, Kim																																																						
PHONE_EXT	JOB_CODE	JOB_GRADE																																																						
22	Eng	2																																																						
HIRE_DATE																																																								
2/7/2008																																																								
<table border="1"> <tr><td>EMP_NO</td><td>DEPT_NO</td><td>FULL_NAME</td></tr> <tr><td>8</td><td>180</td><td>Johnson, Leslie</td></tr> <tr><td>PHONE_EXT</td><td>JOB_CODE</td><td>JOB_GRADE</td></tr> <tr><td>410</td><td>Mktg</td><td>3</td></tr> <tr><td colspan="3">HIRE_DATE</td></tr> <tr><td colspan="3">4/5/2008</td></tr> </table>	EMP_NO	DEPT_NO	FULL_NAME	8	180	Johnson, Leslie	PHONE_EXT	JOB_CODE	JOB_GRADE	410	Mktg	3	HIRE_DATE			4/5/2008			<table border="1"> <tr><td>EMP_NO</td><td>DEPT_NO</td><td>FULL_NAME</td></tr> <tr><td>9</td><td>622</td><td>Forest, Phil</td></tr> <tr><td>PHONE_EXT</td><td>JOB_CODE</td><td>JOB_GRADE</td></tr> <tr><td>229</td><td>Mngr</td><td>3</td></tr> <tr><td colspan="3">HIRE_DATE</td></tr> <tr><td colspan="3">4/17/2008</td></tr> </table>	EMP_NO	DEPT_NO	FULL_NAME	9	622	Forest, Phil	PHONE_EXT	JOB_CODE	JOB_GRADE	229	Mngr	3	HIRE_DATE			4/17/2008			<table border="1"> <tr><td>EMP_NO</td><td>DEPT_NO</td><td>FULL_NAME</td></tr> <tr><td>11</td><td>130</td><td>Weston, K. J.</td></tr> <tr><td>PHONE_EXT</td><td>JOB_CODE</td><td>JOB_GRADE</td></tr> <tr><td>34</td><td>SRep</td><td>4</td></tr> <tr><td colspan="3">HIRE_DATE</td></tr> <tr><td colspan="3">1/17/2009</td></tr> </table>	EMP_NO	DEPT_NO	FULL_NAME	11	130	Weston, K. J.	PHONE_EXT	JOB_CODE	JOB_GRADE	34	SRep	4	HIRE_DATE			1/17/2009		
EMP_NO	DEPT_NO	FULL_NAME																																																						
8	180	Johnson, Leslie																																																						
PHONE_EXT	JOB_CODE	JOB_GRADE																																																						
410	Mktg	3																																																						
HIRE_DATE																																																								
4/5/2008																																																								
EMP_NO	DEPT_NO	FULL_NAME																																																						
9	622	Forest, Phil																																																						
PHONE_EXT	JOB_CODE	JOB_GRADE																																																						
229	Mngr	3																																																						
HIRE_DATE																																																								
4/17/2008																																																								
EMP_NO	DEPT_NO	FULL_NAME																																																						
11	130	Weston, K. J.																																																						
PHONE_EXT	JOB_CODE	JOB_GRADE																																																						
34	SRep	4																																																						
HIRE_DATE																																																								
1/17/2009																																																								
<table border="1"> <tr><td>EMP_NO</td><td>DEPT_NO</td><td>FULL_NAME</td></tr> <tr><td>12</td><td>000</td><td>Lee, Terri</td></tr> <tr><td>PHONE_EXT</td><td>JOB_CODE</td><td>JOB_GRADE</td></tr> <tr><td>256</td><td>Admin</td><td>4</td></tr> <tr><td colspan="3">HIRE_DATE</td></tr> <tr><td colspan="3">5/1/2009</td></tr> </table>	EMP_NO	DEPT_NO	FULL_NAME	12	000	Lee, Terri	PHONE_EXT	JOB_CODE	JOB_GRADE	256	Admin	4	HIRE_DATE			5/1/2009			<table border="1"> <tr><td>EMP_NO</td><td>DEPT_NO</td><td>FULL_NAME</td></tr> <tr><td>14</td><td>900</td><td>Hall, Stewart</td></tr> <tr><td>PHONE_EXT</td><td>JOB_CODE</td><td>JOB_GRADE</td></tr> <tr><td>227</td><td>Finan</td><td>3</td></tr> <tr><td colspan="3">HIRE_DATE</td></tr> <tr><td colspan="3">6/4/2009</td></tr> </table>	EMP_NO	DEPT_NO	FULL_NAME	14	900	Hall, Stewart	PHONE_EXT	JOB_CODE	JOB_GRADE	227	Finan	3	HIRE_DATE			6/4/2009			<table border="1"> <tr><td>EMP_NO</td><td>DEPT_NO</td><td>FULL_NAME</td></tr> <tr><td>15</td><td>623</td><td>Young, Katherine</td></tr> <tr><td>PHONE_EXT</td><td>JOB_CODE</td><td>JOB_GRADE</td></tr> <tr><td>231</td><td>Mngr</td><td>3</td></tr> <tr><td colspan="3">HIRE_DATE</td></tr> <tr><td colspan="3">6/14/2009</td></tr> </table>	EMP_NO	DEPT_NO	FULL_NAME	15	623	Young, Katherine	PHONE_EXT	JOB_CODE	JOB_GRADE	231	Mngr	3	HIRE_DATE			6/14/2009		
EMP_NO	DEPT_NO	FULL_NAME																																																						
12	000	Lee, Terri																																																						
PHONE_EXT	JOB_CODE	JOB_GRADE																																																						
256	Admin	4																																																						
HIRE_DATE																																																								
5/1/2009																																																								
EMP_NO	DEPT_NO	FULL_NAME																																																						
14	900	Hall, Stewart																																																						
PHONE_EXT	JOB_CODE	JOB_GRADE																																																						
227	Finan	3																																																						
HIRE_DATE																																																								
6/4/2009																																																								
EMP_NO	DEPT_NO	FULL_NAME																																																						
15	623	Young, Katherine																																																						
PHONE_EXT	JOB_CODE	JOB_GRADE																																																						
231	Mngr	3																																																						
HIRE_DATE																																																								
6/14/2009																																																								

## Chapter 4: Basic Data Access 91

### Figure 4-9: The DBCtrlGrid provides flexibility in a multi-record control

Depending on which multi-record control you are using, you can navigate between records using UpArrow, DownArrow, Tab, Ctrl-End, Ctrl-Home, PgDn, PgUp, among others. These keypresses may produce the same effect as clicking the Next, Prior, Last, First, and so on, buttons in a DBNavigator. It is also possible to navigate the records of a dataset using the vertical scrollbar of these controls.

How you edit a record using these controls also depends on which type of control you are using as well as their properties. Using the default properties of these controls, you can typically press F2 or click twice on a field in one of these controls to begin editing.

Posting a record occurs when you navigate off an edited record. Inserting and deleting records, depending on the control's property settings, can also be achieved using Ins and Ctrl-Del, respectively. Other operations, such as Refresh, are not directly supported. Consequently, in most cases, multi-record controls are combined with a DBNavigator to provide a complete set of record

management options.

### Navigation and LiveBindings

LiveBindings, which were originally introduced in Delphi XE2, are specialized classes that associate string expressions, which are evaluated at runtime by Delphi's expression engine, with a property of a class. The expression that is evaluated often includes data from another class, obtained by reading a property



### 92 Delphi in Depth: FireDAC

or executing a method (and can even include reading multiple properties and/or executing several methods). In the end, the function of a LiveBinding is to assign data associated with one class to another, making the target class data-aware (we could also use the term *data binding* here, but for our purposes, the terms *data bound controls* and *data-aware controls* are interchangeable).

LiveBindings were introduced in order to provide data awareness in FireMonkey components, which were not designed to support the data awareness implemented via DataLinks. However, LiveBindings are also available in the VCL, and can be used to implement data awareness in almost any class, not just those that support DataLinks.

DataLinks and LiveBindings differ in several additional ways. One of the most obvious is that, unlike DataLinks, LiveBindings are not encapsulated in some other control. Instead, LiveBindings are standalone classes. It is through the configuration of a LiveBinding that the expression that gets assigned to the target component's property is defined and the source and target components are selected (though through the LiveBindings Designer this process is mostly transparent).

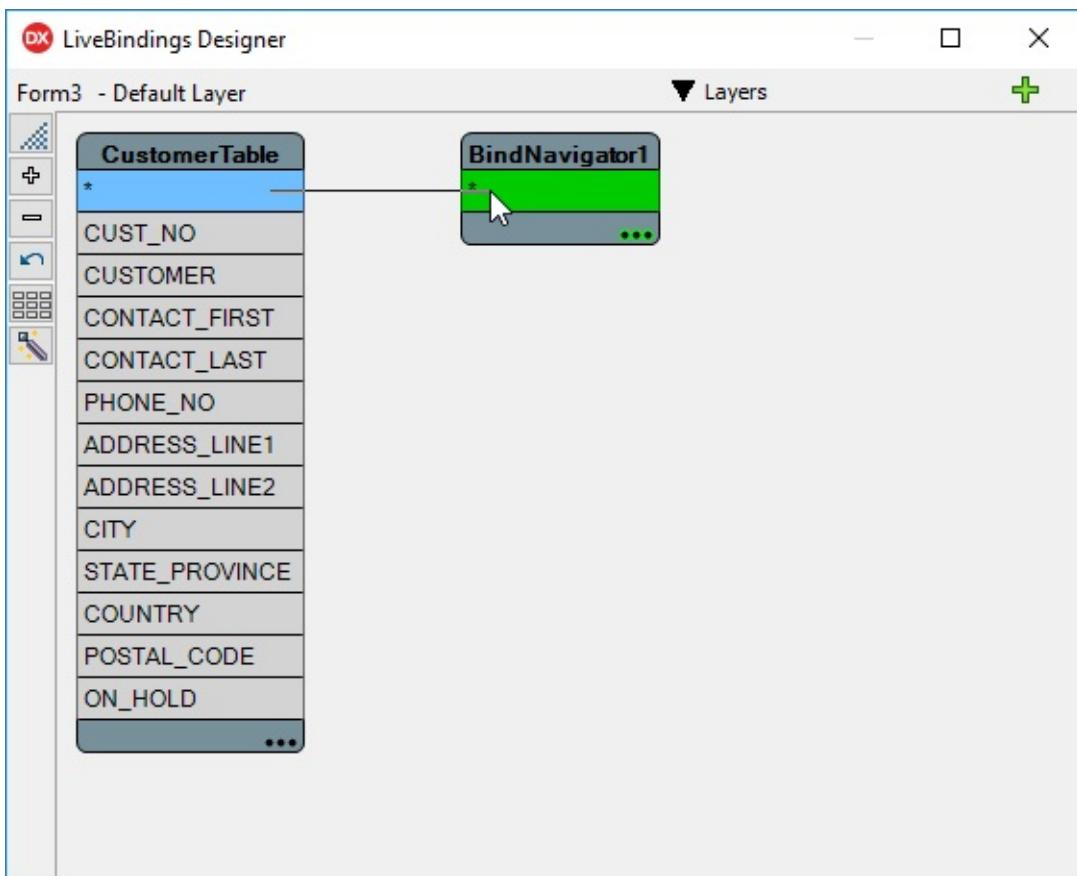
The second major difference between DataLinks and LiveBindings is that DataLinks wire a data-aware control to a DataSource. LiveBindings, by comparison, access the underlying DataSet through a BindSource, which is nearly always an instance of the BindSourceDB class. BindSourceDB classes are designed to make the Fields and other relevant data about their associated datasets available to the LiveBinding and the expression engine through properties.

Similar to the VCL, there are two general navigation-related mechanisms associated with LiveBindings: The BindNavigator and position-related LiveBinding expressions. Let's start with the BindNavigator.

## **The BindNavigator**

Technically speaking, the BindNavigator control is a FireMonkey control, and is not a LiveBinding. Consequently, it is not available for use with VCL applications.

The BindNavigator, like the DBNavigator, is a video player-like control that you can hook to a BindSourceDB in order to control record navigation and editing. The BindNavigator is shown in the following illustration:

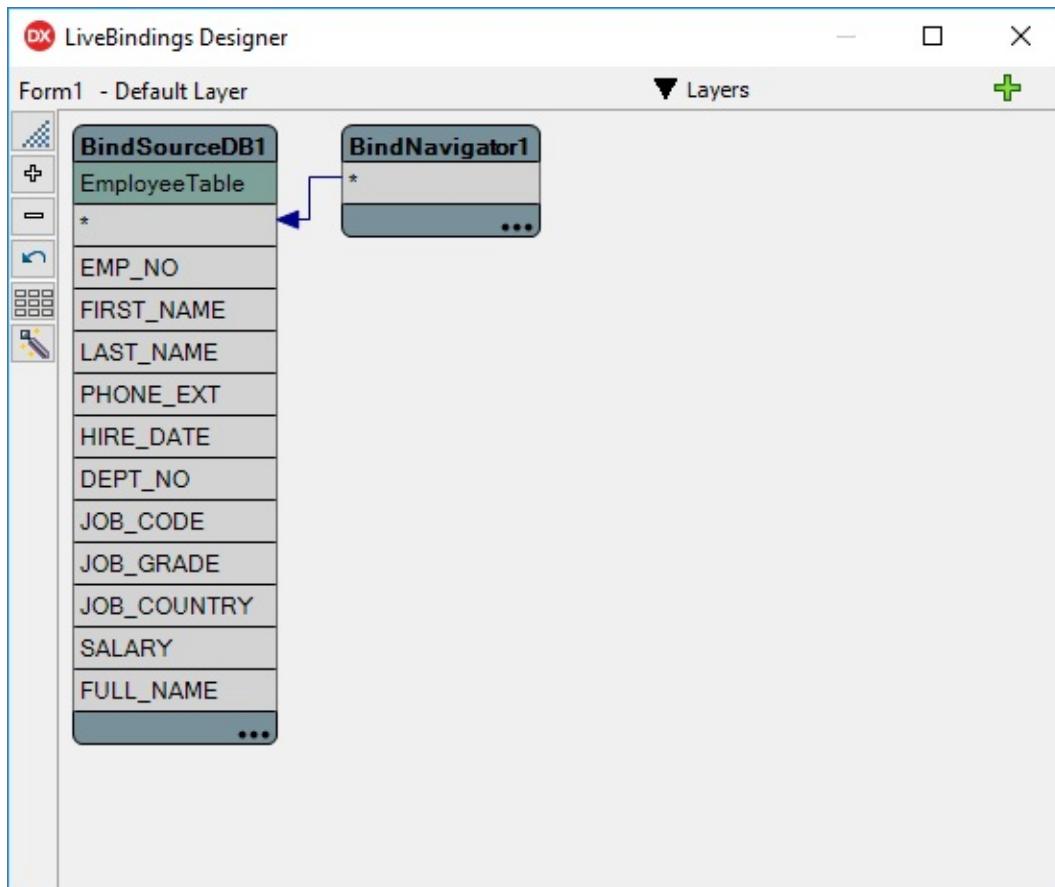


## Chapter 4: Basic Data Access 93

If your FireMonkey application already has a BindSourceDB control pointing to the appropriate DataSet, set its DataSource property to that BindSourceDB. If you do not yet have a BindSourceDB pointing to your FireDAC dataset, use the LiveBindings Designer to associate the dataset with the BindNavigator.

To display the LiveBindings Designer, select View | Tool Windows | LiveBindings Designer (use View | LiveBindings Designer with Delphi 10.1 Berlin and earlier). Figure 4-10 shows an FDQuery being associated with a BindNavigator by using the mouse to drag from the FDQuery to the BindNavigator. Once you release the link in the LiveBindings Designer, a BindSourceDB is created, associated with the dataset, and the BindNavigator is wired to that BindSourceDB. This can be seen in Figure 4-11.

**Figure 4-10: Drag from the FireDAC dataset to the BindNavigator to form an association**

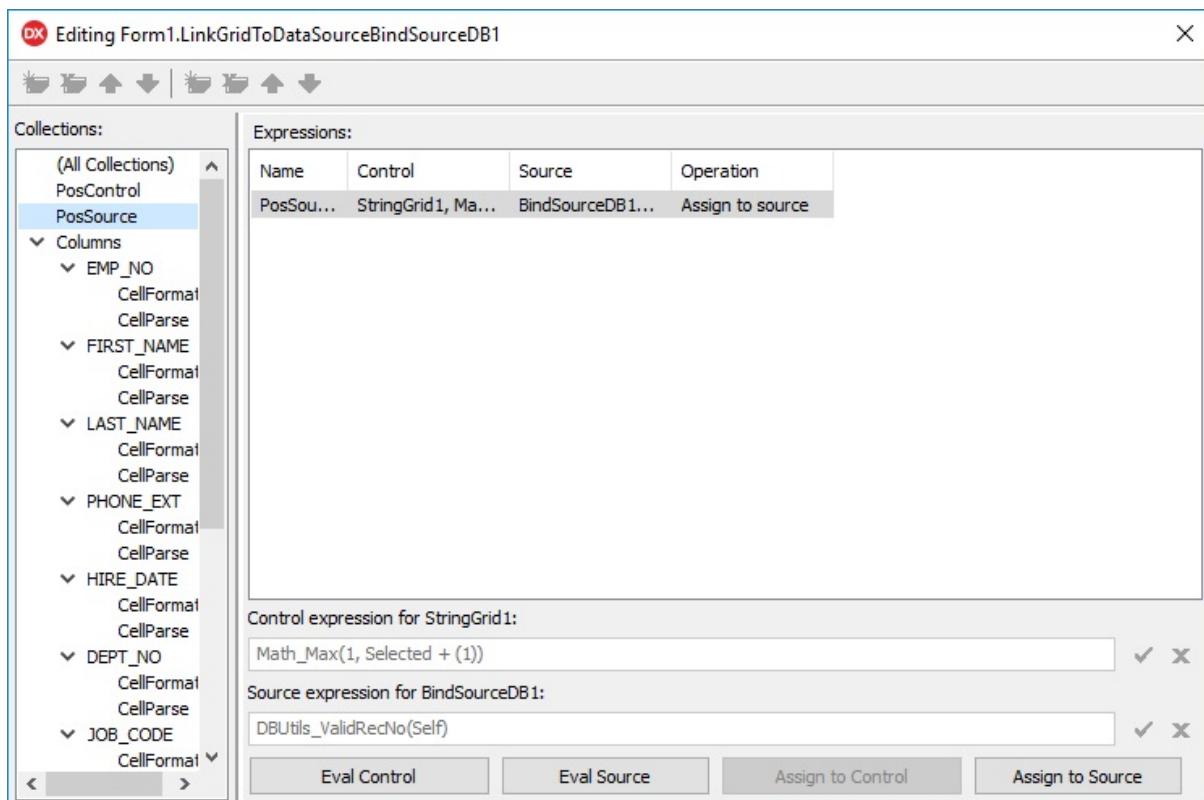


**Figure 4-11: A BindSourceDB is substituted in place of a FireDAC FDQuery, and a BindNavigator has been wired to that BindSourceDB**

Similar to the VCL's DBNavigator, the BindNavigator has a `VisibleButtons` property which you can use to configure which buttons appear. However, unlike the DBNavigator, the BindNavigator does not support Hints.

### Position-Related LiveBindings

There are a number of different LiveBindings, and many of these support a number of different types of LiveBinding expressions. As far as navigation goes, the relevant LiveBinding expressions are `PosControl`, `PosSource`, and `PosClear`. Here is how it works. When you use Visual LiveBindings to connect a multi-record control to a BindSourceDB (which is connected to your FireDAC dataset), the LiveBindings Designer creates a QuickBinding (a type of non-editable LiveBinding) that supports the binding expressions necessary to



## **Chapter 4: Basic Data Access 95**

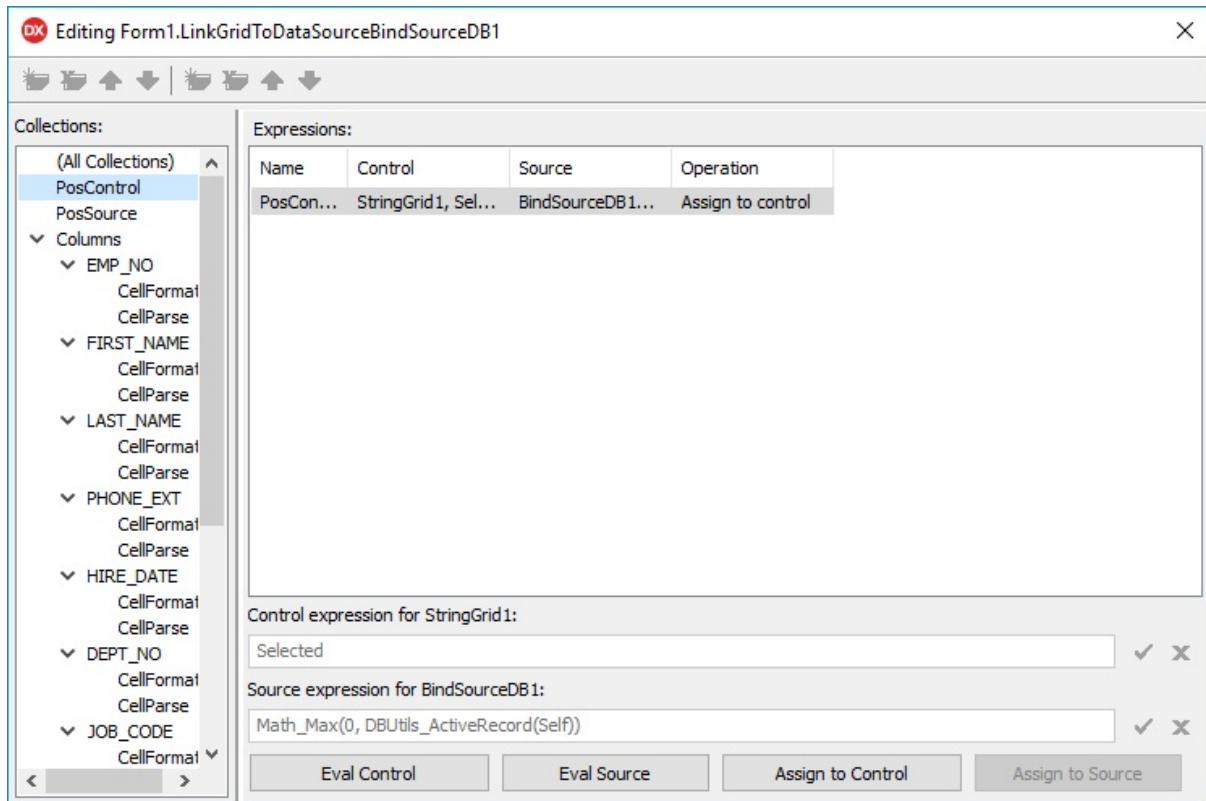
provide for the user's interaction with the control, which in turn affects the dataset.

For example, with a StringGrid connected to a BindSourceDB by way of LiveBindings, when the end user presses Ctrl-End to navigate to the last record displayed in the StringGrid, a PosSource LiveBinding expression (which was automatically created when you bound the StringGrid to the BindSourceDB)

triggers in response to the navigation. This LiveBinding expression is shown in the LiveBindings Expression Editor shown in Figure 4-12. This expression instructs the LiveBinding to set the current record of the dataset to which the BindSourceDB points to either 1, or 1 plus the currently select record in the StringGrid (the StringGrid is zero-based, while the dataset RecNo property is 1-based).

**Figure 4-12: A PosSource LiveBinding expression synchronizes a dataset with the current record of a StringGrid**

PosControl, by comparison, goes the other way. In other words, changes to the current record of the dataset cause the LiveBinding to trigger and assign the



## 96 Delphi in Depth: FireDAC

current record of the StringGrid to match that of the dataset's current record. This can be seen in Figure 4-13.

**Figure 4-13: A PosControl LiveBinding expression synchronizes a StringGrid with a BindSourceDB**

You display the LiveBindings Expression Editor by selecting a LiveBinding in the Object Inspector, and then clicking on the Expressions... link that appears in the lower-left corner of the Object Inspector. However, LiveBindings are not displayed in the Object Inspector in some of the more recent versions of Delphi.

Fortunately, there is an alternate way to display the LiveBindings Expression Editor. You can display a list of all of your current LiveBindings by double-clicking on the BindingsList object created when you define your first LiveBinding, and then double-clicking the LiveBinding of interest from the Bind Components pane of the BindingsList dialog box.

For the remainder of this chapter, I am going to discuss the three main FireDAC

datasets that you are likely to use in your day-to-day applications: FDTable, FDQuery, and FDStoredProc.

## **Understanding FDTable**

FireDAC's FDTable is a dataset component that provides a component-level equivalent of the BDE TTable. While it is simple to use, it is far more capable than the BDE version. Specifically, in addition to the navigating, editing, and searching capabilities provided by the TTable, FDTable also permits filtering, filtered navigation, cached updates, aggregation, and persistence. I will talk about these more advanced features in some of the later chapters of this book.

While FDTable is easy to use, most FireDAC users prefer to use the FDQuery.

FDQueries supports all of the advanced features available through the FDTables, but bases its initial data selection on an SQL query, which is the common query language for most remote database management systems (RDBMS). While it is true that an FDTable also bases its data selection on an SQL query, it is a basic SELECT \* FROM query, whereas with an FDQuery the

SQL is of your design, which means it can include joins, sub-queries, aggregation, or any of the other elements that make SQL such a strong language for data selection and manipulation. Nonetheless, since the FDTable is the simplest of the FireDAC datasets, I will begin by demonstrating its use.

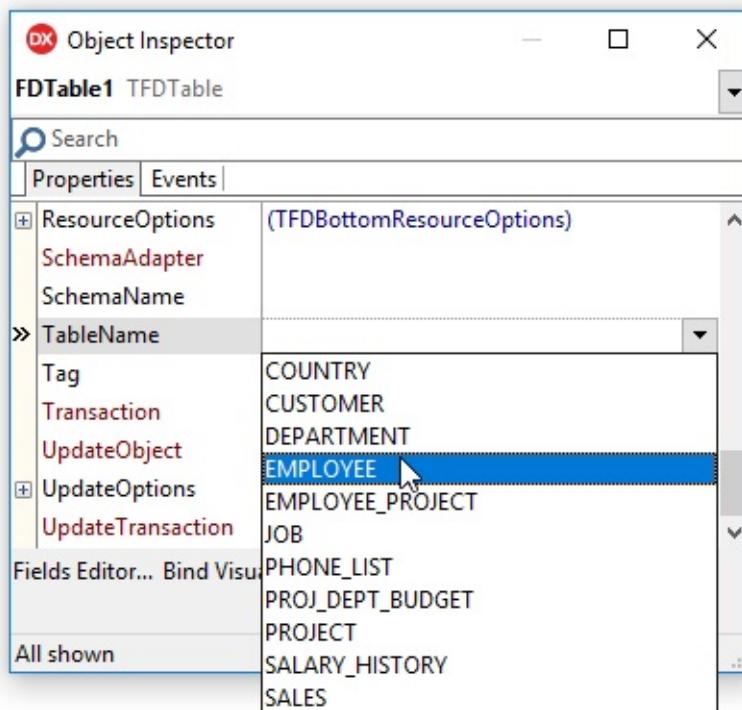
## **Configuring an FDTable**

Configuring an FDTable to retrieve data from a database table could not be easier. All you need is to set the FDTable's Connection property to an active connection to a database, after which you point the FDTable's TableName property to the table whose data you want to work with. This is demonstrated in the following steps. Since these steps are similar to those given earlier in this chapter, I will keep them brief:

1. Select File | New | VCL Forms Application from Delphi's main menu.
2. From the Tool Palette, place a DBNavigator, a DBGrid, and a DataSource onto the form and adjust their placement so that the DBNavigator is at the top and the DBGrid occupies most of the space below it. Also place an FDGUIxLoginDialog on the form as well. (You might also need to place an FDPhysIBDriverLink and

FDGUIxWaitCursor on your form, depending on the version of Delphi you are running.)

3. From the Data Explorer, select the node for the EMPLOYEE database onto the form designer. This will create an FDConnection named EmployeeConnection. If you accidentally drag-dropped the node of the employee table, you will get both an FDConnection named



## 98 Delphi in Depth: FireDAC

EmployeeConnection and an FDQuery named EmployeeTable. If that happened, just remove the FDQuery, leaving only the FDConnection.

4. Select the FDConnection and set its LoginPrompt property to True, and its LoginDialog property to the FDGUIxLoginDialog that you placed in step 2.

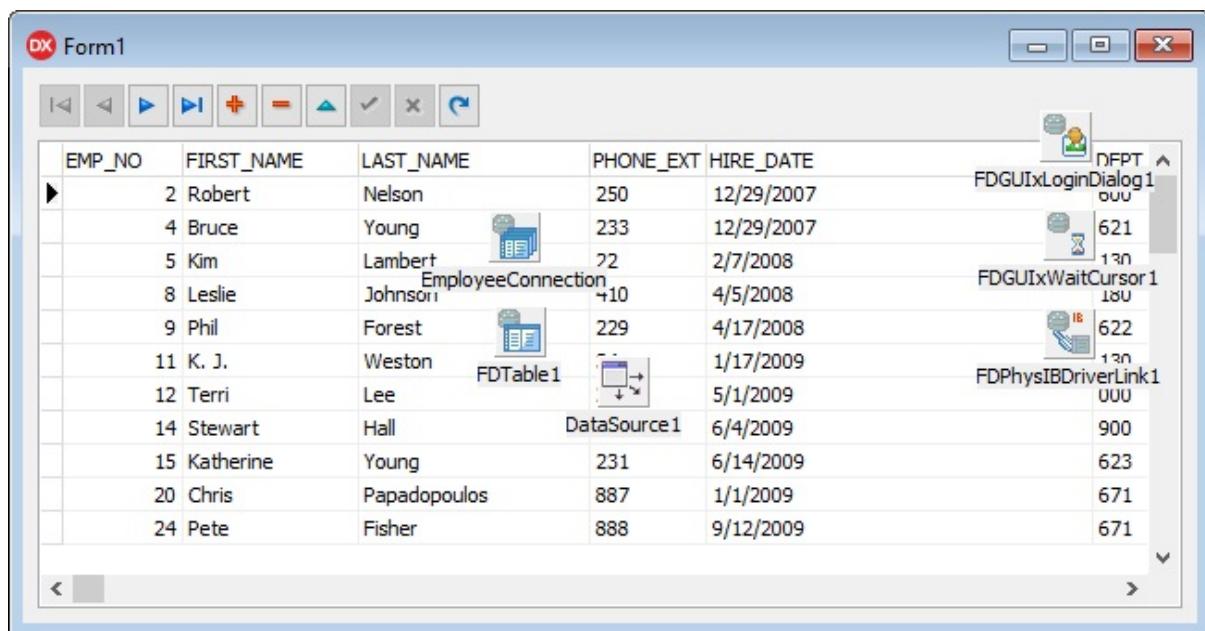
5. From the Tool Palette, drop an FDTable onto the form designer. The FDTable should automatically hook up to the FDConnection. You can confirm this by checking the Connection property of the FDTable in the Object Inspector. (If the Connection property is blank, use the

Connection property drop-down list to set Connection to EmployeeConnection.)

6. Hook up the DBNavigator and the DBGrid to the DataSource, and point the DataSource to the FDTable.

7. Now, with the FDTable selected in the Object Inspector, use the drop-down list on the TableName property to display a list of tables available in the employee.gdb database, as shown in here. If the FDConnection displays a dialog box asking you to login, select OK to continue.

Select Employee.



## Chapter 4: Basic Data Access 99

8. Set the Active property of the FDTable to True. The DBGrid and the DBNavigator should now be active, and your form may look something like that shown in Figure 4-14.

**Figure 4-14: An FDTable is accessing the Employee table from the employee.gdb database**

### Executing Datasets at Design Time

One of Delphi's real strengths, and one that FireDAC datasets share with most other Delphi datasets, and which is not supported by most other database development environments, is the ability to display and configure live data during the design-time process. This feature in Delphi is obvious from Figure 4-14, in which data from the Employee table is displayed within the form designer without having to run the project.

We used an FDTable in this preceding example. However, we could have used an FDQuery, an FDStoredProc, or an FDCommand and we would have achieved the same thing. By setting the Active property of the FireDAC dataset to True, given that the dataset returns a result set of one or more records, that data would be returned, and if we have our dataset hooked up to controls that can display that data, the data is displayed.

Even the FDCommand component, which can execute a wide range of SQL statements, can be executed at design time. If the dataset returns a result set, you

### 100 Delphi in Depth: FireDAC

can set Active to True to execute the query. On the other hand, if you have assigned an SQL statement that creates a table, inserts a record, or deletes records, at design time you can right-click the FDCommand and select Execute.

For those FireDAC datasets that return a result set, if you close the project and open it again, that query will execute again once the dataset has been created.

For example, if you have an FDQuery whose Active property is True, and you have saved that project, the next time you open the form or data module on

which that query appears, by default the query will execute once it is created.

## **Executing DataSets at Runtime**

In practice, you typically do not want your datasets to automatically open, instead opting to have them to open only when you need either the result set they return or the effects they produce, such as creating a new table or removing unwanted records. For those datasets that do not execute automatically, how you make them execute depends on whether or not they return a result set.

For those FireDAC datasets that return a result set, you execute the dataset programmatically by setting Active to True or by calling the Open or Execute method. If you are not sure whether or not a result set is returned, you can call OpenOrExecute.

For FireDAC datasets that do not return a result set, you call a method appropriate for the particular FireDAC component you are using. For example, with a FireDAC FDQuery, you call ExecSQL. When using an FDStoredProc,

you can invoke ExecProc or ExecFunc, depending on whether the

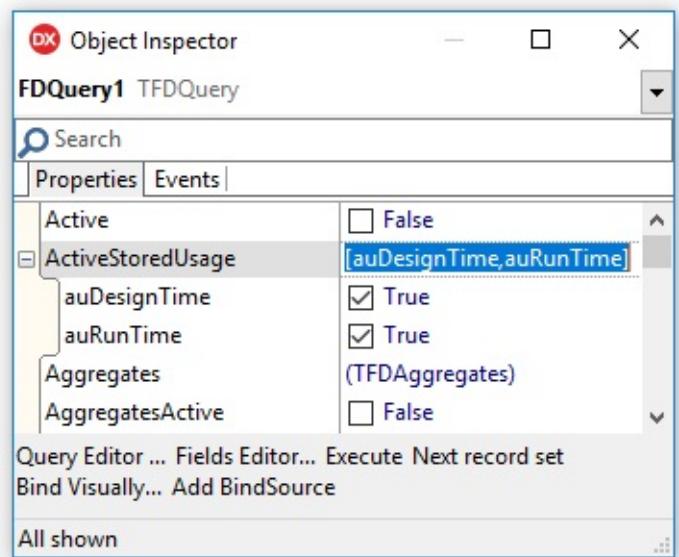
FDStoredProc returns no data or a scalar value, respectively.

## **When Should You Connect?**

For those FireDAC datasets that return a result set, being able to set Active to True at design time is a real advantage. It permits you to configure data-aware controls in a realistic setting, and lets you see your data as you design your forms. On the other hand, in most cases you want to control when your datasets become active at design time. For example, imagine that you have a data

module with 50 FireDAC FDQueries on it. The last thing you want to happen is have all 50 queries execute as part of the data module creation. That might take a long time. It is much better, both resource- and performance-wise, to make your FireDAC datasets active only when you need their data.

Even more to the point, in many cases you will not know what data you want until the user has interacted with your application. For example, you might have set up some dummy parameters in order to test a parameterized query at design time, but it really doesn't make sense to execute the runtime query until you get



## Chapter 4: Basic Data Access 101

the data you want to use for the parameters from the user at runtime. Executing the query automatically using the dummy parameters at runtime is a complete waste of resources.

For Delphi developers using a data access framework other than FireDAC, this means remembering to set Active to False on each of your datasets before compiling your quality control or release build. By doing that, however, you will need to once again set Active to True the next time you are working with your datasets at design time.

Fortunately, FireDAC has a very nice solution. Using the ActiveStoredUsage property of FDConnections and FireDAC datasets, you can define the circumstances under which the Active property is saved. When you expand the ActiveStoredUsage property in the Object Inspector, you can select to restore the value of the Active property either at runtime or at design time.

If you enable the auDesignTime flag, and Active is True, each time the FireDAC component is created at design time, its Active property will be once again set to True. The same rule applies for the auRunTime flag. So, if you want to automatically execute your connection or query at design time, but leave the connection or query closed at runtime (until you are ready to open it), enable the auDesignTime flag but leave the auRuntime flag disabled. It's a beautiful solution to an otherwise nagging problem.

102 Delphi in Depth: FireDAC

### Live Data Window

This discussion started with the FDTable component, and has strayed into issues that affect not only FDTables but other FireDAC datasets as well. I will now return to the FDTable-specific discussion, and this one involves Live Data Window, or LDW, a feature exclusive to FDTables.

LDW is a feature that provides for limited data requests coupled with paging data into and out of memory. The results are typically fast result set return times, limited memory usage, easy navigation through large data tables, and high data concurrency (fresher data).

FireDAC accomplishes LDW by generating SQL queries that limit the number

of records retrieved during any one execution to twice that of the `FetchOptions.RowsetSize`, and includes an appropriate ORDER BY clause in the generated query. The result set returned is bidirectional, and supports all of the typical FireDAC dataset features, including `FindKey`, `FindNearest`, `Locate`, `Lookup`, `Filter`, `IndexFieldNames`, bookmarks, and more.

Behind the scenes, FireDAC is generating SQL SELECT statements that retrieve the Live Data Window data (and provide for other client-side requests, such as filters, sorting, and the like), and for the most part, this is entirely transparent. On the other hand, LDW requires that the underlying database table that the FDTable is referencing in its `TableName` property to either own a

unique index or a unique primary index. In addition, both the server-side collation and client-side collation must be compatible.

There are other requirements, and these are related to the shared instance properties that I described in detail in *Chapter 3 Configuring FireDAC*.

Specifically, `FetchOptions.Items` must include the `fiMeta` flag, `FetchOptions.Unidirectional` must be `False`, `FetchOptions.CursorKind` must be either `ckAutomatic` or `ckDynamic`. In addition, the `FDTable.CachedUpdates` property must be `False`.

When the requirements for LDW are not met, FDTable operates in essentially the same mode as FDQuery, using a `SELECT * FROM` query. And although LDW simplifies data access for developers, it places a larger burden on the database server than does an equivalent FDQuery.

There is one final aspect of LDW to note, and this is related to the `FetchOptions` that apply specifically to LDW. Beginning with Delphi 10.0 Seattle, the

`FetchOptions.LiveWindowParanoic` property is `True` by default, whereas in earlier versions, this property was `False` by default. `LiveWindowParanoic`, when `True`, sacrifices some efficiencies for greater data accuracy. When `False`, some properties, such as `RecNo`, and the `Locate` method, might not behave correctly.

Chapter 4: Basic Data Access 103

## Executing Queries and Stored Procedures

This chapter is about basic data access, and the only thing more basic than using an FDTable is a simple, unadorned call to an SQL (Structured Query Language) query or stored procedure. As a result, I will conclude this chapter with these two operations.

## Executing Queries

You use the FireDAC FDQuery component to execute SQL statements against a

database, be it a database server, such as InterBase or MS SQL Server, or a set of datasets using local SQL. SQL is the standard language for data manipulation and data definition in the world of databases. Sadly, in the world of languages, the word *standard* has come to mean a general rule that nobody follows closely enough to be entirely compatible, but at least enough to be recognizable.

Regardless, in nearly all situations, writing your own SQL statements to execute against your database gives you broad control over how your queries operate, and what they return (when result sets are what you want in return). And,

FDQueries are your primary components for executing custom SQL statements against your data.

Executing an SQL statement against your database with FireDAC is easy.

Assign an SQL statement to the SQL property of an FDQuery and then either Open (or make Active) the FDQuery (if it returns a result set), or execute the query by calling the ExecSQL method. (Alternatively, you can call the Execute method, to which you pass the SQL that you want to execute.) Using the SQL

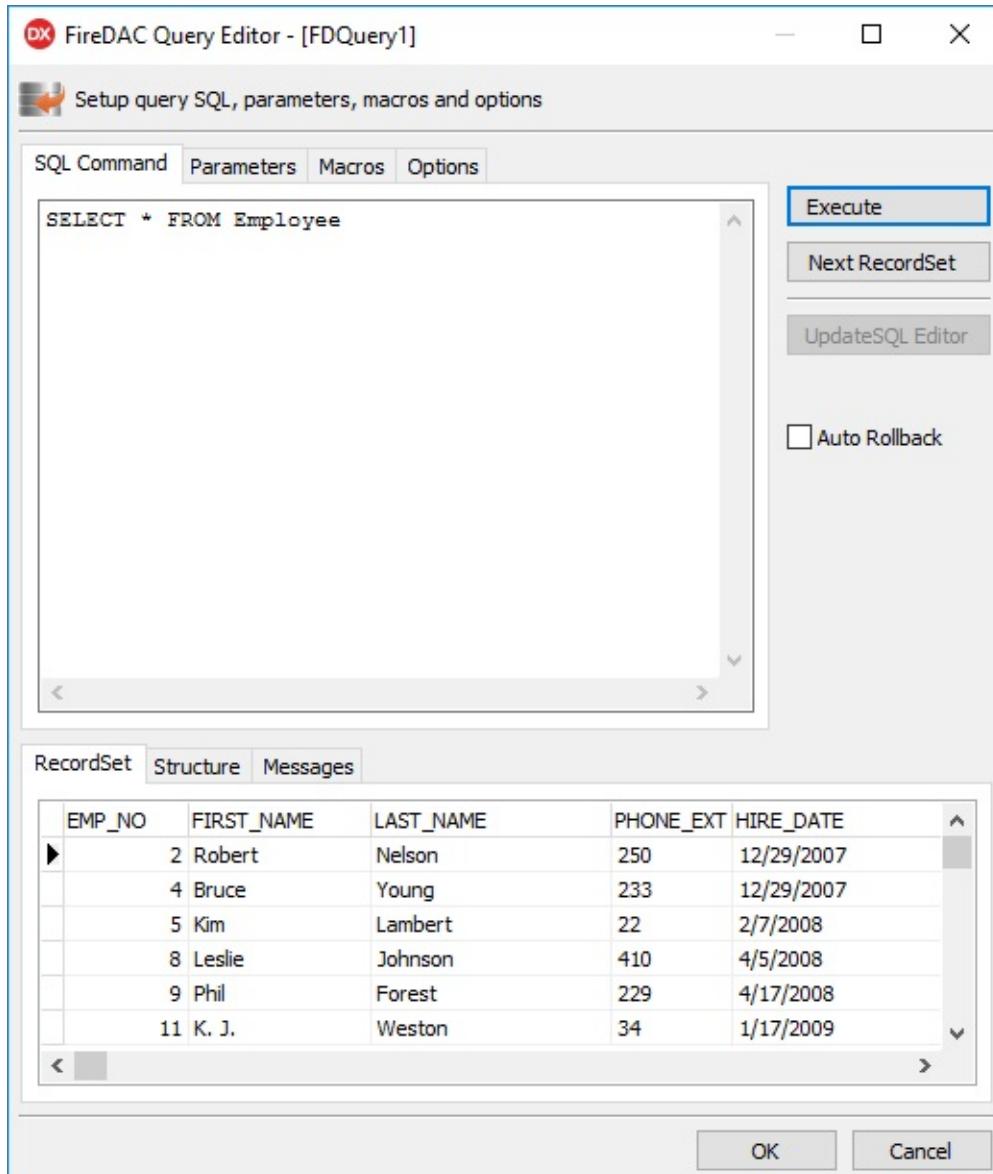
property is demonstrated in the following steps, which build on the steps created earlier to use an FDTable.

Using the project you created earlier in this chapter, perform the following steps:

1. Begin by setting the FDTable component that you placed in your project inactive by setting its Active property to False.
2. Add an FDQuery component to your form. The Connection property of this FDQuery should automatically set itself to the EmployeeConnection FDConnection found on this form. If it does not, set Connection to the

available FDConnection.

3. Double-Click the FDQuery component to display the FireDAC Query Editor. Alternatively, right-click the FDQuery and select Query Editor.



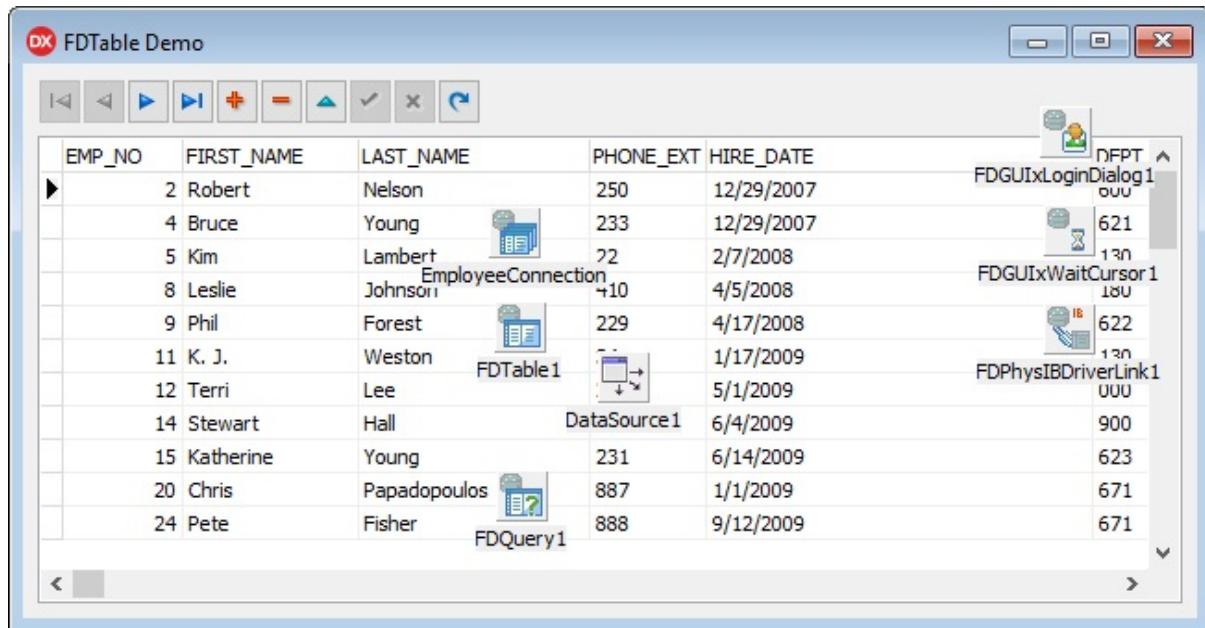
## 104 Delphi in Depth: FireDAC

4. With the SQL Command tab selected, enter the following SQL statement into the provided memo field:

```
SELECT * FROM Employee
```

Click the button labeled Execute, which you will find to the right of the memo field, in order to execute the statement that you entered. Your Query Editor should now look something like that shown in Figure 4-15.

**Figure 4-15: The FireDAC Query Editor**



## Chapter 4: Basic Data Access 105

5. Click OK to save the query to the FDQuery's SQL property and to close the Query Editor.
6. Set the DataSet property of the DataSource to point to your newly added FDQuery.
7. Finally, set the Active property of the FDQuery to True. Your form should now look similar to that shown in Figure 4-16.

### **Figure 4-16: The results of an SQL query is displayed in a DBGrid**

These results do not look any different than those produced by the FDTable, and in this case, the only difference is that LDW is not an option since it is not supported by FDQueries. On the other hand, although we specified an SQL statement that is the same as that which was generated by the FDTable (SELECT \* FROM Employee), we could have provided any valid SQL in order

to get our result, or even provided a DDL (data definition language) statement that doesn't generate a result set at all. In other words, we exerted more control over the query operation with minimal effort. It is for this reason that most Delphi developers prefer using FDQueries over FDTables, in most cases.

### **Executing Stored Procedures**

Stored procedures are subroutines defined in your database. While almost every remote database management system (RDBMS) supports stored procedures,

#### **106 Delphi in Depth: FireDAC**

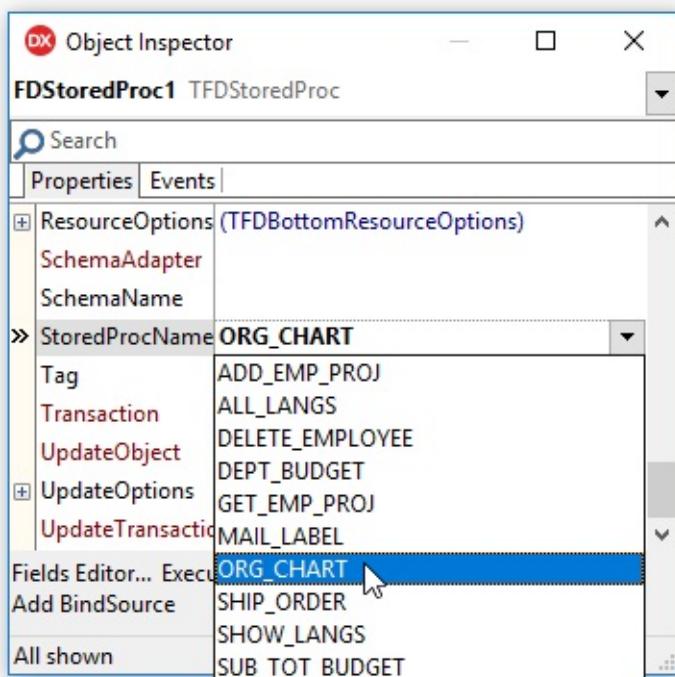
some databases supported by FireDAC, such as MS Access and Paradox (file server databases), do not. Nonetheless, stored procedures are an important entity for most database applications, permitting you to encapsulate database operations in a procedural call that can be invoke by a database client.

A stored procedure can return a result set, a scalar value, or simply perform a server-side operation, depending on how it is designed. Regardless, there are two ways to invoke a stored procedure on your database server. One way is to use an FDQuery, and execute a query that uses a syntax similar to the

following: EXECUTE PROCEDURE *StoredProcName*(  
parameters\_if\_required )

In that case, you use an FDQuery and the SQL statement will be a stored procedure invocation similar to the one shown above (although if the stored procedure does not return a result set or value, you invoke ExecSQL or OpenOrExecute instead of setting Active to True). The other option is to use an FDStoredProc component, as shown in the following steps:

1. Using the existing Delphi project that you created earlier in this chapter, and which you modified in the preceding section, let's modify it further to use a stored procedure.
2. Begin by setting the FDQuery's Active property to False.
3. Next, add an FDStoredProc component to the main form of this application. It should automatically set its Connection property to the EmployeeConnection FDConnection. If it does not, set the stored procedure's Connection property to the FDConnection.
4. With the FDStoredProc selected, use the drop-down list on the StoredProcName property to display a list of defined stored procedures available in the employee.gdb database, as shown in here.



## Chapter 4: Basic Data Access 107

5. Select the stored procedure named ORG\_CHART.
6. Set the Active property of the FDStoredProc component to True.
7. Finally, set the DataSet property of the DataSource component on the form to FDStoredProc1. Your form should now look something like that shown in Figure 4-17.

HEAD_DEPT	DEPARTMENT	MNGR_NAME	TITLE	EMP_CNT
	Corporate Headquarters	Bender, Oliver H.	CEO	
Corporate Headquarters	Sales and Marketing	MacDonald, Mary S.	VP	2
Sales and Marketing	Pacific Rim Headquarters	Baldwin, Janet	Sales	7
Pacific Rim Headquarters	Field Office: Japan	Yamamoto, Takashi	SRep	2
Pacific Rim Headquarters	Field Office: Singapore	--TBH--		0
Sales and Marketing	European Headquarters	Reeves, Roger	Sales	2
European Headquarters	Field Office: Switzerland	Dosborne, Pierre	SRep	1
European Headquarters	Field Office: France	DataSource1, Jacques	SRep	1
European Headquarters	Field Office: Italy	Ferrari, Roberto	SRep	1
Sales and Marketing	Field Office: East Coast	Montgomery, K. J.	SRep	2
Sales and Marketing	Field Office: Canada	Hardy, Claudia	SRep	1
Sales and Marketing	Marketing	FDStoredProc1		2

108 Delphi in Depth: FireDAC

**Figure 4-17: A stored procedure call returns data from the ORG\_CHART**

**stored procedure from the EMPLOYEE database**

In the next chapter, I will explore data access in more detail.

Chapter 5: More Data Access 109

# Chapter 5

## More Data Access

In the preceding chapter, you learned how to access your database tables using FDTables, FDQueries, and FDStoredProc components. This chapter extends this discussion with a look at parameterized queries and stored procedures. I

continue this discussion of working with data by looking at several additional classes that you may want to employ in your database applications. These include the FDUpdateSQL component, which simplifies the process of customizing updates to database tables, FDCommands, which are used within other FireDAC datasets, and FDTransactions, which you can use to ensure that your updates are completed in an all-or-none manner. I also talk about asynchronous versus synchronous query execution, as well as the FireDAC Monitor utility.

Let's begin with parameterized queries and stored procedures.

### Parameterized Queries and Stored Procedures

Parameters are placeholders used in the predicates that appear in the WHERE clause of SELECT and DELETE queries, the SET clause of UPDATE queries,

and the VALUES part of INSERT queries. For example, the following is an SQL SELECT statement with a single predicate in the WHERE clause. In this case, the right-hand side of the predicate is a literal value:

```
SELECT FirstName, LastName FROM EMPLOYEE WHERE Emp_No = 2;
```

By comparison, the following statement is a parameterized query, where the literal value has been replaced by a parameter:

```
SELECT FirstName, LastName FROM EMPLOYEE WHERE Emp_No = :en;
```

In the first example, the query will always return data associated with records where the Emp\_No field contains the value 2. Assuming that a valid value has been assigned to the parameter in the second query, this query will return data

## 110 Delphi in Depth: FireDAC

associated with any record whose Emp\_No field matches the value assigned to the parameter. The first example is static, while the second is dynamic.

### The Advantages of Parameters

Parameters serve a number of purposes. These include permitting you to write flexible queries that can be customized prior to their execution, improving performance when many different yet similar queries need to be executed, and preventing the injection of potentially damaging SQL statements into the queries that you execute.

Before continuing, it is worth considering each of these advantages in greater depth.

#### **GREATER FLEXIBILITY**

Simply by introducing a single parameter, you turn a static query, one that returns a similar result each time it is executed, into a flexible query whose result sets can be changed merely by changing the value of a parameter. You might notice that I did not say that static queries always return the same result sets, because in most cases they don't. The data they return is based on the data contained in the database, and most databases change day by day, if not second by second. As a result, the same static query is not expected to return the same result set each time.

However, static queries do return the same information, even though the actual values may change. By comparison, a parameterized query may return very

different data, depending on the values of the parameters employed.

A simple example makes this obvious. Imagine a query that returns detailed information about a particular client's purchases. Assuming that the parameter defines which client the data pertains to, not only do the results change, but the interpretation changes as well. A static query may return information about one client's purchases, and those change as the client makes additional purchases, but changing the client means that entirely different data is being returned.

When you consider that a parameterized query may include many different parameters, it is easy to see why parameterized queries provide flexibility well beyond that afforded by static queries.

#### **IMPROVED PERFORMANCE**

Most database servers perform an analysis of a query prior to its execution,

creating an execution plan to optimize what it's being asked to do. When a query is parameterized, this preparation needs to be performed only once, prior to the first time the query is executed. So long as that query has not been unprepared,

Chapter 5: More Data Access 111

the values of the parameters can be changed and the query can be executed again without requiring a new execution plan.

The time saved by reusing the execution plan is significant, especially if the query is one that is being executed hundreds, or even thousands of times, which is what often happens during large reporting operations. Any operation where a parameterized query is executed repeatedly, albeit with different values in the parameters, produces superior performance to the sequential execution of

different static queries.

## **PREVENTION OF SQL INJECTION**

When accepting input from the user interface for use in a predicate of an SQL statement, it is essential that you employ a parameter to hold that data, rather than concatenating the user input into a string that you execute. Most database servers permit SQL queries to include two or more individual SQL statements separated by a special character, most often a semicolon. If you construct a query string at runtime by concatenating literal SQL statements with data input by a user, there is a possibility that a user with a knowledge of SQL could exploit this feature to undermine your database. This is called *SQL injection*.

Here's an example. Imagine that your client application includes a query that is constructed at runtime based on the user's entry of a Customer ID.

Consider the following code, which wrongly builds an SQL statement based on the customer ID that the user enters into an Edit in your user interface:

```
//Construct the query
FDQuery1.SQL.Text := 'SELECT * FROM CUSTOMER ' +
'WHERE [CustomerID] = ' + Edit1.Text;
//Execute the query
FDQuery1.Open;
```

Here, the value entered into the Edit is concatenated to the query being assigned to the SQL property of the FDQuery. Now consider what would

happen if the user enters the following data into the edit:

```
1001;DROP TABLE CUSTOMER;//
```

Assuming that the // characters are comment identifiers, making everything that follows them in the SQL statement appear to be a comment, the resulting query

112 Delphi in Depth: FireDAC

would actually be an SQL script (two SQL statements), and would look like the following:

```
SELECT * FROM CUSTOMER  
WHERE [Last Name] = 1001;DROP TABLE CUSTOMER;//
```

Obviously, if you were to execute this SQL script, it would seriously compromise your database.

The easiest way to prevent SQL injection is to use parameterized queries, where you bind the user's input to query parameters. Because of the way parameterized queries are executed, FireDAC will never confuse a semicolon in a parameter as a statement separator.

The following code produces the same intention as the preceding code. This time, however, the user's input is employed using a parameterized query:

```
//Construct the query  
FDQuery1.SQL.Text := 'SELECT * FROM CUSTOMER ' +  
'WHERE [CustomerID] = :ID';  
  
// Get the user input  
FDQuery1.ParamByName('ID').Value := Edit1.Text;  
  
//Execute the query  
FDQuery1.Open;
```

In this case, if the user typed 1001;DROP TABLE CUSTOMER;// into the Edit, the resulting query would actually try to select a customer whose CustomerID is 1001; DROP TABLE CUSTOMER;//, and such a query would likely produce a

null result set. More importantly, it would cause no damage to any of the database tables.

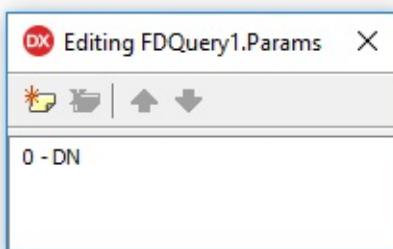
## Defining Parameters at Design Time

Both the FDQuery and FDStoredProc components have a Params property, and

these properties are of the type TFDParams, a collection of TFDParam items. So long as the ParamCreate flag is set in the ResourceOptions property of the associated dataset (the default), FireDAC will create one TFDParam instance for each parameter defined in the SQL property of the FDQuery component, as well as one for each corresponding parameter in the stored procedure whose name has been assigned the StoredProcName property of an FDStoredProc

component.

For those parameters that are input parameters, you can select the property and assign data to its Value property. For example, consider the following query:



DX Object Inspector

FDQuery1.Params[0] TFDParam

Properties Events

Search

ArraySize	1
ArrayType	atScalar
DataType	<b>ftWideString</b>
DataTypeName	
FDDataType	dtUnknown
IsCaseSensitive	<input type="checkbox"/> False
Name	<b>DN</b>
NumericScale	0
ParamType	<b>ptInput</b>
Position	0
Precision	0
Size	0
StreamMode	smOpenRead
Value	623
Type	String

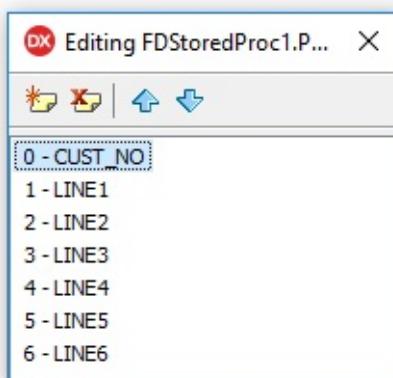
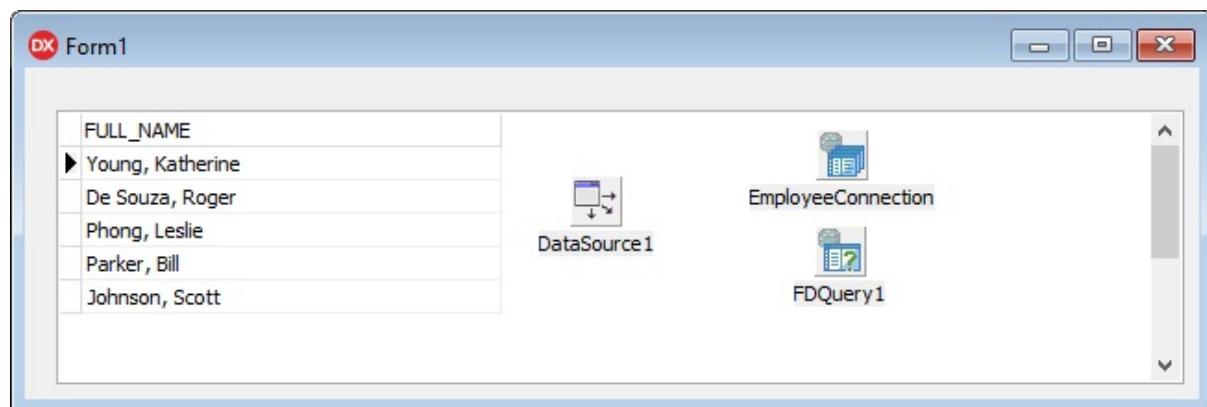
All shown

## Chapter 5: More Data Access 113

SELECT Full\_Name FROM Employee WHERE DEPT\_NO = :dn

After entering this query into the SQL property of an FDQuery, if you open the Params collection editor from the Object Inspector you will see the named parameter, as shown here:

Select the parameter named DN, and the Object Inspector displays the properties of the TFDParam. Enter 623 into Value to define that you want to restrict the selection to employees from department 623, as shown here:



### 114 Delphi in Depth: FireDAC

If you now set the Active property of this query to True, and your FDQuery is connected to a DataSource which in turn is connected to a DBGrid, you will get a result set that looks similar to that shown in Figure 5-1.

**Figure 5-1: The Parameter of a parameterized query has been set at design time, and the query has been executed**

If you are working with a parameterized stored procedure, the process is very similar. Consider the Params collection shown in the following illustration,

which is displayed if you set an FDStoredProc component's StoredProcName property to Mailing\_Label in the employee.gdb database. Here you see seven parameters. The first parameter, CUST\_NO, is an input parameter, and you can assign a value to it at design time. The remaining six parameters are output parameters, and they will get values only after you execute this stored procedure with a valid value assigned to the CUST\_NO named parameter (and these

values might be null, depending on the value of the input parameter).

Chapter 5: More Data Access 115

### **Parameterized FDQueries and the Query Editor**

The FireDAC Query Editor, which we encountered in the preceding chapter, not only provides you with a platform for testing queries, but it can also be used to test and assign parameters to your FDQuery.

Let's start with an SQL statement that is a little more involved. This SQL statement includes two parameters as well a sub-query and an inner join:

SELECT

```
(SELECT e.Full_Name FROM Employee e  
WHERE e.Emp_No = s.SALES REP) as SalesRep,  
c.Contact_FIRST || ' ' || c.Contact_Last as Customer,  
s.Order_Date,  
s.Qty_Ordered as Quantity,  
s.Total_Value as Total
```

FROM SALES s

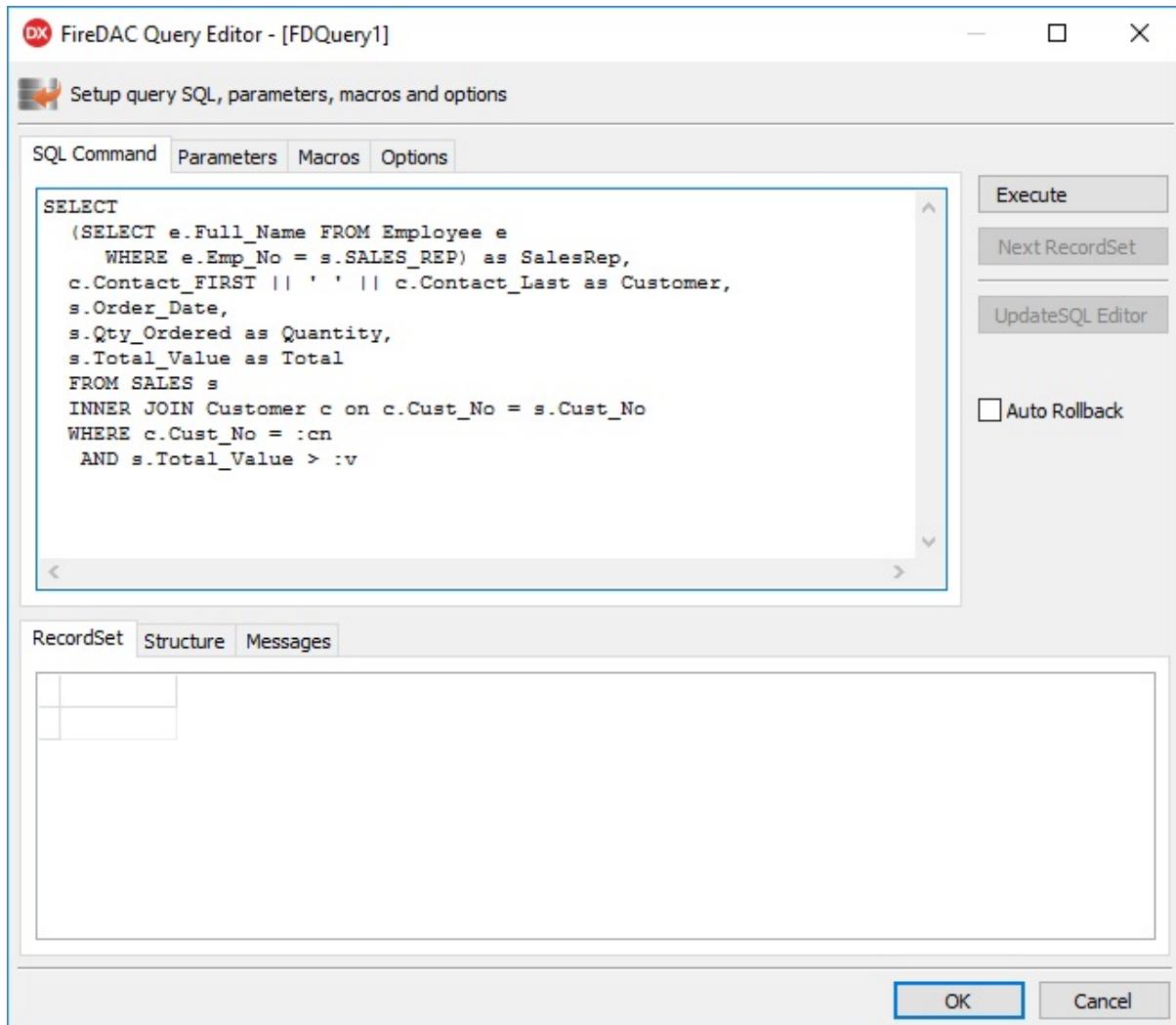
INNER JOIN Customer c on c.Cust\_No = s.Cust\_No

WHERE c.Cust\_No = :cn

AND s.Total\_Value > :v;

Assuming that we have an FDConnection configured to use the employee.gdb database, and an FDQuery, we can double-click the FDQuery to display the FireDAC Query Editor. Enter this query into the SQL Command pane FireDAC

Query Editor, as shown in Figure 5-2.



## 116 Delphi in Depth: FireDAC

**Figure 5-2: A query with two parameters appears in the FireDAC Query Editor**

Since the `ResourceOptions.CreateParams` property is set to True by default, FireDAC examines the query and creates one `FDPParam` instance for each of the parameters that it finds in the query. This can be seen in the FireDAC Query Editor by clicking on the Parameters tab, as shown in Figure 5-3.

## FireDAC Query Editor - [FDQuery1]

Setup query SQL, parameters, macros and options

SQL Command Parameters Macros Options

CN
V

Param type: ptUnknown  
Data type: ftUnknown  
Data size: 0  
Value: <null>  
Array type: atScalar  
Array size: 1  
Position: 0

Execute  
Next RecordSet  
UpdateSQL Editor

Auto Rollback

RecordSet Structure Messages

--	--

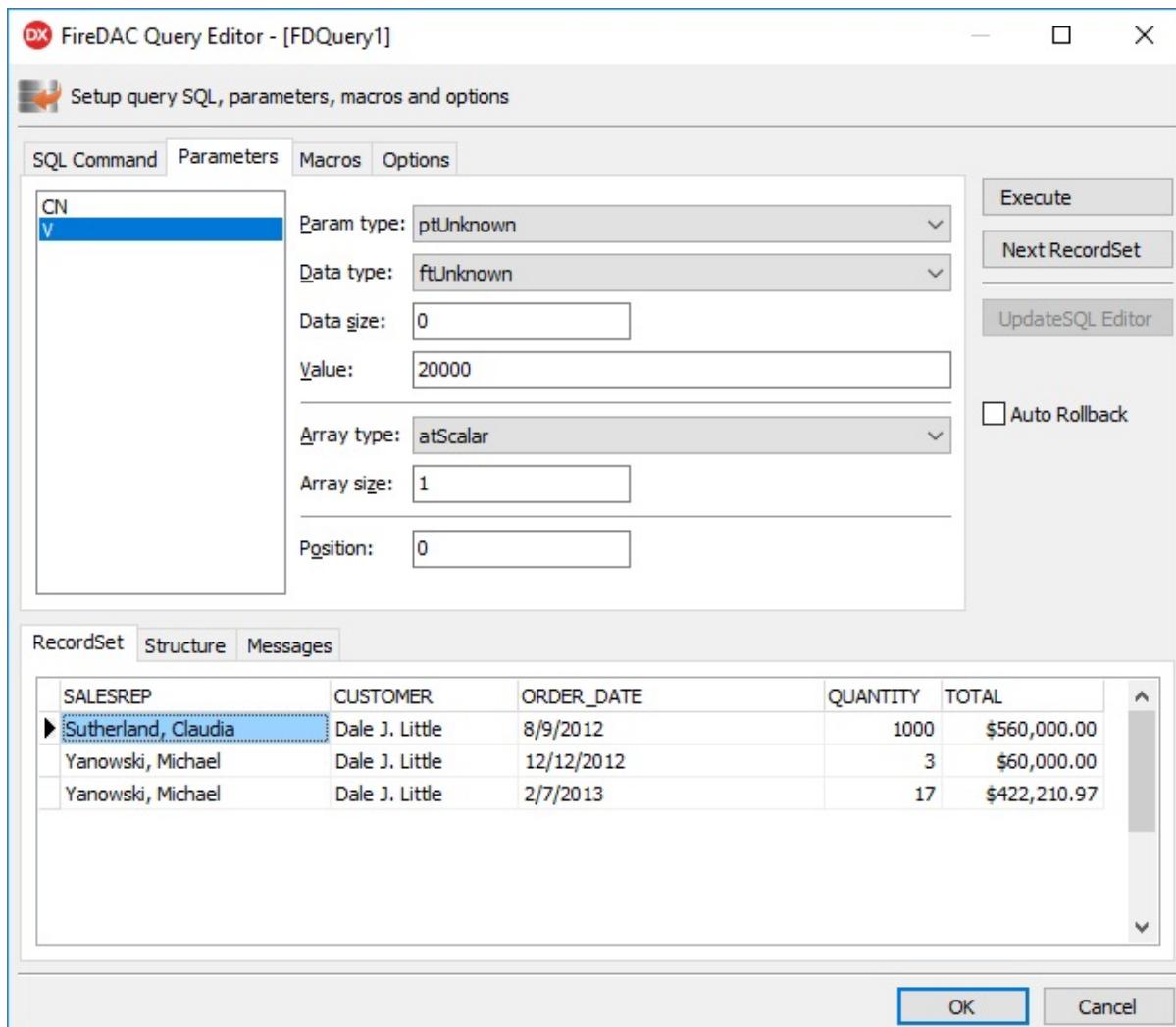
OK

Cancel

## Chapter 5: More Data Access 117

**Figure 5-3: FireDAC has identified the named parameters from the query, and permits you to configure them in the FireDAC Query Editor**

In most cases, it is not necessary to provide the Param type and Data type as the Value property of the TFDParam is a variant, but you can provide that information if you want, or under those conditions where FireDAC cannot correctly resolve the type of your parameters. In this case, it is sufficient to simply provide a value for each parameter. Enter 1001 in the Value field for the CN parameter, and 20000 in the Value field for the V parameter. If you then click Execute, you will get the result set you see in Figure 5-4.



118 Delphi in Depth: FireDAC

**Figure 5-4: The FireDAC Query Editor has bound the two parameters to,**

## **and executed, the query**

If you now click the OK button, the FireDAC Query Editor will not only save the SQL statement to the FDQuery's SQL property, but it will also save the values of the parameters to the FDQuery's Params collection.

## **Defining Parameters at Runtime**

Defining parameters at design time permits you to test your queries and confirm that your parameters are valid. However, it is almost universally the case that you will bind values to your parameters at runtime prior to executing your queries or stored procedures. Otherwise, a static query would work just as well.

Chapter 5: More Data Access 119

Whether you are using an FDQuery or an FDStoredProc, the process is the same. With the dataset inactive, and prior to executing the dataset, you assign a value to each of the defined input parameters.

There are two techniques that you can use when your parameters are named parameters: You can assign the parameters based on position or by name.

To assign the parameters by position, you use the indexed Params property of the FDQuery or FDStoredProc dataset. The position of a named parameter is a zero-based position based on the order in which the parameter names appear in the SQL query or the stored procedure parameter list.

Let's consider once again the more involved query used to demonstrate parameter binding using the FireDAC Query Editor. I am repeating this query here for convenience:

SELECT

```
(SELECT e.Full_Name FROM Employee e  
WHERE e.Emp_No = s.SALES REP) as SalesRep,  
c.Contact_FIRST || ' ' || c.Contact_Last as Customer,  
s.Order_Date,  
s.Qty_Ordered as Quantity,  
s.Total_Value as Total  
FROM SALES s  
INNER JOIN Customer c on c.Cust_No = s.Cust_No  
WHERE c.Cust_No = :cn
```

AND s.Total\_Value > :v

There are two parameters here, and they are named parameters. To assign the parameters by name, you can use the ParamByName method, to which you pass

the name of the parameter. This method returns an FDParam instance, and you use it to assign a value to the parameter. FDParam instances have a variety of properties such as AsString, AsInteger, and so forth (similar to Fields) to which you can assign the value of the parameters. However, the Value property, which is of type variant, is almost always sufficient to handle parameter assignment.

The following is a code segment that demonstrates how to assign data to the named parameters:

```
// FDQuery1 already has the SQL statement assigned to its SQL
```

```
property
```

```
FDQuery1.ParamByName('v').Value := 20000;
```

```
FDQuery1.ParamByName('cn').Value := 1001;
```

```
FDQuery1.Open;
```

## 120 Delphi in Depth: FireDAC

There are a couple of comments that I have about this code. First of all, the parameter names are case insensitive, so it doesn't matter if you reference them using Cn, cN, CN, or cn. It's all the same to FireDAC. Also, it does not matter in which order you assign values to the parameters. The only thing that counts is that each parameter has a value before you attempt to execute the query.

As mentioned, even when you use named parameters you can assign parameter

values to your FireDAC datasets by position. In short, the position of a parameter for named parameters is the order in which the parameter names appear in the query. In the preceding SQL statement, the first parameter (position 0) is associated with the name cn, and the second parameter (position 1) is associated with the v parameter. As a result, the following code performs the same parameter assignment as that shown in the preceding code sample:

```
// FDQuery1 already has the SQL statement assigned to its SQL
```

property

```
FDQuery1.Params[1].Value := 20000;  
FDQuery1.Params[0].Value := 1001;  
FDQuery1.Open;
```

Notice that I didn't assign values to the parameters in order. I did that on purpose just to emphasize that order is not relevant, just so long as each parameter has a value prior to your opening the dataset.

Since positional parameters do not have names, but instead use some sort of marker, most often the ? character, you can only assign these parameters based on position. For the same reason, you must always provide one value for each positional parameter marker that appears in a query, while when using named parameters you assign values by position based on the position of the first instance of a given name in the query. For example, consider the following query, which uses positional parameters:

```
SELECT  
(SELECT e.Full_Name FROM Employee e  
WHERE e.Emp_No = s.SALES REP) as SalesRep,  
(SELECT c.Contact_FIRST || ' ' || c.Contact_Last  
FROM Customer c WHERE c.Cust_No = ?) as Customer,  
s.Order_Date,  
s.Qty_Ordered as Quantity,  
s.Total_Value as Total  
FROM SALES s  
WHERE c.Cust_No = ?  
Chapter 5: More Data Access 121  
AND s.Total_Value > ?
```

This query requires three parameters, even though the first two parameters are likely to be the same value (the same customer number). Granted, I could have written this query in a way that would have required only two parameters, but the point is valid. When using positional parameters, you must provide a value for each positional parameter marker.

So, in this case, the parameter value assignment and query execution would look something like the following:

```
// FDQuery1 already has the SQL statement  
// assigned to its SQL property  
FDQuery1.Params[0].Value := 1001;  
FDQuery1.Params[1].Value := 1001;  
FDQuery1.Params[2].Value := 20000;  
FDQuery1.Open;
```

Assigning parameters to datasets that do not return a result set uses the same approach as shown here. The only difference is that you will not use the Active property or the Open method to execute the dataset. Instead, you will call the appropriate Exec... method. For queries, this is ExecSQL or OpenOrExecute, and for stored procedures, you will invoke ExecProc, ExecFunc, or

OpenOrExecute.

### **Taking Control of Updates: FDUpdateSQL**

As you learned in the preceding chapter, in many cases you can edit the contents of a FireDAC query, and post those changes to the underlying database as if the FDQuery were an FDTable. This is true even when the query includes joins,

though in most cases, it is the base table, the principle table of the query, and not the joined tables, to which data can be written.

Even though FireDAC is exceptionally clever when it comes to figuring out how to write changes back to the underlying database, some queries are just too complicated or convoluted for FireDAC to handle. In those cases, you have

three choices.

One option is to write your own queries, often parameterized ones, and populate those parameters based on the changed data upon post, delete, or insert, after which you execute the query. The second option is to add an OnUpdateRecord

### 122 Delphi in Depth: FireDAC

event handler to the query, and basically do the same thing as in the first option, though with some additional help. (OnUpdateRecord is discussed in detail in *Chapter 16, Using Cached Updates.*) The third option is to use FireDAC's FDUpdateSQL component.

### **Defining the Query**

Here is a rather simple example, but it demonstrates the value of FDUpdateSQL.

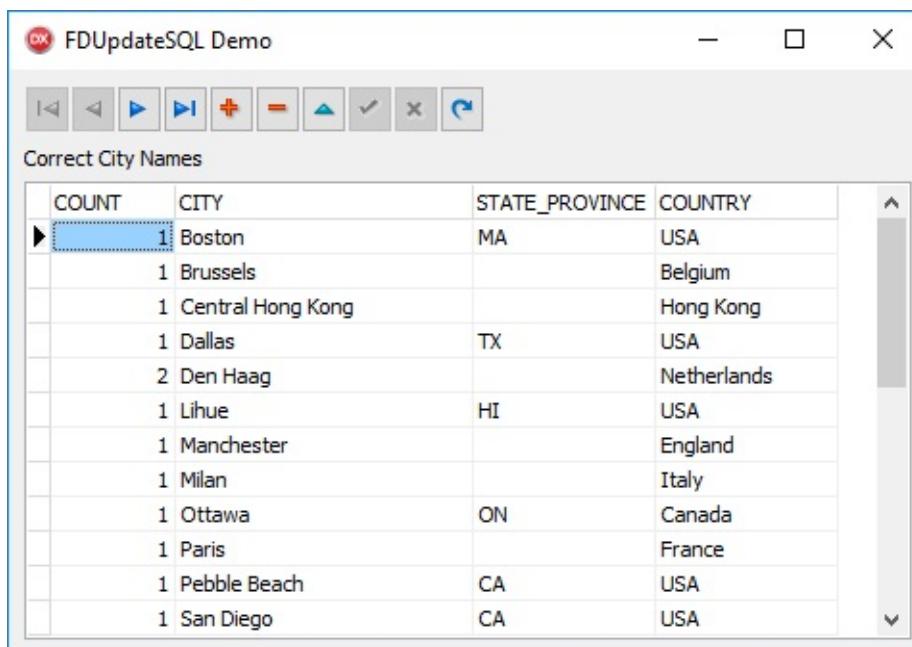
Let's begin with a query that is clearly not updateable:

```
SELECT Count(*), City, State_Province, Country  
FROM Customer  
GROUP BY City, State_Province, Country
```

Here's the story. Imagine that you manage a shipping company, and you need to review shipping orders at the end of each day to ensure that the orders are accurate. Since in the past you have found that most common error is the input of the city name (both State\_Province and Country are validated by a table, but City is not), you need to fix misspelled city names, regardless of how many times an invalid city name was entered.

This requirement, to update a field, in possibly more than one record, from a query that uses aggregation and grouping, violates a number of the restrictions imposed by FireDAC and the default values of the UpdateOptions property.

Fortunately, the FDUpdateSQL component permits you to overcome these restrictions, and this is demonstrated in the FDUpdateSQLDemo project, whose main form is shown in Figure 5-5.



# Chapter 5: More Data Access 123

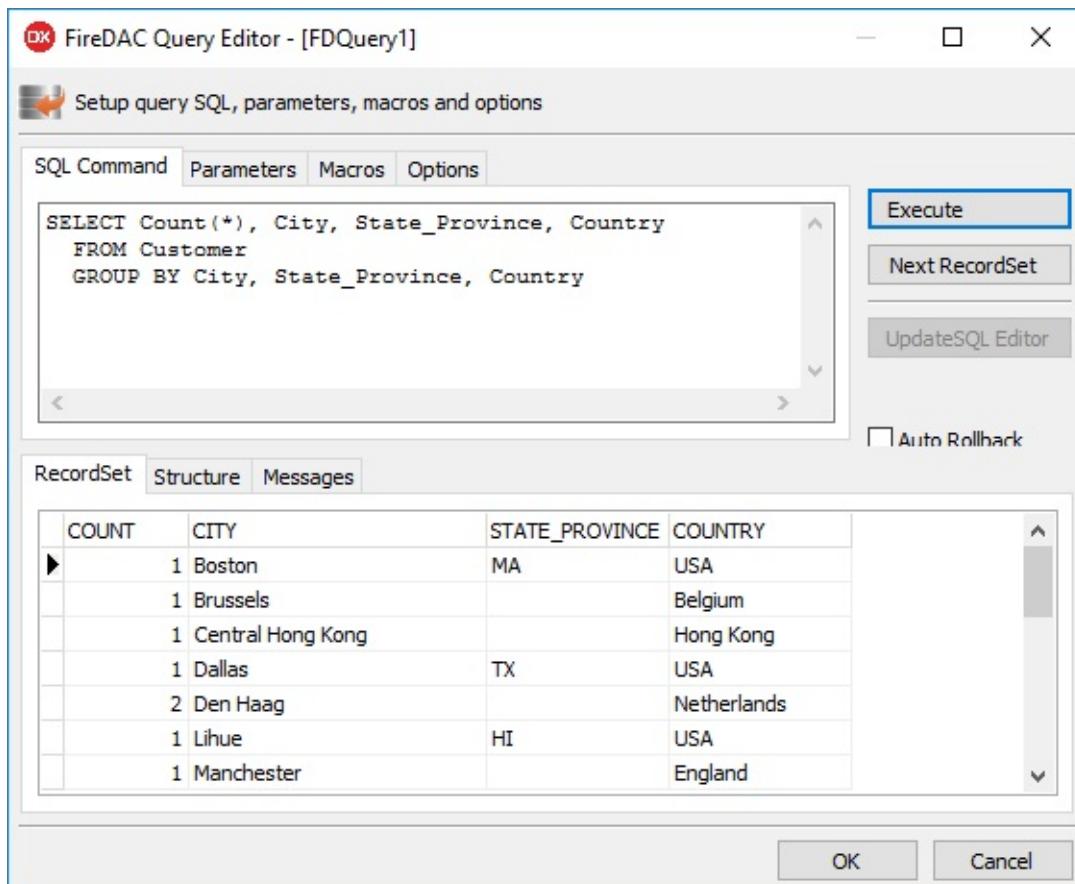
**Figure 5-5: The main form of the FDUpdateSQLDemo project**

*Code: You can find the FDUpdateSQLDemo project in the code download. See Appendix A for details.*

*Note: There were no duplicate cities in the Customer table of the original employee.gdb database. Because I felt it important to demonstrate the use of the UpdateOptions.CheckUpdatedRecords property, and the ability to project an update onto two or more records, I added an extra record to the database — a customer whose city and country fields held the values Den Haag and Netherlands, respectively.*

To begin with, you need to configure your FDQuery with both the query, as well as the UpdateOptions that will permit you to make changes. Specifically, once you have the preceding query assigned to the FDQuery, its default options

would otherwise prevent any editing, since FireDAC cannot figure out what you are trying to accomplish:



1. Using the FireDAC Query Editor, enter the preceding query into the SQL Command field of the query editor.
2. Click Execute to verify that you have entered the query correctly and FireDAC can retrieve your aggregate query. Your FireDAC Query Editor should look like that shown in Figure 5-6.

**Figure 5-6: A readonly query has been executed using the FireDAC Query Editor**

### Editor

### Configuring the DataSet

You can now edit the UpdateOptions using either the FireDAC Query Editor or the Object Inspector. Since we have the FireDAC Query Editor already open, let's use it:

### Chapter 5: More Data Access 125

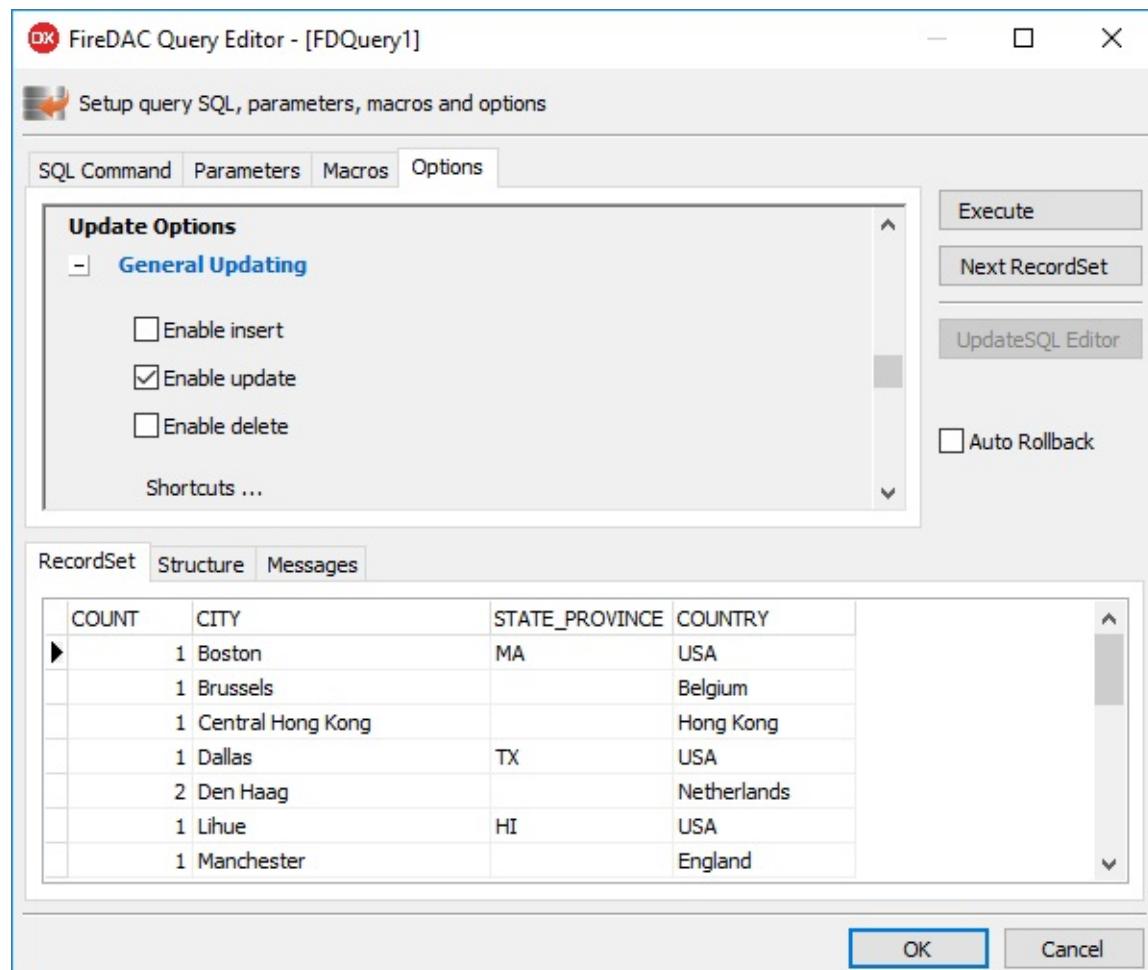
1. Click on the Options tab of the FireDAC Query Editor and scroll to the UpdateOptions section.
2. Expand the General Updating group. Since we only need to update the City field of the query result, uncheck the Enable insert and Enable delete checkboxes. Leave Enable update checked.
3. Next, expand the Posting Changes group. Leave Update mode set to upWhereKeyOnly. However, we do not want FireDAC to enforce the readonly fields flag or to check update records count. This query is a readonly query, and as a result, all fields will be seen internally as readonly. By unchecking the Check “readonly” field flag, we are telling FireDAC to ignore this information and let us make the changes we want.
4. Normally, FireDAC will raise an exception if an update affects more or less than one record. In this case, though, we want to correct all misspelled city names, regardless of how many times the misspelling occurred. To do this, uncheck Check updated record count.

*Tip: When I first created this example, the Check updated records count option was checked and disabled on the Options page of the FireDAC Query Editor. As a result, I could not control this property from the Options page of the FireDAC*

*Query Editor. Instead, I had to select the FDQuery in the Object Inspector and expand its UpdateOptions property. From there I was able to set*

*CheckUpdatedRecords to False. If you find that there are properties that you cannot set from the FireDAC Query Editor, you should try using the Object Inspector.*

We are through configuring our FDQuery. The Options page of the FireDAC Query Editor should now look similar to that shown in Figure 5-7.

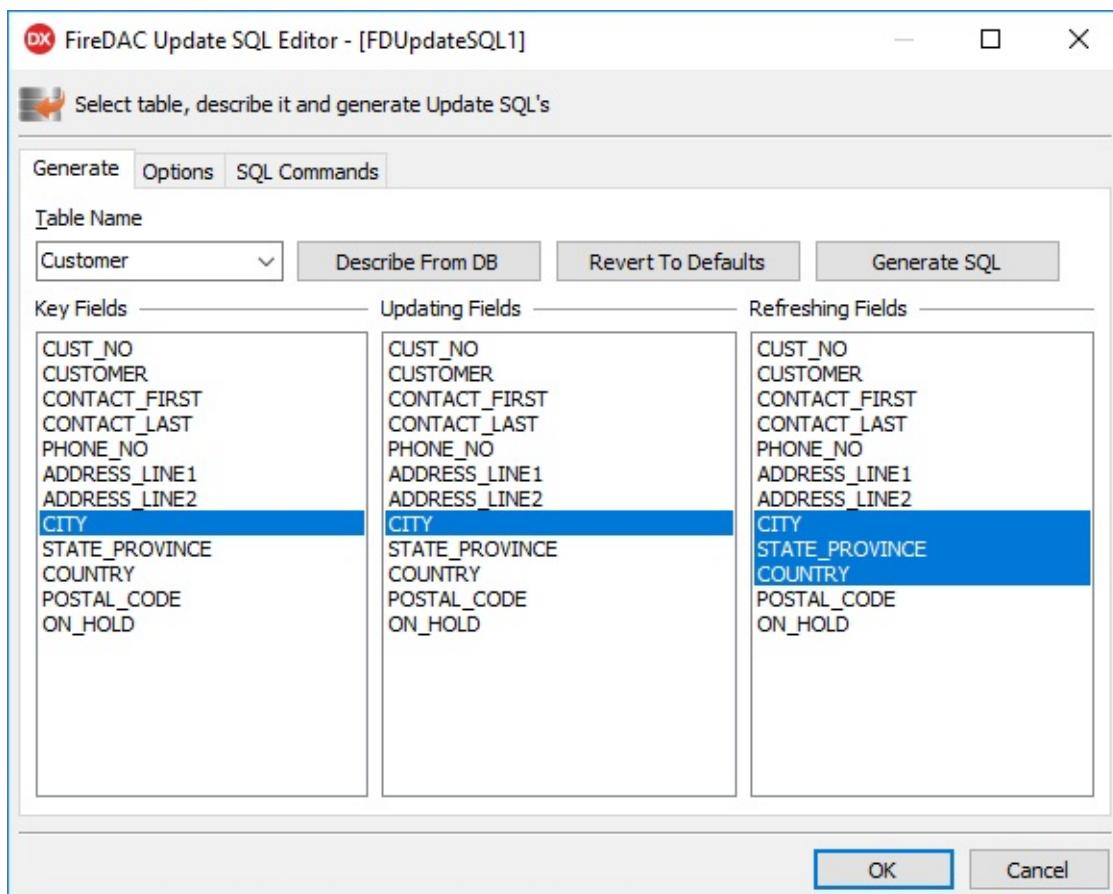


126 Delphi in Depth: FireDAC

### Figure 5-7: The Update Options of the query have been modified

5. It's now time to add our FDUpdateSQL component from the Tool Palette onto our form.

6. Once the FDUpdateSQL component is in place, select the FDQuery and set its UpdateObject property to point to the FDUpdateSQL component.
7. Next, set the FDQuery's Active property to True, since the FDUpdateSQL will need access to the metadata that the FDQuery will collect. If you do not do this in advance, the FDUpdateSQL component will prompt you to execute the query, since it needs this metadata.
8. Finally double-click the FDUpdateSQL component to display Update SQL Editor (or alternatively, right-click the FDUpdateSQL component and select Update SQL Editor).

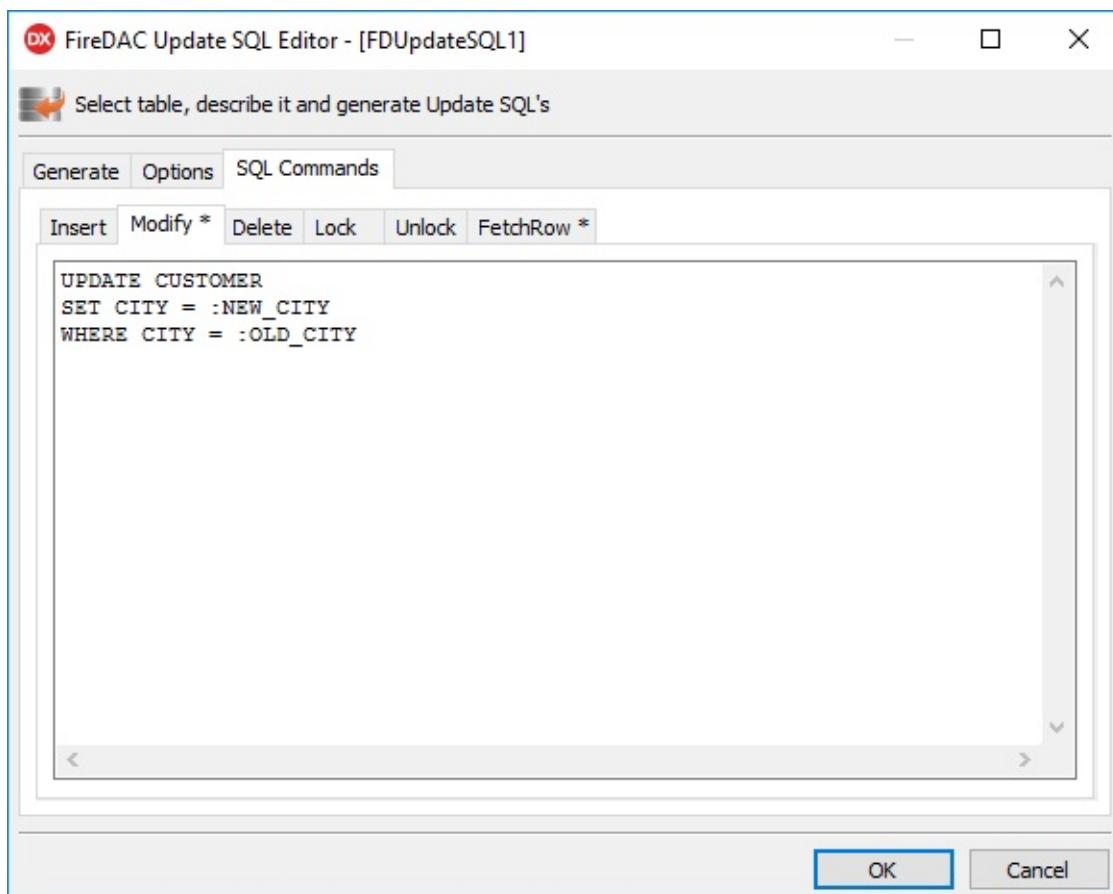


## Chapter 5: More Data Access 127

9. We need to tell the FDUpdateSQL component that it needs to be able to update the City field, and that the City should be considered the key field. Furthermore, when refreshing the query, we need the FDUpdateSQL component to use the City, State\_Province, and Country fields. When we are done configuring the Update SQL editor, it should look like that shown in Figure 5-8.

**Figure 5-8: The Update SQL Editor has been configured to generate our UPDATE query**

10. Now, all we need to do is click on the Generate SQL button. After we do that, we can click on the SQL Commands tab to see the queries that FireDAC has created for us. In this case, due to the properties we have configured for the FDQuery, there are only two generated queries, one



for the UPDATE query, and the other for the SELECT (FetchRow). The UPDATE query is shown in Figure 5-9.

### **Figure 5-9: The generated UPDATE query as shown in the Update SQL Editor**

#### **Creating the Parameterized FDUpdateSQL Queries**

As you can see in Figure 5-9, the generated query is a parameterized query, where the parameter naming convention is strictly defined. This is necessary so that FDUpdateSQL can properly bind data to those parameters before the necessary queries are executed. The query parameters follow the pattern shown in Table 5-1.

Chapter 5: More Data Access 129

#### **Pattern**

#### **Description**

#### **Example**

:NEW\_fieldname The new value from an inserted or

:NEW\_City

updated field.

:OLD\_fieldname

The old value from a deleted or updated :OLD\_City

field.

:fieldname

The current value of the field.

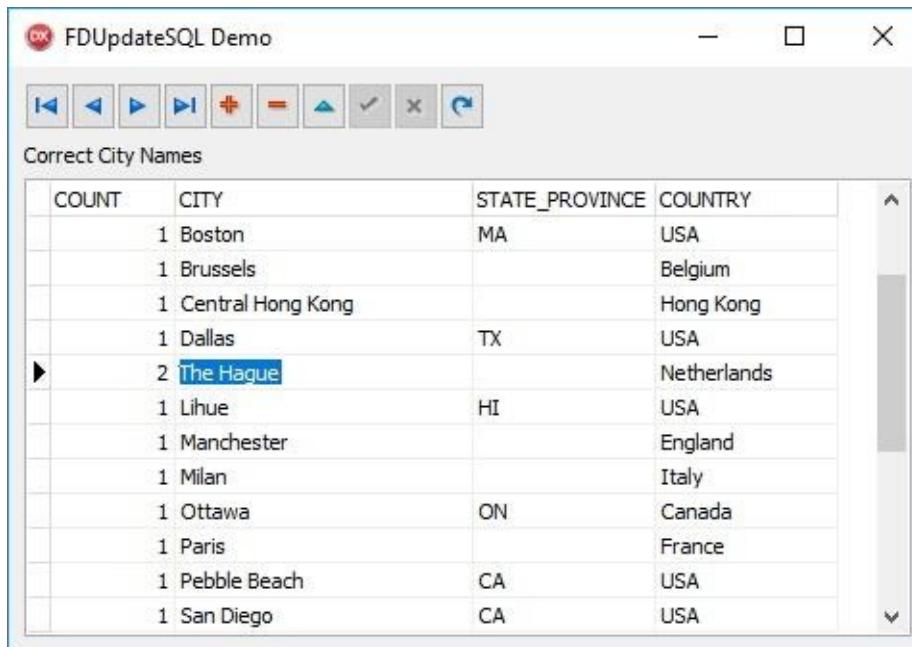
:City

#### **Table 5-1: The query parameter naming conventions required by FDUpdateSQL**

You can now run the project. After running the project, you notice that the Dutch city that you know as The Hague, the city where the United Nation's International Court of Justice meets, has been entered using the Dutch name (Den Haag). Since you've had shipping issues using that name in the past, you move your cursor to the City field for Den Haag, press F2, and enter The Hague.

Now, click the Post button in the DBNavigator to apply this update. If you now click the Refresh button on the DBNavigator, the query will re-execute and you will see that your update has been applied, even though the query should

technically be readonly. The updated form is shown in Figure 5-10.



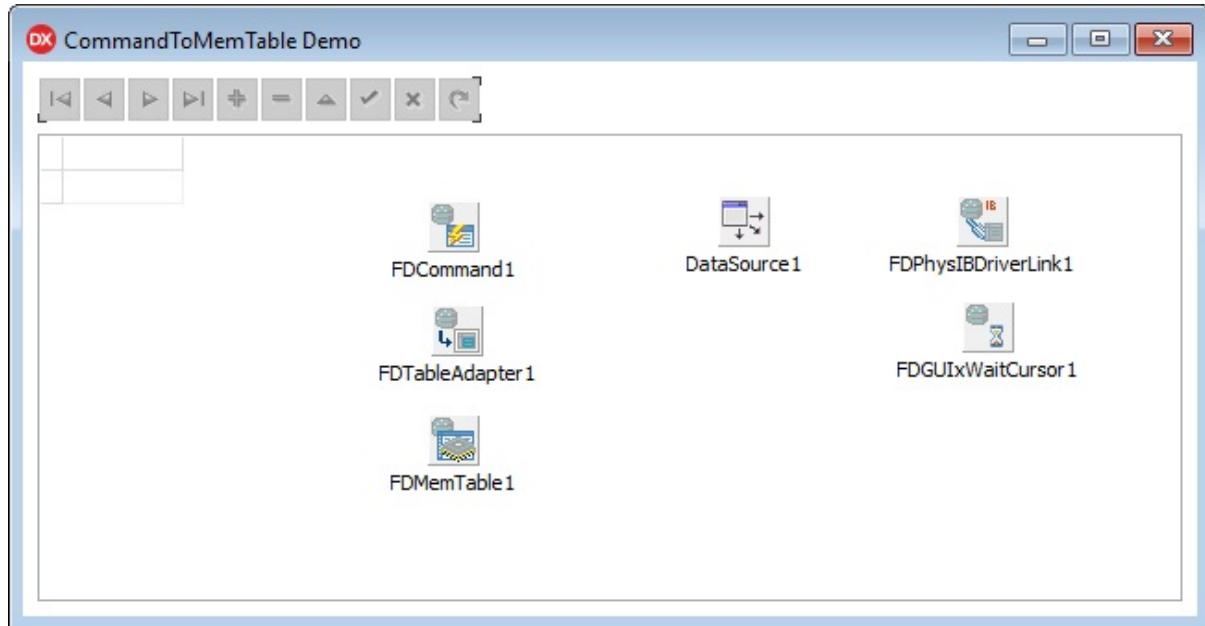
130 Delphi in Depth: FireDAC

**Figure 5-10: What should have been a readonly query has been updated using an FDUpdateSQL component FDCommand, FDTAdapter, and FDMemTable**

Before we move on to the serious matter of managing database transactions, I want to take a moment to demonstrate something that I mentioned in the preceding chapter. Specifically, that FireDAC datasets encapsulate a variety of other FireDAC classes to perform their tasks. For example, a FireDAC query employs an FDMemTable to hold the result set that is returned from SELECT

statements, an FDCommand object to execute the query, and an FDTAdapter that interfaces between these two components. Of course, there are a great many additional FireDAC classes involved, but these are the three that are easy to include in a demonstration, since they are prominent components on the FireDAC tabs on the Tool Palette.

The FDCommandToMemTable project provides a nice little demonstration of this separation of responsibilities. The main form of this project is shown in the form designer in Figure 5-11.



## Chapter 5: More Data Access 131

**Figure 5-11: The main form of the FDCommandToMemTable project**

*Code: The FDCommandToMemTable project is included in the code download.*

With respect to the three FireDAC components that will do the heavy lifting, I have set only five properties total to get this project to work. (Note that I specifically avoided mentioning the FDPhysIBDriverLink and the FDGUIxWaitCursor, since their only purpose is to supply some resources that FireDAC needs.)

For the FDCommand, I set its Connection property to point to the

FDConnection on the data module that I describe in Appendix A (and which is used by nearly every example project in this book). In addition, I set its CommandText property to an SQL string, as shown here:

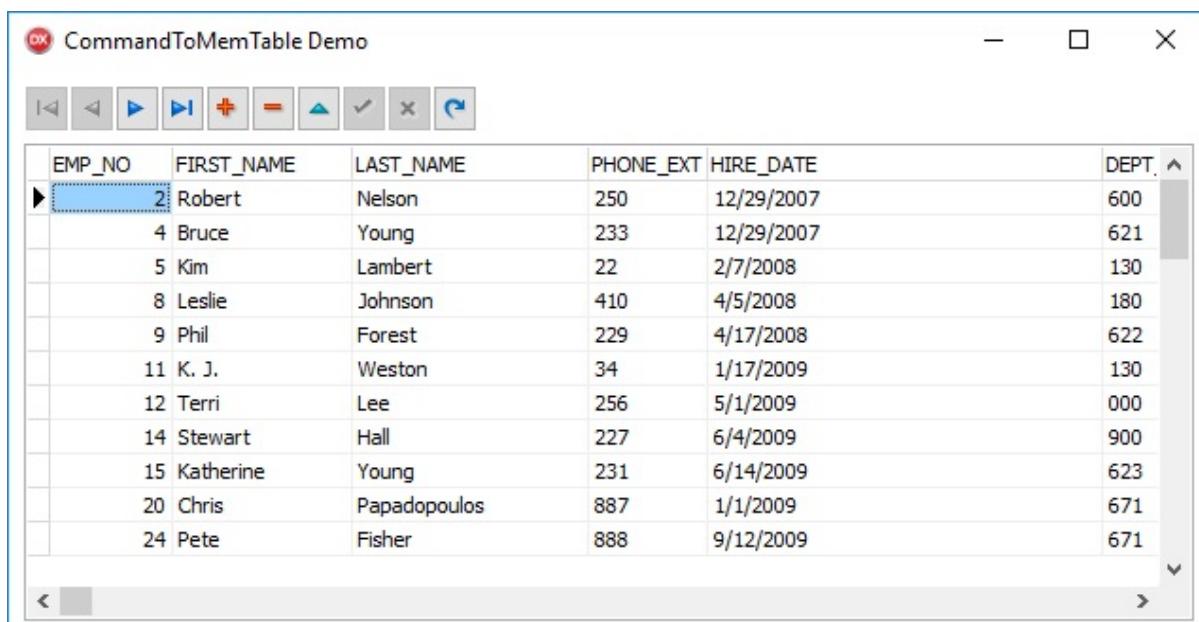
```
SELECT * FROM Employee
```

For the FDTableAdapter, I set its SelectCommand property to FDCommand1.

Finally, I set the Adapter property of the FDMemTable to FDTableAdapter1.

That's it for the FireDAC components.

The rest of the interface is predictable. The DBNavigator and DBGrid use the DataSource, and the DataSource points to the FDMemTable. Really, a demonstration project could not be simpler, or more eye-opening.



## 132 Delphi in Depth: FireDAC

When you run the project, your form looks like that shown in Figure 5-12. And, if you edit a record in the DBGrid and then click the Post button on the DBNavigator, or simply move off that record, the update is posted to the underlying table. You can even delete records by pressing Ctrl-Delete or clicking the Delete button on the DBNavigator (though I don't recommend that as you do this as it will compromise the sample database), and insert records by pressing Ins or clicking the Insert button on the DBNavigator. And it all just works.

### **Figure 5-12: A fully editable table is available on this form**

Ok, I lied just a little. Because I am using the book's data module, which permits you to change the databases for all of the book's projects by editing a single line in the DataPaths.pas file, I did remove the auDesignTime and auRunTime flags from the FDCommand's ActiveStoredUsage property, and I added a single line of code to the form's OnCreate handler to open the FDCommand. But other than that, this project is pristine.

So, what's the point of this demonstration? In short, FireDAC is very capable even with minimal configuration. And, the best news is that you do not need to use an FDCommand, an FDTableAdapter, and an FDMemTable in your application. Just drop a single FDQuery into your project, define an SQL statement to perform your data magic, and point it to an active FDConnection, and you are ready to work.

## Chapter 5: More Data Access 133

### **Managing Transactions**

A transaction is a wrapper for operations against your database. They permit you to signal that you are beginning to perform a task, and likewise signal when that task is complete. More importantly, a transaction permits you to require that the task is completed correctly, or not at all. As a result, the role of a transaction is to ensure that your database remains in a consistent state.

Let's begin with a simple example. Your database is designed to record product orders for your customers. A customer, using your application, orders five items, which are added to their shopping cart. At the conclusion of the interaction, the customer submits their order. A transaction can ensure that all five items are posted to the OrderItems table of your database, or none of them are (after which it is your application's responsibility to notify the customer that something was wrong with the order). You don't want just two

items ordered, or four — you want all five or none.

Talking about transactions is complicated by the fact that not all databases handle transactions in the same way. Some support more features than others, and some can be pretty picky about how to start and commit transactions.

Fortunately, FireDAC handles a lot of this for you, based on the database driver you are using. In addition, using the FDTxOptions object available with

FDConnection and FDTransaction classes, you can control some of the finer points of transaction management without writing a lot of code.

## **Implicit and Explicit Transactions**

FireDAC supports two types of transactions: Implicit and explicit.

An implicit transaction happens in the background without your involvement. In short, when you execute a query without explicitly starting a transaction, FireDAC starts one for you, and at the completion of the query, the transaction is either committed or rolled back (depending on the success of your query).

If you don't want, or don't need, implicit transactions, you can turn them off by setting the FDTxOptions.AutoCommit to False (the default is True).

While implicit transactions are nice, they only work at the single SQL statement execution level. Specifically, when AutoCommit is True, each query you send to your server is wrapped in a transaction, which means that individual queries are ensured to complete in an all or nothing fashion, but this does little for you when you have several operations that must be committed or rolled back as a group.

## 134 Delphi in Depth: FireDAC

And this is where explicit transactions come in, and they are what most database developers think of when they talk about transactions. Explicit transactions are begun through an explicit call to StartTransaction, after which, two or more queries are typically executed. After the last query is invoked, the transaction is concluded with an explicit call to Commit. If everything works properly, the call to Commit ensures that all of the operations performed by the queries executed while the transaction was in force are embodied in the database. In other words, Commit ensures that the database is in a stable state. The one or more query statements that you execute are executed *atomically*, meaning that they all succeed, or none succeed.

If at least one of the SQL queries that you execute against your database

cannot be completed successfully, it is up to you to make sure that the entire transaction, including those queries that had already succeeded, be reversed, and no further queries that are part of the operation are attempted. Canceling the successful queries executed during the course of a transaction is accomplished by calling the transaction's Rollback method.

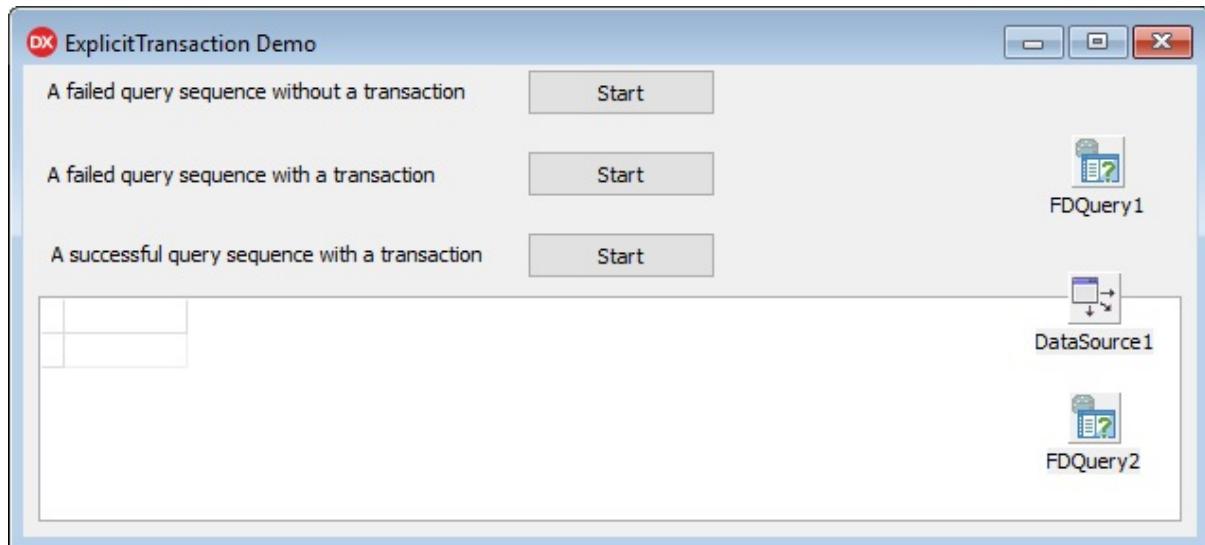
Based on this description, you should already be thinking about the program control structure that you'll need to use, and there is one.

As soon as you start a transaction, you should enter the try block of a try-except.

If even one query raises an exception, your except block will, at a minimum, begin with a call to roll back the transaction. This will have the effect of undoing any work performed since the initiation of the transaction.

Similarly, the very last statement in the try block will be a call to commit the transaction, which, if it doesn't raise an exception and branch to the except block (which it shouldn't), the effects of two or more query operations will be present in their complete form in the underlying database.

Explicit transactions are demonstrated in the FDExplicitTransactions project, whose main form is shown in Figure 5-13.



## Chapter 5: More Data Access 135

### Figure 5-13: The main form of the FDExplicitTransactions project

*Code: You can find the FDExplicitTransactions project in the code download.*  
The top button will execute one good query and one that violates a table constraint (a requirement that the Country field of the Customer table contains a value found in the Country field of the Country table). But there is no transaction to ensure atomicity — that is, complete success or complete failure.

The second button executes the same two queries, but this time within the scope of a transaction. The third button executes two valid queries, again within the scope of a transaction.

All three of these buttons start their processing by first deleting any records associated with the success or failure of the INSERT queries associated with these buttons, which ensures that each button starts from the same starting point.

In addition, after each button has attempted to insert the two records, the current data in the Customer table is displayed in the provided DBGrid, and the last record is made the current record.

Here is an example of how these event handlers look. In this case, I'm displaying the properly designed event handler, which wraps the two insert queries in a transaction, but which includes an invalid Country field value in the second query (the Country table includes the country name England, but not United Kingdom):

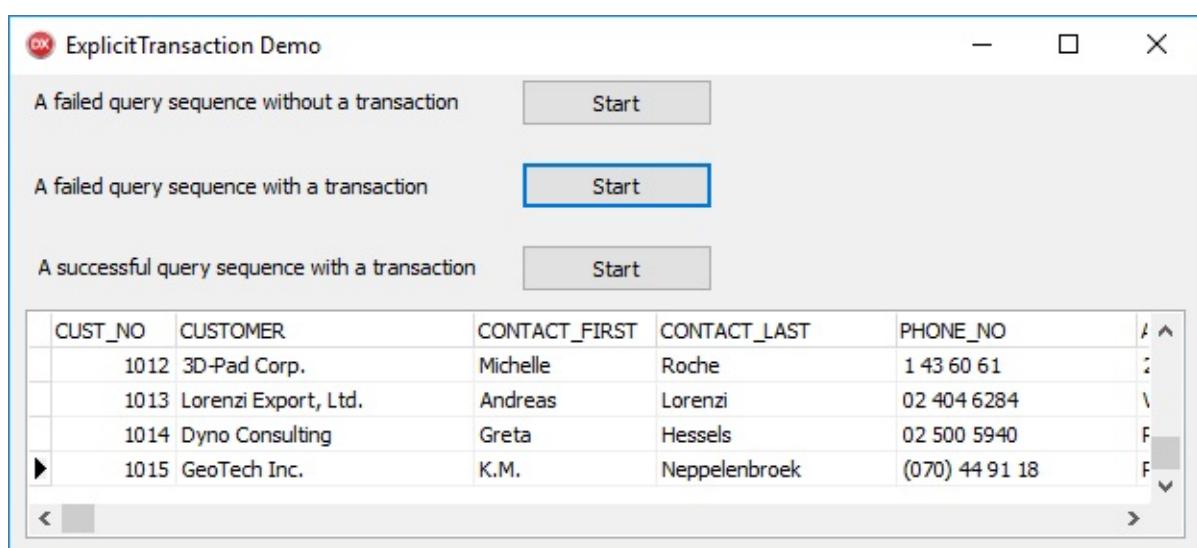
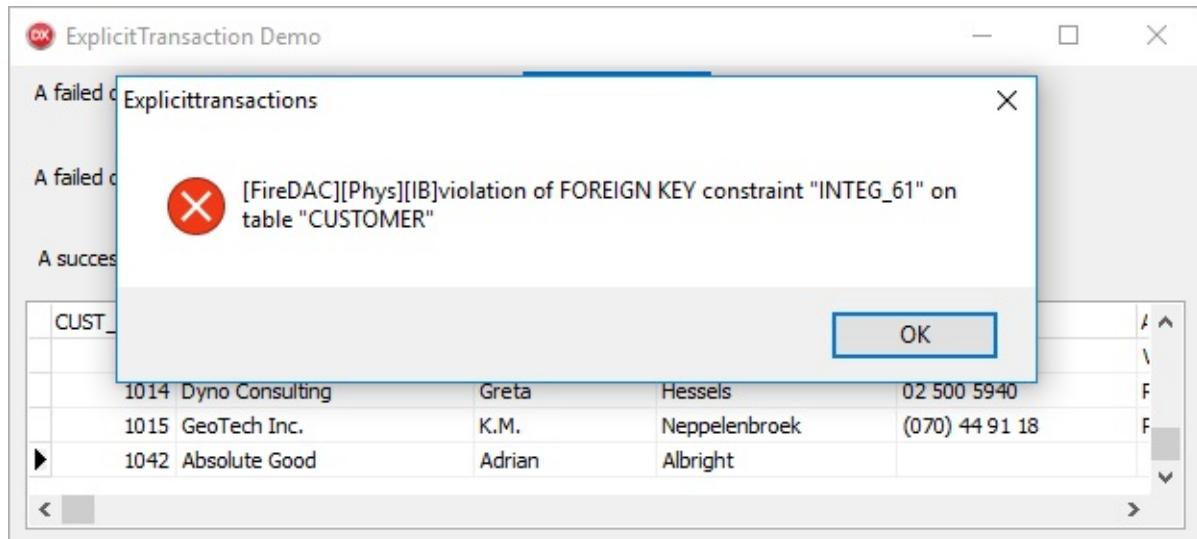
```
procedure TForm1.btnTransactionBadQueryClick(Sender: TObject);
begin
  136 Delphi in Depth: FireDAC
  try
    RemoveTestRecordsIfExist;
    FDQuery1.SQL.Text := SQLstring;
    FDQuery1.Prepare;
    try
      SharedDMVcl.FDConnection.StartTransaction;
```

```

try
  FDQuery1.Params[0].Value := 'Absolute Good';
  FDQuery1.Params[1].Value := 'Adrian';
  FDQuery1.Params[2].Value := 'Albright';
  FDQuery1.Params[3].Value := 'Allanville';
  FDQuery1.Params[4].Value := 'USA';
  FDQuery1.ExecSQL;
  FDQuery1.Params[0].Value := 'Bagle Bonanza';
  FDQuery1.Params[1].Value := 'Brian';
  FDQuery1.Params[2].Value := 'Brukker';
  FDQuery1.Params[3].Value := 'Blackpool';
  FDQuery1.Params[4].Value := 'United Kingdom';
  FDQuery1.ExecSQL;
  SharedDMVcl.FDConnection.Commit;
except
  SharedDMVcl.FDConnection.Rollback;
end;
finally
  FDQuery1.Unprepare;
end;
finally
  UpdateDBGrid;
end;
end;

```

Figure 5-14 shows how the Customer table appears when two queries, one that succeeds and one that fails, are executed without a transaction. Not only does an error message get displayed, noting the constraint violation, but the DBGrid includes one, but not both, of the query results.



## Chapter 5: More Data Access 137

**Figure 5-14: Since there was no transaction, one query succeeded and one failed**

By comparison, Figure 5-15 shows what happens when the same two queries are executed in a transaction. While one query succeeded and one failed, neither of the records was inserted.

**Figure 5-15: Only one of the two queries succeeded, but no records were inserted**

The screenshot shows a Windows application window titled "ExplicitTransaction Demo". It contains three sections: "A failed query sequence without a transaction" with a "Start" button, "A failed query sequence with a transaction" with a "Start" button, and "A successful query sequence with a transaction" with a "Start" button. The third section's "Start" button is highlighted with a blue border. Below these sections is a grid displaying customer data:

CUST_NO	CUSTOMER	CONTACT_FIRST	CONTACT_LAST	PHONE_NO	
1014	Dyno Consulting	Greta	Hessels	02 500 5940	F
1015	GeoTech Inc.	K.M.	Nepellenbroek	(070) 44 91 18	F
1046	Absolute Good	Adrian	Albright		
▶ 1047	Bagle Bonanza	Brian	Brukker		

## 138 Delphi in Depth: FireDAC

Finally, Figure 5-16 shows what happens when both queries succeed. This same result would occur without a transaction, but the transaction is there specifically for those cases where failure might occur.

**Figure 5-16: Both queries were successful, and the transaction committed the results**

### Transaction Isolation

As soon as you begin a transaction, transaction isolation determines what your queries can see from the database, and what other connections to the database can see about the updates you are making, that is until you have committed or rolled back your transaction. What options you have regarding transaction

isolation depends on your database.

*Note: A transaction is always associated with exactly one FDConnection. In the following discussion, when I refer to a query within a transaction, I am*

*referring to a query that uses the connection associated with the transaction.*

*When I refer to a query outside of a transaction, I am referring to a query that uses a connection that is different from the one associated with the transaction.*

FireDAC supports three transaction isolation levels (though some databases do not support all three). These three are: Dirty read, repeatable read, and read committed. Each of these levels is described in the following sections.

Chapter 5: More Data Access 139

## **DIRTY READ ISOLATION**

The dirty read isolation is the least isolated transaction isolation. When employing dirty read, updates made within a transaction that have not yet been committed can be read by queries outside of the transaction. It is called a *dirty read* because, as must be obvious, until that transaction is committed those updates might be reverted through a call to rollback, in which case those outside queries have read data that was not saved.

## **READ COMMITTED ISOLATION**

Queries you execute within a transaction using ReadCommitted isolation can only read values from the database that have already been successfully committed by other transactions. Similarly, queries outside of the current transaction cannot see the updates that you are making to the database until you either commit or rollback your transaction. During your transaction, queries outside your transaction are reading the data as it existed prior to your starting the transaction.

## **REPEATABLE READ ISOLATION**

When using a repeatable read transaction, your queries within the transaction can read the current state of the database once, after which this one read represents the database as far as queries within the transaction are concerned.

Subsequent changes to the database made by queries outside your transaction are not visible to your queries until your transaction is committed or rolled back.

This is the most isolated transaction isolation.

You configure transaction isolation using the FDTxOptions.Isolation property.

This property has four possible values: xiDirtyRead, xiReadCommitted, xiRepeatableRead, and xiUnspecified. The default is xiReadCommitted. If you select xiUnspecified, FireDAC uses the default isolation level supported

by the underlying database.

## Nested Transactions

In theory, a nested transaction is an atomic operation that is started, and then committed or rolled back, from inside an outer transaction, without affecting the integrity of the outer transaction. In addition, even those nested transactions that are successfully committed will be rolled back if the outer transaction is ultimately rolled back. In practice, there are only two databases that support nested transactions. These are InterBase and Firebird (an open source InterBase spin-off).

### 140 Delphi in Depth: FireDAC

FireDAC supports nested transactions when using the InterBase or Firebird drivers against an InterBase or Firebird database. For other databases that do not support nested transactions, but that do support save points, Firebird emulates nested transactions using save points. For those remaining databases, an attempt to start a new transaction when a transaction is already under way will result in the raising of an exception.

## Asynchronous Queries

FireDAC also support the asynchronous execution of queries. For example, if you set the `ResourceOptions.CmdExecMode` property of an FDQuery to `amNonBlocking`, FireDAC will execute the query on a worker thread.

Importantly, FireDAC manages the details of this worker thread, destroying the thread once the query is finished. While this makes asynchronous execution easy to employ, you still need to observe sound multithreaded practices. For example, a query executed asynchronously should not be connected to a

DataSource or BindSourceDB during its execution. For example, you can call `FDQuery.DisableControls` to disassociate a query from its DataSource prior to executing the query asynchronously, calling `FDQuery.EnableControls` from the `FDQuery.AfterOpen` event handler.

When a query is running asynchronously, you can monitor its progress by reading the query's `Command.State` property. Among other things, you can use this property to determine that the query is executing, is in the process of fetching the result set, or has completed execution and has an accessible result set.

*Note: FireDAC permits only one query to be executed asynchronously at a time from your application's main thread of execution. If you need to execute*

*two or more queries concurrently, you need to encapsulate those queries in separate threads.*

There are several additional execution modes supported by FireDAC datasets.

The default mode is amBlocking, which blocks the calling code until the query returns. The amBlocking mode is the recommended mode when executing a

query asynchronously from a custom TThread. Using this mode, the calling thread is blocked until the query returns, but all other threads, including the main thread (the user interface), remain responsive.

## Chapter 5: More Data Access 141

Executing amAsync executes the query asynchronously, like amNonBlocking (there is a difference between amAsync and amNonBlocking, but I am not entirely sure what it is). Finally, you can execute a query using the amCancelDialog mode. In this mode, a dialog box is displayed immediately prior to the execution of the query. The query is then executed in a blocking mode, where the calling thread and the user interface are blocked. Importantly, the user can interact with the dialog box, permitting the user to cancel (abort) the query before it returns. This mode is useful when you need to execute a potentially lengthy query, but do not want to have to manage the additional complexity introduced by worker threads.

When your application includes more than one connection to a single database, something that will necessarily happen when you create custom worker threads from which to execute concurrent queries, you should consider using an

FDManager. The FDManager class defines a connection definition that can be shared by multiple FDConnections, simplifying the process of configuring each FDConnection and potentially enabling connection pooling.

*Code: Examples of asynchronous queries can be found in the FDThreadedExecute project in the code download.*

## Monitoring FireDAC Queries

FireDAC includes components and utilities that make it easy to trace FireDAC's interaction with the underlying databases. Using the FDMoniFlatFileClientLink, you can write trace information to a file, or you can implement your own

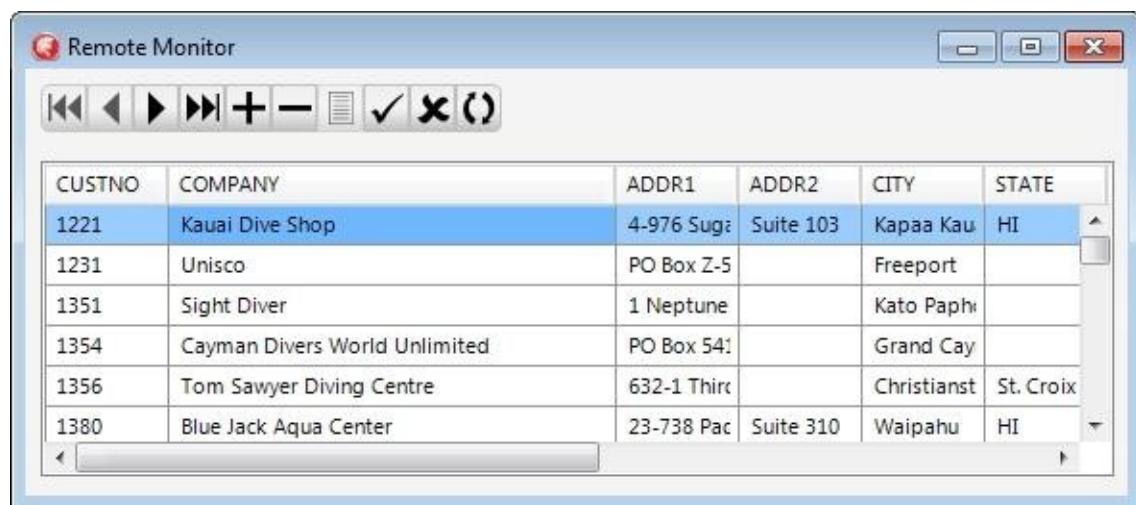
monitor mechanism using the FDMoniCustomClientLink, which triggers an event handler with informative parameters. You implement your custom tracing operations from within this event handler.

But the real gem in this collection is the FDMoniRemoteClientLink. Using this component, you can monitor trace information at runtime using the FireDAC

Monitor utility that ships with Delphi. (The FireDAC Monitor utility is located in RAD Studio's bin directory.) You can monitor local applications, but since FDMoniCustomClientLink and the FireDAC Monitor communicate via TCP/IP

(Transmission Control Protocol/Internet Protocol), you can monitor the trace stream from almost anywhere on the Internet.

Remote monitoring is demonstrated by the FDRemoteMonitorFMX project, shown in the Figure 5-17.



## 142 Delphi in Depth: FireDAC

**Figure 5-17: The main form of the FDRemoteMonitorFMX project**

*Code: The FDRemoteMonitorFMX project is in the code download.*

This project is a simple FireMonkey application that permits the editing of the Customer table in the dbdemos.gdb InterBase database. So long as the FireDAC

Monitor utility is running before this project is loaded, all SQL operations that FireDAC submits to the InterBase server can be viewed. Figure 5-18 depicts the FireDAC Monitor output after a change has been made to the record shown in Figure 5-17, and the monitor trace shown in Figure 5-18 represents FireDAC's updates when the Post button on the BindNavigator is clicked.

**FireDAC Monitor**

File Edit Clients Tools Help

Clear Save... Pause tracing Stay on Top About ...

Trace Output

Trace Output Application Objects

**Log**

No	Time	Text
195	21:03:52:165	<< FetchRow [ARow.Table.Name="customer"]
196	21:03:52:165	>> Form2.FDConnection1.Commit [ConnectionDef="", Retaining=False]
197	21:03:52:165	>> customer: TFDDAptTableAdapter(\$047ECDB0).Update: TFDPhysIBCommand(\$0645102)
198	21:03:52:165	. isc_dsql_free_statement [stmt_handle=\$06963260, option='DSQL_drop']
199	21:03:52:165	<< Unprepare [Command="UPDATE CUSTOMER ..."]
200	21:03:52:165	. isc_commit_transaction [tra_handle=\$06961CE8]
201	21:03:52:165	<< Form2.FDConnection1.Commit [ConnectionDef="", Retaining=False]
202	21:03:52:165	<< Update [ARow.Table.Name="customer"]
203	21:03:52:165	. Destroy [Command="UPDATE CUSTOMER ..."]
204	21:03:52:165	. Adapter customer: TFDDAptTableAdapter(\$047ECDB0).Update: TFDPhysIBCommand(\$0645102)
205	21:03:52:165	>> UnLock [ARow.Table.Name="customer"]
206	21:03:52:165	<< UnLock [ARow.Table.Name="customer"]

**Details**

0 Version: 19.0.13476.1 Copyright © 2011-2012

The screenshot shows the FireDAC Monitor application window. The title bar reads "FireDAC Monitor". The menu bar includes "File", "Edit", "Clients", "Tools", and "Help". Below the menu is a toolbar with icons for "Clear", "Save...", "Pause tracing", "Stay on Top", and "About ...". The main area is titled "Trace Output" and contains two tabs: "Trace Output" (selected) and "Application Objects". The "Trace Output" tab displays a "Log" table with columns "No", "Time", and "Text". The log entries show a sequence of database operations, primarily related to the "customer" table, involving FetchRow, Commit, Update, Unprepare, Destroy, and UnLock commands. The timestamp for all entries is 21:03:52:165. The "Application Objects" tab is currently empty. At the bottom of the window, there is a status bar with the number "0" and the text "Version: 19.0.13476.1 Copyright © 2011-2012".

## Chapter 5: More Data Access 143

**Figure 5-18: The FireDAC Monitor can capture the communication between an application and the underlying database**

Simply placing an FDMoniRemoteClientLink component in a project is not sufficient to enable tracing. You also need to activate the

FDMoniRemoteClientLink by setting its Tracing property to True. In addition, you must configure the MonitorBy parameter of any FDConnection you want to monitor, as well as enable tracing on the FDConnection.

To configure the MonitorBy parameter of the FDConnection, use the Connection Editor and set the MonitorBy property of the FireDAC driver to Remote, Custom, or FlatFile. Alternatively, you can add MonitorBy=*MonType* to the Params collection property, where *MonType* is either Remote, Custom, or FlatFile. Set MonitorBy to Remote when using an FDMoniRemoteClientLink.

Set it to Custom when using an FDMoniCustomClientLink, and to FlatFile when using an FDMoniFlatFileClientLink.

144 Delphi in Depth: FireDAC

You enable tracing on the associated FDConnection by setting its ConnectionIntf.Tracing property to True. These steps can be seen in the following code, which can be found in the OnCreate event handler on the main form of the FDRemoteMonitorFMX project:

```
procedure TForm2.FormCreate(Sender: TObject);  
begin  
  FDMoniRemoteClientLink1.Tracing := True;  
  FDConnection1.Params.Clear;  
  FDConnection1.Params.Add('DriverID=IB');  
  FDConnection1.Params.Add('Database=' +  
    DataPaths.SamplePath + '/dbdemos.gdb');  
  FDConnection1.Params.Add('MonitorBy=Remote');  
  FDConnection1.Params.Add('User_Name=sysdba');
```

```
FDConnection1.Params.Add('Password=masterkey');
FDConnection1.Open();
FDConnection1.ConnectionIntf.Tracing := True;
FDQuery1.Open();
end;
```

FDMoniRemoteClientLink has an additional capability that you might find useful. It permits you to dump the contents of the current record, or even the entire dataset, for retrieval by the FireDAC Monitor utility. In addition, custom objects in your application can be monitored, though this requires some custom programming.

In the following chapter, I show you how to navigate and edit FireDAC datasets.

Chapter 6: Navigating and Editing Data 145

# **Chapter 6**

## **Navigating and**

### **Editing Data**

This chapter describes basic operations that are universal to most TDataSet descendants, with particular attention paid to FireDAC datasets. If you are a seasoned Delphi database developer, you will likely be familiar with most, if not all, of the topics that I introduce in this chapter. In that case, you may want to scan through this chapter.

If you are new to Delphi database development, this chapter is designed to familiarize you with the fundamentals of working with the data exposed by datasets programmatically. It begins with a discussion of Fields, the classes that permit you to read and write data from individual columns in your datasets.

This chapter continues with an introduction to the concept of the current record, followed by a look at the methods and properties that you can use to navigate and edit your data programmatically.

### **Understanding Fields**

There are two basic uses for datasets. The first is to initiate an operation on the connected database, such as creating a new table, altering an index, or inserting data using an SQL INSERT statement. These types of operations do not return a result set.

The second use of a dataset, typically the most common one, is to hold a reference to a tabular structure that consists of zero or more rows and one or more columns. This structure often comes from an SQL SELECT statement or

the execution of a stored procedure. In Delphi's terminology, the rows are referred to as *records* and the columns are called *fields*.

The term field also refers to instances of a class that descends from TField, and you use them to read and write data returned in columns, as well as read the associated metadata. For example, fields that contain text are often members of

146 Delphi in Depth: FireDAC

the TStringField class. This class supports an AsString property of type string

that can be used to read the text from the corresponding column. Similarly, so long as the corresponding column is not readonly, you can write to the underlying field by assigning text to the AsString property.

More accurately, TStringField, like other TField descendants, inherits AsString from TField. In addition to AsString, there are many other properties that permit you to read, and sometimes write, data using other data types. These properties include AsBoolean, AsInteger, and AsFloat, as well as a Value property, which is of type variant.

*Note: If your code must read a large number of fields programmatically, it is most efficient to use one of the Asdatatype properties rather than the Value property. Value, being a variant, involves overhead associated with Delphi determining the data type of the variant, and can slow down your code if you are reading and/or writing a large number of fields.*

Metadata inherited from TField permits you to determine the data type of the underlying data from which the field gets its data (DataType), as well as other information, such as the maximum length of a text field (Size), or whether the field is a key field in an underlying table (pfInKey in ProviderFlags).

You access the fields of a dataset through its Fields property, or from its FieldByName, FieldByNumber, or FindField methods. The Fields property of TDataSet is a reference to a TFields instance, and the default property of TFields is an indexed property called Fields. As a result, the following code refers to the data in the first column returned by a FireDAC query:

```
var
  Data: Variant;
begin
```

```
  Data := FDQuery1.Fields.Fields[0].Value
```

Since TFields.Fields is the default property, you can omit the reference to TDataSet.TFields.Fields and simply index the TDataSet.Fields property, as shown here:

```
var
  Data: Variant;
begin
  Data := FDQuery1.Fields[0].Value;
```

Since Delphi's supported databases can return null values in one or more columns, there are times when a field does not have a value. Nonetheless, if you attempt to read a null field it might actually return a value, such an empty string for a string field, or 0 for a null integer field. However, a null value is not an empty string or zero. As a result, if a null value is a possibility, you may want to first test for a null value using the `TField.IsNull` property. Such a test might look something like the following:

```
var  
Data: Variant;  
begin  
if not FDQuery1.Fields[0].IsNull then  
  Data := FDQuery1.Fields[0].Value;
```

The `FieldByName` method can be used to access a field using a case-insensitive string. If the dataset does not have a field with the name that you pass in the single parameter, an exception is raised. `FindField` also takes a single string parameter. Unlike `FieldByName`, if the dataset does not have a field with the name provided when you call `FindField`, a null value is returned and no

exception is raised. `FieldByNumber` will return a reference to a field based on its zero-based position in the `Fields`.`Fields` property, so long as the number you pass is a valid field position.

In addition to providing you with access to data and metadata, fields also support a number of event handlers. For example, `OnGetText` can be used to modify the data returned when a field is being read, and `OnSetText` can be used to modify the data being written to the current record buffer (this record buffer will be written to the underlying database only if the changes are subsequently posted). Similarly, `OnValidate` permits you to provide custom validation of data being assigned to a field, and `OnChange` will fire in response to a change being accepted to a field.

I really wanted to provide you with a basic introduction to fields in this section, so I have touched on only the most salient features of fields. I will talk more about fields in *Chapter 10, Creating and Using Virtual Fields*, *Chapter 11, Persisting Data*, and *Chapter 12, Understanding FDMemTables*. I do suggest, however, that you find some time to examine the `TField` class declaration in the `Data.DB` unit in detail. There is a lot of good stuff there that can be quite handy when you are working with data.

## The Current Record

As mentioned at the outset of this chapter, datasets often consist of one or more fields and zero or more records. And in the world of Delphi TDataSets, one of the more important concepts is that of the *current record*.

The Delphi TDataSet class is built around a navigational model of data access.

Specifically, a given dataset, when it contains data, will contain one or more records and exactly one of those records (it may be the only record) will be the current record.

The current record, which by default is always the first record in the dataset once the dataset becomes active, is the record whose columns can be accessed using the Fields property of the dataset. As a result, immediately after a dataset becomes active, and assuming that there is at least one record in the result set, the Fields property will provide you with access to the data and the metadata of the first record.

If you want to change the record to which the Fields property points, you need to navigate to another record, and navigation is one of the principle topics of this chapter. In short, navigation is the process of changing which record is the current record.

## Detecting Changes to Record State

Changes that occur when a user navigates or manages a record using a data-aware control is something that you may want to control programmatically. For those situations, there are a variety of event handlers that you can use to evaluate what a user is doing, and provide a customized response.

In addition to the event handlers supported by fields, all datasets possess the event handlers listed in Table 6-1.

AfterCancel

AfterClose

AfterDelete

AfterEdit

AfterInsert

AfterOpen

AfterPost

AfterRefresh

AfterScroll  
BeforeCancel  
BeforeClose  
BeforeDelete  
BeforeEdit  
BeforeInsert  
BeforeOpen  
BeforePost  
BeforeRefresh BeforeScroll  
OnCalcFields  
OnDeleteError  
Chapter 6: Navigating and Editing Data 149  
OnEditError  
OnFilterRecord OnNewRecord OnPostError

### **Table 6-1: The event handlers introduced in TDataSet**

There are additional event handlers that are available in most situations where a dataset is being navigated and edited, and which are always available when data-aware VCL controls are used. These are the event handlers associated with a DataSource. Since all data-aware VCL controls must be connected to at least one DataSource, the event handlers of a DataSource provide you with another source of customization when a user navigates and edits records. These event handlers are: OnDataChange, OnStateChange, and OnUpdateData.

*Code: The code project FDNavigation is available from the code download. See Appendix A for details.*

Several examples of these event handlers are demonstrated in the FDNavigation project, which is used for the remainder of this chapter. For example, the Open and Close menu items, which appear under the File menu, are enabled and

disabled from the FDQuery's AfterOpen and AfterClose event handlers, as shown in the following code segments:

```
procedure TForm1.FDQuery1AfterOpen(DataSet: TDataSet);  
begin
```

```

Open1.Enabled := False;
Close1.Enabled := True;
end;
procedure TForm1.FDQuery1AfterClose(DataSet: TDataSet);
begin
  Open1.Enabled := True;
  Close1.Enabled := False;
end;

```

Of somewhat greater interest are the event handlers associated with the DataSource in this project. For example, the OnDataChange event handler is used to display which record in the FDQuery is the current record in the first panel of the StatusBar, which appears at the bottom of the main form, as shown in Figure 6-1.

The screenshot shows a Windows application window titled "FireDAC Navigation". At the top, there is a toolbar with buttons for First, Next, Prior, Last, Controls (radio buttons for Enabled and Disabled), and various search and navigation controls like Scan Forward, Scan Backward, Move By, RecNo =, and Get Bookmark. Below the toolbar is a large grid table displaying customer data. The columns are labeled CUST\_NO, CUSTOMER, CONTACT\_FIRST, CONTACT\_LAST, PHONE\_NO, ADDRESS\_LINE1, and ADDRESS\_LINE2. The data grid contains 15 rows of customer information. At the bottom of the application window, there is a status bar with the text "Record 1 of 18", "State = dsBrowse", and "BOF = True, EOF = False."

CUST_NO	CUSTOMER	CONTACT_FIRST	CONTACT_LAST	PHONE_NO	ADDRESS_LINE1	ADDRESS_LINE2
1001	Signature Design	Dale J.	Little	(619) 530-2710	15500 Pacific Heights Blvd.	
1002	Dallas Technologies CCC	Glen	Brown	(214) 960-2233	P. O. Box 47000	
1003	Buttle, Griffith and Co.	James	Buttle	(617) 488-1864	2300 Newbury Street	Suite 101
1004	Central Bank DDD	Elizabeth	Brocket	61 211 99 88	66 Lloyd Street	
1005	DT Systems, LTD.	Tai	Wu	(852) 850 43 98	400 Connaught Road	
1006	DataServe International	Tomas	Bright	(613) 229 3323	2000 Carling Avenue	Suite 150
1007	Mrs. Beauvais		Mrs. Beauvais		P.O. Box 22743	
1008	Anini Vacation Rentals	Leilani	Briggs	(808) 835-7605	3320 Lawai Road	
1009	Max	Max		22 01 23	1 Emerald Cove	
1010	MPM Corporation	Miwako	Miyamoto	3 880 77 19	2-64-7 Sasazuka	
1011	Dynamic Intelligence Corp	Victor	Granges	01 221 16 50	Florhofgasse 10	
1012	3D-Pad Corp.	Michelle	Roche	1 43 60 61	22 Place de la Concorde	
1013	Lorenzi Export, Ltd.	Andreas	Lorenzi	02 404 6284	Via Eugenia, 15	
1014	Dyno Consulting	Greta	Hessels	02 500 5940	Rue Royale 350	
1015	GeoTech Inc.	K.M.	Neppelenbroek	(070) 44 91 18	P.O.Box 702	
1054	Absolute Good	Adrian	Albright			
1055	Bagle Bonanza	Brian	Brucker			

## 150 Delphi in Depth: FireDAC

**Figure 6-1: The StatusBar on the main form of the FDNavigation project is updated by OnDataChange and OnStateChange event handlers**

This OnDataChange event handler not only displays the current record number, and how many records are in the FDQuery, but also indicates in the fourth panel of the StatusBar whether an attempt was made to navigate above the first record (Bof) or below the last record (Eof). Bof and Eof are described

in a bit more detail a little later in this chapter.

The following is the OnDataChange event handler for the DataSource in this project:

```
procedure TForm1.DataSource1DataChange(Sender: TObject; Field:  
TField);  
begin  
  StatusBar1.Panels[1].Text := 'Record ' +  
    IntToStr(FDQuery1.RecNo) + ' of ' +  
    IntToStr(FDQuery1.RecordCount);  
  StatusBar1.Panels[3].Text :=  
    'BOF = ' + BoolToStr(FDQuery1.Bof, True) + '. ' +  
    'EOF = ' + BoolToStr(FDQuery1.Eof, True) + '. '  
end;
```

## Chapter 6: Navigating and Editing Data 151

*Note: The value returned by the RecordCount property depends on how you have configured your dataset. Specifically, the Mode, RecordCountMode, MaxRecs, and RowSetSize property of FetchOptions will all affect the number of records returned in the datasets, as well as how RecordCount operates.*

The OnStateChange event handler also updates the StatusBar shown in Figure 6-1. In this event handler, RTTI (runtime type information) is used to display the human-readable version of the FDQuery's State property, which is of type TDataSetState. Here is the definition of TDataSetState:

```
TDataSetState = (dsInactive, dsBrowse, dsEdit, dsInsert,  
dsSetKey, dsCalcFields, dsFilter, dsnewValue, ds oldValue,  
ds curValue, dsBlockRead, dsInternalCalc, dsOpening);
```

In Figure 6-1, the FDQuery is in the Browse state, as seen in the third panel of the StatusBar. The following is the OnStateChange event handler used by the DataSource:

```
procedure TForm1.DataSource1StateChange(Sender: TObject);  
begin  
  StatusBar1.Panels[2].Text :=  
    'State = ' + GetEnumName(TypeInfo(TDataSetState),
```

```
Ord(FDQuery1.State));
```

```
end;
```

OnDataChange triggers whenever an FDQuery arrives at a new record, when a

new value is posted to a field, when an edit is canceled, as well as when an FDQuery arrives at the first record when it is initially opened.

OnStateChange triggers when an FDQuery changes state, such as when it changes from dsBrowse to dsEdit (when a user enters the edit mode), or when it changes from dsEdit to dsBrowse (following the posting or cancellation of a change).

Finally, OnUpdateData triggers when the dataset to which the DataSource points is posting the contents of the record buffer.

152 Delphi in Depth: FireDAC

## **Calculating Performance**

There is one more piece of information that is written to the status bar, and that is the length of time that a particular operation takes. In this project, and several others discussed in later chapters, a TStopWatch is used to calculate how long a particular operation takes. TStopWatch is a record type declared in the

System.Diagnostics unit, and it makes calculating latency of operations easy. I have introduced two helper subroutines to assist in calculating how long operations take, Start and Complete. Here are these methods, which are self-explanatory:

```
procedure Start;
```

```
begin
```

```
StopWatch := TStopWatch.StartNew;
```

```
end;
```

```
procedure Complete;
```

```
begin
```

```
StopWatch.Stop;
```

```
Form1.StatusBar1.Panels[0].Text := 'Elapse time: ' +
```

```
StopWatch.ElapsedMilliseconds.ToString + ' milliseconds';
```

**end;**

Figure 6-2 shows the output of these routines in the first segment of the status bar after a forward scan operation has been executed. In this case, disable controls was not used and the scan took approximately 30 milliseconds.

The screenshot shows a Windows application window titled "FireDAC Navigation". The interface includes a toolbar with buttons for First, Next, Prior, Last, Scan Forward (which is highlighted in blue), Scan Backward, Move By, RecNo =, and Goto Bookmark. Below the toolbar is a status bar displaying "Elapse time: 30 milliseconds", "Record 18 of 18", "State = dsBrowse", and "BOF = False, EOF = True". The main area is a grid table with columns: CUST\_NO, CUSTOMER, CONTACT\_FIRST, CONTACT\_LAST, PHONE\_NO, ADDRESS\_LINE1, and ADDRESS\_LINE2. The data consists of 18 records, with the last record shown being record 18.

CUST_NO	CUSTOMER	CONTACT_FIRST	CONTACT_LAST	PHONE_NO	ADDRESS_LINE1	ADDRESS_LINE2
1002	Dallas Technologies CCC	Glen	Brown	(214) 960-2233	P. O. Box 47000	
1003	Buttle, Griffith and Co.	James	Buttle	(617) 488-1864	2300 Newbury Street	Suite 101
1004	Central Bank DDD	Elizabeth	Brocket	61 211 99 88	66 Lloyd Street	
1005	DT Systems, LTD.	Tai	Wu	(852) 850 43 98	400 Connaught Road	
1006	DataServe International	Tomas	Bright	(613) 229 3323	2000 Carling Avenue	Suite 150
1007	Mrs. Beauvais		Mrs. Beauvais		P.O. Box 22743	
1008	Anini Vacation Rentals	Leilani	Briggs	(808) 835-7605	3320 Lawai Road	
1009	Max	Max		22 01 23	1 Emerald Cove	
1010	MPM Corporation	Miwako	Miyamoto	3 880 77 19	2-64-7 Sasazuka	
1011	Dynamic Intelligence Corp	Victor	Granges	01 221 16 50	Florhofgasse 10	
1012	3D-Pad Corp.	Michelle	Roche	1 43 60 61	22 Place de la Concorde	
1013	Lorenzi Export, Ltd.	Andreas	Lorenzi	02 404 6284	Via Eugenia, 15	
1014	Dyno Consulting	Greta	Hessels	02 500 5940	Rue Royale 350	
1015	GeoTech Inc.	K.M.	Neppelebroek	(070) 44 91 18	P.O.Box 702	
1054	Absolute Good	Adrian	Albright			
1055	Bagle Bonanza	Brian	Brukker			
1056	Nutty Forms	Alan	Ark			

## Chapter 6: Navigating and Editing Data 153

**Figure 6-2: A TStopWatch record is used to measure the speed of operations**

### Navigating Programmatically

In *Chapter 4, Basic Data Access*, I discussed data binding, both with VCL data-aware controls as well as with Live Bindings. That discussion demonstrated how a user could navigate and edit data through the user interface.

This section looks at navigation whether or not data-aware controls are involved, for example, to move to a record that you want to change programmatically. For a dataset, these core navigation methods include: First, Next, Prior, Last, MoveBy, RecNo, and GotoBookmark. Each of these is described in the following sections.

Before I continue, I want to differentiate between navigating and searching. Navigating refers to moving to another record relative to the current record, making the new record the current record. Searching, by comparison, involves

an attempt to locate a record with a particular value or set of values in its fields.

I am covering navigation in this chapter, and cover searching in *Chapter 8, Searching Data*. Bookmarks, a topics that I cover in this chapter, is more similar to searching than the other navigational techniques that I cover in this chapter,

## 154 Delphi in Depth: FireDAC

but I included it here since I felt it had more in common with navigation than the searching techniques that I cover in Chapter 8.

### **Basic Navigation**

The TDataSet interface provides for basic navigation through four methods: First, Next, Prior, and Last. These methods are pretty much self-explanatory.

Each one produces an effect similar to the corresponding button on a DBNavigator.

There is another type of navigation that is similar to First, Next, and so forth.

That navigation, however, is associated with a filter, and is referred to as *filtered navigation*. Filtered navigation is discussed in detail in *Chapter 9, Filtering Data*.

### **Have I Gone Too Far? Bof and Eof**

You can determine whether an attempt to navigate has tried to move outside of the range of records in the dataset by reading the Bof (beginning-of-file) and Eof (end-of-file) properties. Eof will return True if a navigation method attempted to move beyond the end of the table. When Eof returns True, the current record is the last record in the dataset.

Similarly, Bof will return True if a navigation attempt tried to move before the beginning of the dataset. In that situation, the current record is the first record in the dataset.

Bof and Eof were used in the OnDataChange event handler shown earlier in this chapter, to indicate in the StatusBar whether the last navigation attempt tried to move before, or beyond the records of the FDQuery, respectively.

### Figure 6-3

shows how the main form of the FDNavigation project looks when you have used MoveBy to attempt to move beyond the end of the FDMemTable.

The screenshot shows a FireDAC Navigation application window. At the top, there's a toolbar with buttons for First, Next, Prior, Last, Controls (Enabled/Disabled), and a numeric input field for 'Move By' set to 20. Below the toolbar is a grid displaying customer data from a dataset. The columns are labeled CUST\_NO, CUSTOMER, CONTACT\_FIRST, CONTACT\_LAST, PHONE\_NO, ADDRESS\_LINE1, and ADDRESS\_LINE2. The data includes records such as 1002 Dallas Technologies CCC, 1003 Buttle, Griffith and Co., etc. At the bottom of the grid, status messages indicate 'Elapse time: 7 milliseconds', 'Record 18 of 18', 'State = dsBrowse', and 'BOF = False. EOF = True.'

CUST_NO	CUSTOMER	CONTACT_FIRST	CONTACT_LAST	PHONE_NO	ADDRESS_LINE1	ADDRESS_LINE2
1002	Dallas Technologies CCC	Glen	Brown	(214) 960-2233	P. O. Box 47000	
1003	Buttle, Griffith and Co.	James	Buttle	(617) 488-1864	2300 Newbury Street	Suite 101
1004	Central Bank DDD	Elizabeth	Broket	61 211 99 88	66 Lloyd Street	
1005	DT Systems, LTD.	Tai	Wu	(852) 850 43 98	400 Connaught Road	
1006	DataServe International	Tomas	Bright	(613) 229 3323	2000 Carling Avenue	Suite 150
1007	Mrs. Beauvais		Mrs. Beauvais		P.O. Box 22743	
1008	Anini Vacation Rentals	Leliani	Briggs	(808) 835-7605	3320 Lawai Road	
1009	Max	Max		22 01 23	1 Emerald Cove	
1010	MPM Corporation	Miwako	Miyamoto	3 880 77 19	2-64-7 Sasazuka	
1011	Dynamic Intelligence Corp	Victor	Granges	01 221 16 50	Florhofgasse 10	
1012	3D-Pad Corp.	Michelle	Roche	1 43 60 61	22 Place de la Concorde	
1013	Lorenzi Export, Ltd.	Andreas	Lorenzi	02 404 6284	Via Eugenia, 15	
1014	Dyno Consulting	Greta	Hessels	02 500 5940	Rue Royale 350	
1015	GeoTech Inc.	K.M.	Nepellenbroek	(070) 44 91 18	P.O.Box 702	
1054	Absolute Good	Adrian	Albright			
1055	Bagle Bonanza	Brian	Brukker			
1056	Nutty Forms	Alan	Ark			

## Chapter 6: Navigating and Editing Data 155

**Figure 6-3: An attempt to use MoveBy to move beyond the end of the FDQuery has set Eof to True**

### Using MoveBy

MoveBy permits you to move forward and backward in a dataset, relative to the current record. For example, the following statement will move the current cursor five records forward in the FDQuery (if possible):

```
FDQuery1.MoveBy(5);
```

To move backwards in a dataset, pass MoveBy a negative number. For example, the following statement will move the cursor to the record that is 100 records prior to the current records (again, if possible):

```
FDQuery1.MoveBy(-100);
```

MoveBy is demonstrated by the OnClick event handler associated with the button labeled Move By in the FDNavigation project. This event handler is shown in the following code segment:

156 Delphi in Depth: FireDAC

```
procedure TForm1.MoveByBtnClick(Sender: TObject);
```

```
begin
```

```
  FDQuery1.MoveBy(UpDown1.Position);
```

**end;**

## Navigating Using RecNo

The use of RecNo to navigate might come as a surprise. In a dataset, this property can be used for two purposes. You can read this property to learn the position of the current record in the current record order (based on the index currently in use), or the position of the current record in the natural order of the records if no index is selected.

You also can write to this property. Doing so moves the cursor to the record in the position defined by the value you assign to this property. For example, the following statement will move the cursor to the record in the fifth position of the current index order (if possible):

```
FDQuery1.RecNo := 5;
```

The description of the preceding example was qualified by the statement that the operation will succeed if possible. This qualification has two aspects to it. First, the cursor movement will not take place if the current record has been edited, but cannot be posted. For example, if a record has been edited but not yet posted, and the data cannot pass at least one of the dataset's constraints, attempting to navigate off that record will raise an exception and the navigation will not occur.

The second situation where the record navigation might not be possible is related to the number of records in the dataset. Attempting to set RecNo to a record beyond the end of the table, or prior to the beginning of the table, raises an exception.

The use of RecNo is demonstrated in the following event handler, which is associated with the button labeled RecNo =:

```
procedure TForm1.RecNoBtnClick(Sender: TObject);
```

```
begin
```

```
  FDQuery1.RecNo := UpDown2.Position;
```

```
end;
```

Chapter 6: Navigating and Editing Data 157

## Scanning a FireDAC DataSet

Combining several of the methods and properties described so far provides you with a mechanism for scanning a dataset. Scanning begins by moving either to the first record or to the last record of the dataset, and then

systematically moving to each of the remaining records in the dataset, one record at a time. The following pseudo code demonstrates how to scan an FDQuery:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
if not FDQuery1.Active then
  FDQuery1.Open;
  FDQuery1.First;
while not FDQuery1.Eof do
begin
  //perform some operation based on one or
  //more fields of the DataSet
  FDQuery1.Next;
end;
end;
```

### **Disabling Controls While Navigating**

If the dataset that you are navigating programmatically is attached to data-aware controls through a DataSource, and you take no other precautions, the data-aware controls will be affected by the navigation. In the simplest case, where you move directly to another record, the update is welcome, causing the controls to repaint with the data of the new current record. However, when your

navigation involves moving to two or more records in rapid succession, such as is the case when you scan a dataset, the updates can have severe results.

There are two major problems associated with rapid navigation of enabled data-aware controls. First, the flicker caused by the repainting of the data-aware controls as the dataset arrives at each record is distracting. More importantly, however, is the overhead associated with the repaint itself.

Repainting visual controls is one of the slowest processes in most GUI (graphical user interface) applications. If your navigation involves visiting many records, as is often the case when you are scanning, the repaints of your data-aware controls represent a massive and normally unnecessary amount of overhead.

## 158 Delphi in Depth: FireDAC

To prevent your data-aware controls from repainting when you need to programmatically change the current record quickly and repeatedly, you should call the dataset's DisableControls method. When you call DisableControls, the dataset stops communicating with any DataSources that point to it. As a result, the data-aware controls that point to those DataSources are never made aware of the navigation.

Once you are done navigating, call the dataset's EnableControls. This will resume the communication between the dataset and any DataSources that point to it. It will also result in the data-aware controls being immediately instructed to repaint themselves. However, this repaint occurs only once, in response to the call to EnableControls, and is not due to any of the individual navigations that occurred since DisableControls was called.

You may recall that I used Figure 6-2 to demonstrate the output created by the TStopWatch when it is used to measure the speed of an operation. That figure shows that a forward scan without disabling controls took only 30 milliseconds, which probably seems pretty fast.

However, that scan required many repaints, and as I've been describing, repaints are slow. Compare Figure 6-2 to Figure 6-4. In this figure, I have selected the Disabled radio button of the Controls radio group, which produces the same scan but without the repainting. As you can see, in this case no measurable time lapse has been detected. In other words, as far as the TStopWatch is concerned, the forward scan took less than a millisecond.

CUST_NO	CUSTOMER	CONTACT_FIRST	CONTACT_LAST	PHONE_NO	ADDRESS_LINE1	ADDRESS_LINE2
1002	Dallas Technologies CCC	Glen	Brown	(214) 960-2233	P. O. Box 47000	
1003	Buttle, Griffith and Co.	James	Buttle	(617) 488-1864	2300 Newbury Street	Suite 101
1004	Central Bank DDD	Elizabeth	Brocket	61 211 99 88	66 Lloyd Street	
1005	DT Systems, LTD.	Tai	Wu	(852) 850 43 98	400 Cornaught Road	
1006	DataServe International	Tomas	Bright	(613) 229 3323	2000 Carling Avenue	Suite 150
1007	Mrs. Beauvais		Mrs. Beauvais		P.O. Box 22743	
1008	Anini Vacation Rentals	Leilani	Briggs	(808) 835-7605	3320 Lawai Road	
1009	Max	Max		22 01 23	1 Emerald Cove	
1010	MPM Corporation	Miwako	Miyamoto	3 880 77 19	2-64-7 Sasazuka	
1011	Dynamic Intelligence Corp	Victor	Granges	01 221 16 50	Florhofgasse 10	
1012	3D-Pad Corp.	Michelle	Roche	1 43 60 61	22 Place de la Concorde	
1013	Lorenzi Export, Ltd.	Andreas	Lorenzi	02 404 6284	Via Eugenia, 15	
1014	Dyno Consulting	Greta	Hessels	02 500 5940	Rue Royale 350	
1015	GeoTech Inc.	K.M.	Neppelebroek	(070) 44 91 18	P.0.Box 702	
1054	Absolute Good	Adrian	Albright			
1055	Bagle Bonanza	Brian	Brukker			
1056	Nutty Forms	Alan	Ark			

**Figure 6-4: A forward scan with controls disabled is fast**

It is important to recognize that between the time you call DisableControls and EnableControls, the dataset is in an abnormal state (the GUI is detached, at least with respect to the dataset). In fact, if you call DisableControls and never call a corresponding EnableControls, the dataset will appear to the user to have

stopped functioning based on the lack of activity in the data-aware controls. As a result, it is essential that if you call DisableControls, you structure your code in such a way that a call to EnableControls is guaranteed. One way to do this is to enter a try block immediately after a call to DisableControls, invoking the corresponding EnableControls in the finally block.

This is demonstrated in the OnClick event handler associated with the button labeled Scan Forward on the main form of the FDNavigation project. In this event handler, if the RadioGroup named ControlsStateBtnGrp is set to 1 (the Disabled radio button is selected), the DisableControls method of the dataset is called before the scanning takes place. Furthermore, once the scanning is complete, the finally clause ensures that controls are once again enabled (if they were initially disabled):

```
procedure TForm1.ScanForwardBtnClick(Sender: TObject);
```

```
begin
```

```
 160 Delphi in Depth: FireDAC
```

```
  Start;
```

```
  if ControlsStateBtnGrp.ItemIndex = 1 then
```

```
    FDQuery1.DisableControls;
```

```
    try
```

```
      FDQuery1.First;
```

```
      while not FDQuery1.Eof do
```

```
        begin
```

```
          //do something with a record
```

```
          FDQuery1.Next;
```

```
        end;
```

```
      finally
```

```
if ControlsStateBtnGrp.ItemIndex = 1 then
  FDQuery1.EnableControls;
end;
Complete;
end;
```

## Navigating Using Bookmarks

A bookmark is a reference to a record to which you want to return. You create a bookmark by calling the GetBookmark method of the dataset. GetBookmark returns a TBookmark instance (which in all of the recent versions of Delphi is a generic array of TByte). This bookmark references the current record of the dataset. You can then navigate off of that record with the knowledge that you can return to it by calling the GotoBookmark method, to which you pass the previously saved bookmark.

The use of GetBookmark and GotoBookmark is demonstrated in the FDNavigation project. This project supports only one bookmark at a time, and it is stored in a private field of the form named FBookmark, as shown here:

```
private
{ Private declarations }
FBookmark: TBookmark;
```

Initially, the GotoBookmark button is not enabled. It is enabled only after you have created a bookmark by clicking the button labeled Get Bookmark. This is shown in the OnClick event handler of this button:

```
procedure TForm1.btnGetBookmarkClick(Sender: TObject);
begin
  FBookmark := FDQuery1.GetBookmark;
  btnGotoBookmark.Enabled := True;
```

Chapter 6: Navigating and Editing Data 161

```
end;
```

Once you have created a bookmark, if you navigate off of the record, you can make it the current record again by calling GotoBookmark as shown in the following code:

```
procedure TForm1.btnGotoBookmarkClick(Sender: TObject);
```

```
begin
  Start;
  FDQuery1.GotoBookmark( FBookmark );
  Complete;
end;
```

*Note: DBGrids support multi-record selection when the dgMultiSelect flag appears in its Options property. When multi-select is enabled, selected records are identified by the SelectedRows property, which is of the type TBookmarkList. A TBookmarkList identifies the selected records using a collection of Bookmarks.*

## **Editing a DataSet**

While not technically a navigation issue, one of the primary reasons for navigating a dataset programmatically is to locate a record that you want to change. Since this topic is covered only in passing in other chapters of this book, I am going to take this opportunity to discuss the programmatic editing of records in a dataset in a bit more detail here.

You edit a current record in a dataset by calling its Edit method, after which you can change the values of one or more of its Fields. As mentioned earlier in this chapter, those changed values are stored in the current record's record buffer until that record is posted. After changing one or more fields in the current record, you can post the record by explicitly calling the Post method.

Alternatively, you can simple navigate off the record to attempt to post the new values.

If you modify a record, and then decide not to post the change, or discover that you cannot post the change, you can cancel all changes to the record by calling the dataset's Cancel method. For example, if you change a record, and then find

### 162 Delphi in Depth: FireDAC

that calling Post raises an exception, you can call Cancel to cancel the changes and return the dataset to the dsBrowse state.

To insert and post a record, you have several options. You can call Insert or Append, after which your cursor will be on a newly inserted record, assuming that you started from the dsBrowse state. If you were editing a record prior to calling Insert or Append, a new record will not be inserted if the record being edited cannot be posted. Once a record is successfully inserted, assign data to

the Fields of that record and call Post to post those changes (or successfully navigate off of the record).

*Note: So long as there is no active index to control the record order, Insert will insert the new record at the position of the current record, while Append will add the new record to the end of the dataset.*

The alternative to calling Insert or Append is to call InsertRecord or AppendRecord. These methods insert a new record, assign data to one or more fields, and attempt to post, all in a single call. The following is the syntax of the InsertRecord method. The syntax of AppendRecord is identical (with the

obvious exception being that the methods have different names):

**procedure** InsertRecord(**const** Values: array of **const**); You include in the constant array the data values that you want to assign to each field in the dataset. If you want to leave a particular field unassigned, include the value null in the constant array. Fields you want to leave unassigned at the end of the record can be omitted from the constant array.

For example, if you are inserting and posting a new record into a four-field FDQuery, and you want to assign the first field the value 1000 (a field associated with a unique index), leave the second and fourth fields unassigned, but assign the value ‘new’ to the third record, your InsertRecord invocation may look something like this:

```
FDQuery1.InsertRecord([1001, null, ‘new’]);
```

The following code segment demonstrates another instance of record scanning, this time with edits that need to be posted to each record. In this example, Edit and Post are performed within try blocks. If the record was placed in the edit

Chapter 6: Navigating and Editing Data 163

mode (which corresponds to the dsEdit state) and cannot be posted, the change is canceled. If the record cannot even be placed into edit state, which for a dataset should only happen if the dataset’s CanModify property is false, or if the FireDAC dataset’s UpdateOptions property

(TFDBottomUpdateOptions.EnableUpdate and TFDBottomUpdateOptions.ReadOnly) do not allow editing, the attempt to post changes is skipped:

```
procedure TForm1.Button1Click(Sender: TObject);
```

```

begin
if not FDQuery1.Active then
  FDQuery1.Open;
  FDQuery1.First;
while not FDQuery1.EOF do
  begin
    try
      FDQuery1.Edit;
    try
      FDQuery1.Fields[0].Value :=  

       UpperCase(FDQuery1.Fields[0].Value);
      FDQuery1.Post;
    except
      //record cannot be posted. Cancel;
      FDQuery1.Cancel;
    end;
    except
      //Record cannot be edited. Skip
    end;
    FDQuery1.Next;
  end; //while
end;

```

*Note: Rather than simply canceling changes that cannot be posted, an alternative except clause would identify why the record could not post, and produce a log that could later be used to apply the change. Also note that if these changes are being cached, the FireDAC datasets class provides*

*OnReconcileError and OnUpdateError event handlers that can be used to process failed postings for updates in a subsequent call to ApplyUpdates.*

In the next chapter, I show you how to create and use indexes.

# Chapter 7

## Creating Indexes

In many respects, an index on a FireDAC dataset is like that on any other DataSet descendant. Specifically, an index controls the order of records in the dataset, as well as enables or enhances a variety of other operations based on record-specific contents, such as searches, ranges, and dataset linking.

Unlike a dataset's structure, which is normally obtained from an SQL SELECT

statement or the execution of a stored procedure that returns a result set, a dataset's indexes are not. Specifically, when a FireDAC dataset is loaded with data obtained from a result set, or is loaded from a previously saved file, the dataset's structure is largely (and usually entirely) defined by the result set, or defined by the metadata loaded from the saved file. Indexes, by comparison, are defined for the dataset's data contents explicitly.

Even if the underlying table in the database has indexes, an SQL SELECT from that table will not result in an index in the dataset. In addition, the dataset can have indexes even when the table in the database does not. In other words, a database's indexes and a FireDAC dataset's indexes are completely

independent.

Consider the Customer table found in the example employee.gdb InterBase database that ships with Delphi. There are four customer table-related indexes present in the database: RDB\$PRIMARY1, CUSTNAMEX, CUSTREGION, and RDB\$FOREIGN23. Accessing that data using an FDTable or an FDQuery

with a SELECT \* FROM Customer query will load that table's data into the dataset, and its structure will reflect that of the Customer table. But after the execution of the query, the FireDAC dataset will not have any indexes. Indexes, if you want them, must be defined explicitly for the FireDAC dataset either at design time or at runtime.

This is not to say that FireDAC ignores constraints defined for tables in a database. Indeed, by default, it observes and enforces primary keys, unique keys, foreign keys, and other constraints. These operations are controlled

though the `UpdateOptions` properties, including `CheckReadOnly`, `CheckRequired`, and

`CheckUpdatable`, which you can customize to override those default behaviors.

## 166 Delphi in Depth: FireDAC

### Index Overview

An index is a data structure that contains pointers to the data in your dataset, which in most cases is about one or more columns. When a dataset has more than one index, each index is based on either a different column, a collection of the dataset's columns, or expressions based on data in the dataset's columns.

The data in each index is ordered, permitting it to be searched very quickly.

Indexes serve five distinct purposes. These are to:

- ▀ Guarantee uniqueness of table data
- ▀ Provide sorted views of data
- ▀ Provide quick searches for records
- ▀ Allow for master-detail joins
- ▀ Support customized views of the data based on expressions or filters

First of all, an index can be used to prevent duplicate data from appearing in a table. This type of index is referred to as a *unique index*. A unique index can prevent duplicate values from being inserted into your dataset, even if the value that you want to assure uniqueness is not part of the primary key.

The second purpose of an index is to provide sorted views of the data in a dataset. The index does not actually change the order of the records in a dataset's data store, but instead causes those records to behave as if they were ordered based on the index.

The third purpose of an index is to quickly locate one or more records based on data in the indexed fields. These indexes may or may not be unique indexes, depending on the fields used to define the index. Since indexes are already sorted, attempting to located records based on the one or more fields of an index means that the data can be searched very quickly in order to locate the

associated records. Searching datasets is discussed in detail in *Chapter 8, Searching Data*.

The fourth purpose of an index is closely related to the second and third purposes. An index can be used to support master-detail joins between data in two related datasets.

Finally, an index can restrict the dataset's result set in a particular view. Specifically, an index can suppress some of the records in a FireDAC dataset, Chapter 7: Creating Indexes 167

based on either the contents of specific fields, or based on criteria applied to fields of the dataset.

In general, the indexes of a FireDAC dataset can be divided into two categories: temporary indexes, and persistent indexes. Persistent indexes can be further categorized as being based on IndexDef collections or FDIndex collections.

Each of these index types is discussed in the following sections.

## **Temporary Indexes**

Temporary indexes are created with the IndexFieldNames property. To create a temporary index, set the IndexFieldNames property to the name of the field or fields on which you want to base the index. When you need a multi-field index, separate the field names with semicolons.

*Note: The name temporary index is derived from the ClientDataSet class, which introduced the IndexFieldNames property. In the ClientDataSet, the index created when you assign a string to IndexFieldNames is transient, in that it is destroyed when you assign a different string to the IndexFieldNames property.*

*FireDAC datasets, by comparison, do not destroy an existing index created with IndexFieldNames when you change the value of this property. Nonetheless, I have opted to continue to refer to indexes created using IndexFieldNames as temporary indexes.*

Fields in a temporary index can be sorted in ascending order (the default) or descending order. In order to sort a field in descending order, follow the field name with a colon followed by a modifier, the letter D.

For example, imagine that you have an FDQuery that contains customer records, including account number, first name, last name, city, state, and so on. If you want to sort this data by last name and first name (and assuming that these fields are named FirstName and LastName, respectively), you can create a temporary index by setting the FDQuery's IndexFieldNames property to the following

string:

LastName;FirstName

If you want to sort by LastName in descending order followed by FirstName in ascending order, you would use this string:

168 Delphi in Depth: FireDAC

LastName:D;FirstName;

In addition to the letter D, FireDAC supports two additional modifiers. For example, you can explicitly specify an ascending order by following the field name with a colon followed by the letter A. Since an ascending sort is the default, this modifier doesn't actual change any behavior. Similarly, temporary indexes are case insensitive. However, you can specify case insensitivity by following the field name with a colon and the letter N.

As with all published properties, setting IndexFieldNames can be done at design time, or it can be done in code at runtime using a statement similar to the following:

```
FDQuery1.IndexFieldNames := 'LastName:D;FirstName';
```

When you assign a value to the dataset's IndexFieldNames property, the dataset immediately generates the index if it does not already exist. If the contents of the data are being displayed, once the index has been created, those records will appear sorted based on the fields of the index, with the first field in the index sorted first, followed the second field (if present), and so on.

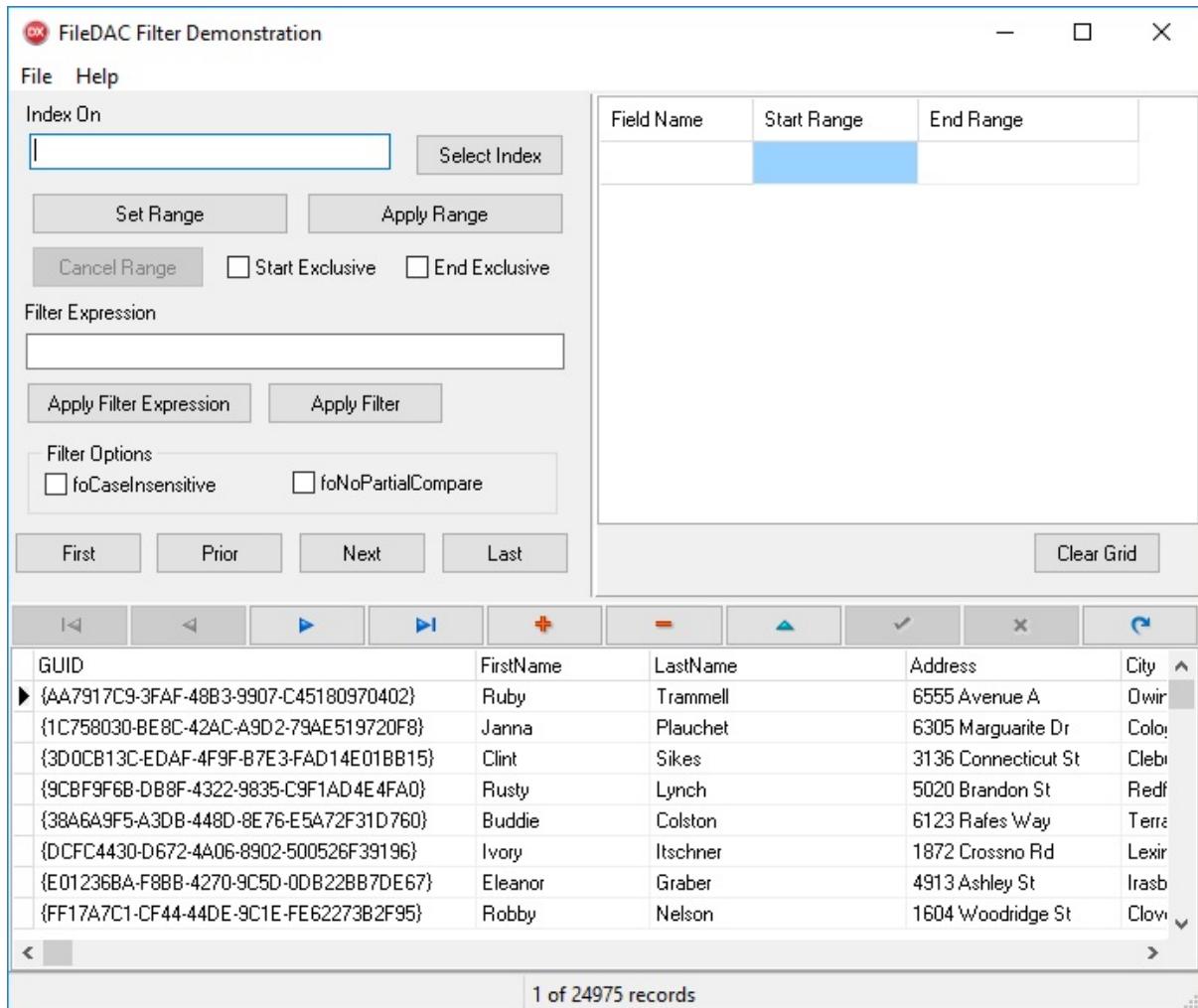
Once the index is active, it is maintained. For example, each time you insert, update, or delete data from the associated dataset, the index is updated to reflect these changes.

*Code: The project FDFilter is in the code download. See Appendix A for details.*

### An Example of Creating Temporary Indexes at Runtime

The following code demonstrates one technique that you might be able to adapt that permits the end user to select their own index. This code comes from the FDFilter project. The main form for the FDFilter project is shown in Figure 7-1.

The Edit labeled Index On can hold an index definition, but we should not permit the user to define their index without verifying that it is valid. As a result, this Edit is readonly.



## Chapter 7: Creating Indexes 169

**Figure 7-1: The main form of the FDFilter project. The button labeled “Select Index” permits a user to choose their own index at runtime** In order to enter an index, the user clicks the Select Index button. This button is associated with the OnClick event handler that appears at the top of the following code segment. This event handler calls the GetTemporaryIndexFromUser custom method, passing in the currently selected index. If GetTemporaryIndexFromUser does not raise an exception, it returns a validated string that can be used to request a temporary index. At this point, the value is assigned to the Edit, which in turn triggers an OnChange event for the Edit. The Edit’s OnChange event assigns the value in the Edit to the

FDMemTable’s IndexFieldNames property.

The real work is performed by GetTemporaryIndexFromUser, and this code is shown below. This method begins by displaying an InputQuery dialog box,

requesting the index text from the user. (A custom dialog box could make this  
170 Delphi in Depth: FireDAC

significantly more convenient in that it could actually contain a list of indexable fields, and offer drag-and-drop configuration, but it would make the example much more complicated.) If the user enters a new field or field list, the code strips off unwanted spaces, verifies that the fields are valid, and then returns the validated string. If the user does not accept the InputQuery dialog box, the original index string is returned. If the user enters at least one invalid field name or invalid character, an exception is raised and the index is unchanged.

```
procedure TForm1.SelectIndexBtnClick(Sender: TObject);
var
  NewIndexFields: String;
begin
  NewIndexFields :=
    GetTemporaryIndexFromUser(IndexFieldNamesEdit.Text);
  if NewIndexFields <> IndexFieldNamesEdit.Text then
    begin
      IndexFieldNamesEdit.Text := NewIndexFields;
      ClearGridBtnClick(Sender);
    end;
  end;
procedure TForm1.IndexFieldNamesEditChange(Sender: TObject);
begin
  Start;
  FDMemTable1.IndexFieldNames := IndexFieldNamesEdit.Text;
  Done;
  UpdateGridLabels;
end;
function TForm1.GetTemporaryIndexFromUser(
  CurrentIndex: String): String;
```

```
//local function to verify field list
function FieldsValid(FDDataSet: TFDataSet;
FieldList: String): Boolean;
var
i: Integer;
SList: TStringList;
s1, s2: string;
begin
Result := False;
SList := TStringlist.Create;
SList.Delimiter := ‘;’;
SList.DelimitedText := FieldList;
Chapter 7: Creating Indexes 171
try
if SList.Count = 0 then
begin
Exit( True );
end
else
for s1 in SList do
begin
if UpperCase(s1).EndsWith(‘:D’) then
s2 := copy( s1, 1, Length(s1) - 2 )
else
s2 := s1;
if FDDataSet.FindField(s2) = nil then
exit( False );
end;
Result := True;
```

```

finally
  SList.Free;
end;
end; //function FieldsValid

begin
  Result := IndexFieldNamesEdit.Text;
  if InputQuery('Enter index field(s) [ex: field1;field2:D]', 'Index on', Result) then
    begin
      while Pos(' ',Result) <> 0 do
        Result := StringReplace(Result, ' ',
        '', [rfReplaceAll]);
      if not FieldsValid(FDMemTable1, Result) then
        raise EBadFieldInIndex.Create('IndexFieldNames ' +
        'contains at least one invalid field name');
    end; //if InputQuery
  end;

```

### **Temporary Indexes and FormatOptions.SortOptions**

You can have some impact on the temporary index using the FormatOptions.SortOptions property. For example, if you want all of the fields in the IndexFieldNames property to sort by descending order, add the soDescending flag to the FormatOptions.SortOptions property. Adding this flag will sort all fields in descending order (even you include the :A modifier in the IndexFieldNames string).

#### 172 Delphi in Depth: FireDAC

In addition, adding the soUnique flag will ensure that the combination of values in the fields of the index are unique, which will cause an exception to be raised when the index is applied if the data already includes duplicate values.

Unfortunately, the soNoCase flag default value is False, which should produce a case-sensitive index, but temporary indexes are case insensitive by default. I have not been able to create a case-sensitive index using temporary

indexes.

Temporary indexes are extremely useful in a number of situations, such as in the preceding example where you want to permit your users to sort the data based on any field or field combination. There are, however, some drawbacks to

temporary indexes.

Specifically, temporary indexes do not support more advanced index options, such as distinct indexes, expression indexes, and filter-based indexes. If you need some of these more sophisticated features, you will need to create persistent indexes.

## Persistent Indexes

Persistent indexes, when created, are similar to temporary indexes in many ways. Unlike temporary indexes, though, they support a variety of options not available for temporary indexes.

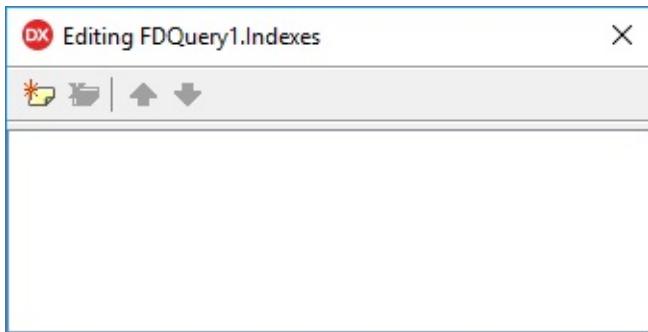
All FireDAC datasets support persistent indexes based on a collection of FDIndex objects, each of which can represent the properties that define the associated index. The FDIndex collection can be accessed through the Indexes property of the FireDAC datasets to which the individual indexes belong.

FDMemTables support a second type of persistent index based on IndexDefs (a collection of index definitions). IndexDefs support most, but not all of the features supported by the FDIndex class, and are exposed by FDMemTables in order to provide source code compatibility with the ClientDataSet class, which only supports persistent indexes through IndexDefs.

In this chapter, I am going to focus on creating persistent indexes using the Indexes property of FireDAC datasets. I have made this decision for several reasons. First, all FireDAC datasets support a published Indexes property, while only the FDMemTable exposes the IndexDefs property as a published property.

Second, FDIndex instances support all of the features supported by IndexDefs, but IndexDefs do not support all of the features of FDIndex instances. Finally, even when you are using an FDMemTable, you can use IndexDefs or Indexes,

but not both. As a result, the only reason to use IndexDefs with an FDMemTable is to provide source code compatibility with ClientDataSets. For



## Chapter 7: Creating Indexes 173

this reason, I will not be covering IndexDefs in this book, though you can apply much of what you learn about FDIndexes here to the use of IndexDefs, if

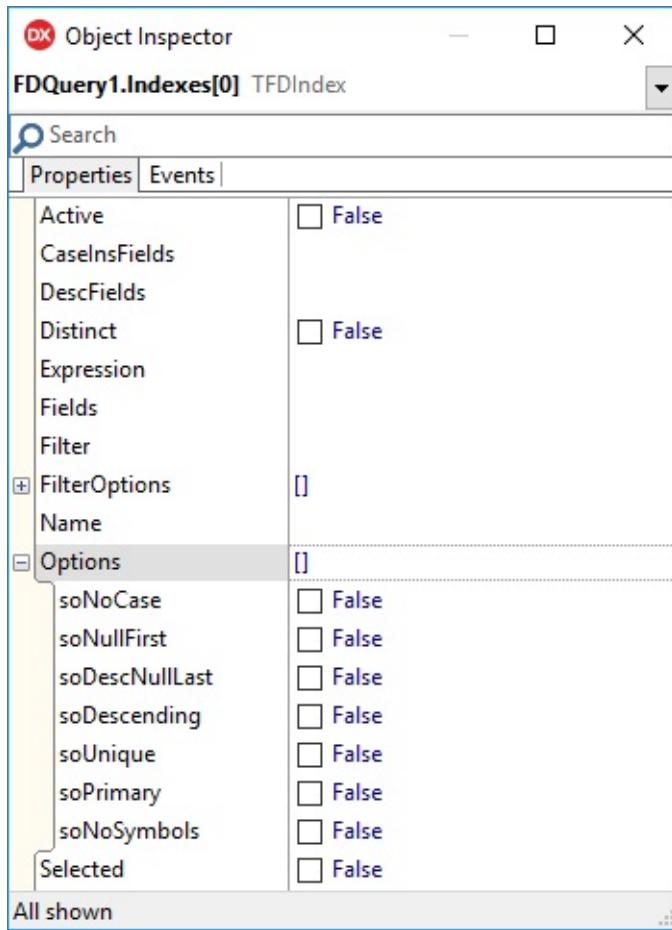
necessary.

### Defining Persistent Indexes

Persistent indexes are defined at design time using the Indexes collection property editor of the FireDAC dataset. To display this collection editor, select the Indexes property of a FireDAC dataset in the Object Inspector and click the ellipsis button that appears.

Click the Add New button on the Indexes collection editor toolbar (or press the Ins key) once for each persistent index that you want to define for a FireDAC

dataset. Each time you click the Add New button (or press Ins), a new FDIndex is created. Complete the index definition by selecting each FDIndex in the Indexes collection editor, one at a time, and configure it using the Object Inspector. The Object Inspector with an FDIndex selected is shown in Figure 7-2. Note that the Options property has been expanded to show its various flags.



## 174 Delphi in Depth: FireDAC

**Figure 7-2: A new FDIndex selected in the Object Inspector**

*Code: This sample code is found in the FDIndexes project of the code download.*

At a minimum, you must define what to index. There are two ways to do this. The first is to set the Fields property of an FDIndex to the name of the field or fields to be indexed. Similar to how you use the IndexFieldNames property, if you are building a multi-field index, you separate the field names with semicolons. You cannot include virtual fields, such as Calculated or Aggregate fields, in an index, though you can use InternalCalc virtual fields in an index.

Also similar to IndexFieldNames, the Fields property of a FireDAC dataset can include the :D modifier following a particular field name to sort that field in descending order. You can also include :A and :N modifiers after one or more fields, but as I described earlier, these do not appear to have an effect.

The second way to define the index is to set the Expression property. The expression property can contain any value expression, which may include references to fields, constants, FireDAC scalar functions, and expression engine functions. When using an expression index, a value is calculated for each record in the result set using that expression, and it is this value that serves as the basis for the index.

You can use the Fields property or the Expression property, but not both. If you have one of the properties set to a value, say Fields, and then assign a value to the other (Expression in this case), the other property is set to an empty string.

By default, FDIndexes are ascending indexes. If you want the index to be a descending index, set the soDescending flag in the Options property.

Alternatively, you can set the DescFields property to a semicolon-separated list of the fields that you want sorted in descending order. DescFields provides an alternative to using the :D modifier in the Fields property.

As you might imagine, you use the DescFields property only when the Fields property is used to define the index, and the DescFields property is limited to including only fields that appear in the Fields property. So long as the

soDescending flag is absent from the Options property of the FDIndex, fields that appears in the DescFields property will be sorted in descending order, and the remaining fields of the Fields property that do not appear in DescFields will be sorted in ascending order.

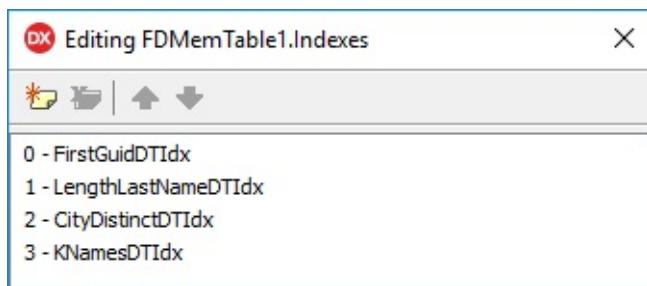
The following sections demonstrate the creation of four types of indexes, both at design time and at runtime. These indexes include basic field-based indexes, expression-based indexes, distinct indexes, and finally, filter-based indexes.

The examples given here make use of the FDIndexes project, found in the code download. In this project, you will find a number of indexes that were defined at design time, along with code that defines additional indexes at runtime.

In the following sections, I walk you through the creation of indexes at design time. If you want to follow these steps, I suggest that you make a copy of the FDIndexes project, and place it in a directory at the same level as FDIndexes.

For example, if you have extracted the sample code files and stored them in a directory named FireDAC on your C drive, the FDIndexes project will be located in the following directory path:

C:\FireDAC\FDIndexes



## 176 Delphi in Depth: FireDAC

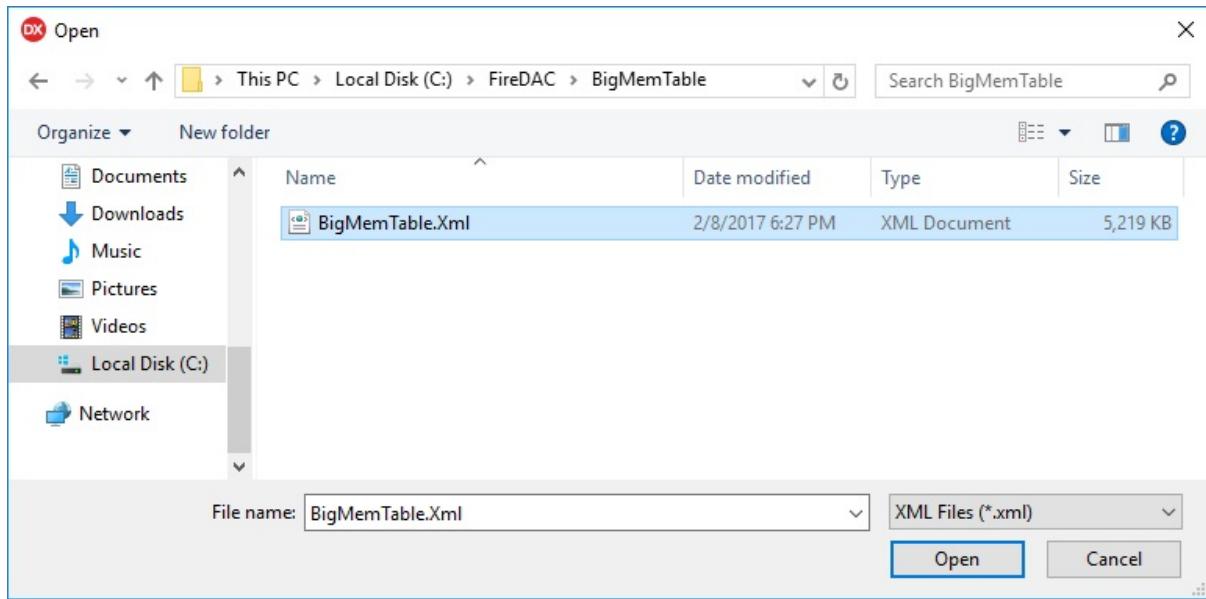
Create a new directory under FireDAC named FDIndexesCopy and copy the files from C:\FireDAC\FDIndexes, and paste those copied files in C:\FireDAC\FDIndexesCopy.

Creating a copy of the FDIndexes project, and placing it in a directory parallel to the original files is essential, in that it will permit the code to find the sample FDMemTable table that it needs for data. This file is in a directory called BigMemTable, and the corresponding file contains close to 25,000 records.

Once you have copied the original files to the new directory, open the copied project and select the FDMemTable. Select the Indexes property in the Object Inspector and double-click the ellipsis button to display the existing indexes in the Indexes collection editor, as shown here:

To delete these indexes, right-click inside the collection editor and choose Select All from the displayed context menu. Next, press Del to delete these existing indexes. You might also have to delete the generated IndexDefs as well. You are now ready to proceed with the hands-on examples described in the following sections.

Before you start, however, you will want to use the FireDAC FDMemTable to load the BigMemTable table from a file. To do this, right-click the FDMemTable and select Load From File from its displayed context menu. Navigate to the BigMemTable directory, which should be parallel to your current directory. Select the BigMemTable.Xml file, as shown in Figure 7-3, and click open.



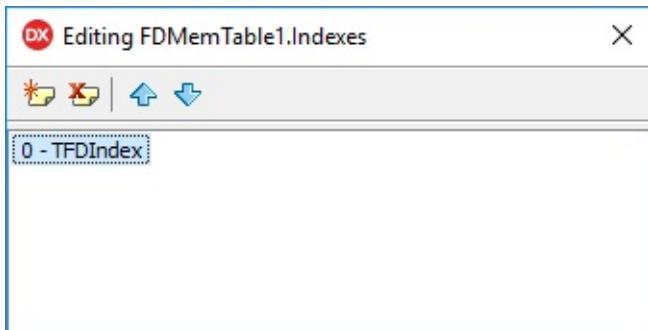
GUID	FirstName	LastName	Address
{AA7917C9-3FAF-48B3-9907-C45180970402}	Ruby	Trammell	6555 Avenue A
{1C758030-BE8C-42AC-A9D2-79AE519720F8}	Janna	Plauchet	6305 Marguarite
{3D0CB13C-EDAF-4F9F-B7E3-FAD14E01BB15}	Clint	Sikes	FDMemTable1
{9CBF9F6B-DB8F-4322-9835-C9F1AD4E4FA0}	Rusty	Lynch	5020 Brandon S
{38A6A9F5-A3DB-448D-8E76-E5A72F31D760}	Buddie	Colston	6123 Rafes Wa
{DCFC4430-D672-4A06-8902-500526F39196}	Ivory	Itschner	1872 Crossno R
{E01236BA-F8BB-4270-9C5D-0DB22BB7DE67}	Eleanor	Graber	4913 Ashley St
{FF17A7C1-CF44-44DE-9C1E-FE62273B2F95}	Robby	Nelson	1604 Woodridge
{A59072CA-0E7A-486C-B16C-FC3BB6685A48}	Anne	Sorror	4006 Waco St
{69CC0E85-114D-4062-B0B1-7E4DCDE22A3C}	Griselda	McCleer	FDStanStorageXMLLink1400 Tennyson
{2128A062-6E91-4C68-8320-4130258DED2F}	Jimmy	McIntyre	852 Clifton St

Chapter 7: Creating Indexes 177

**Figure 7-3: Select the BigMemTable.XML file to load it into the FireDAC FDMemTable**

At this point the FDMemTable should load the data from BigMemTable.xml, and your form should look like that shown in Figure 7-4.

**Figure 7-4: The FDIndexes project in the Delphi form designer**



## 178 Delphi in Depth: FireDAC

### Creating Field-Based Indexes

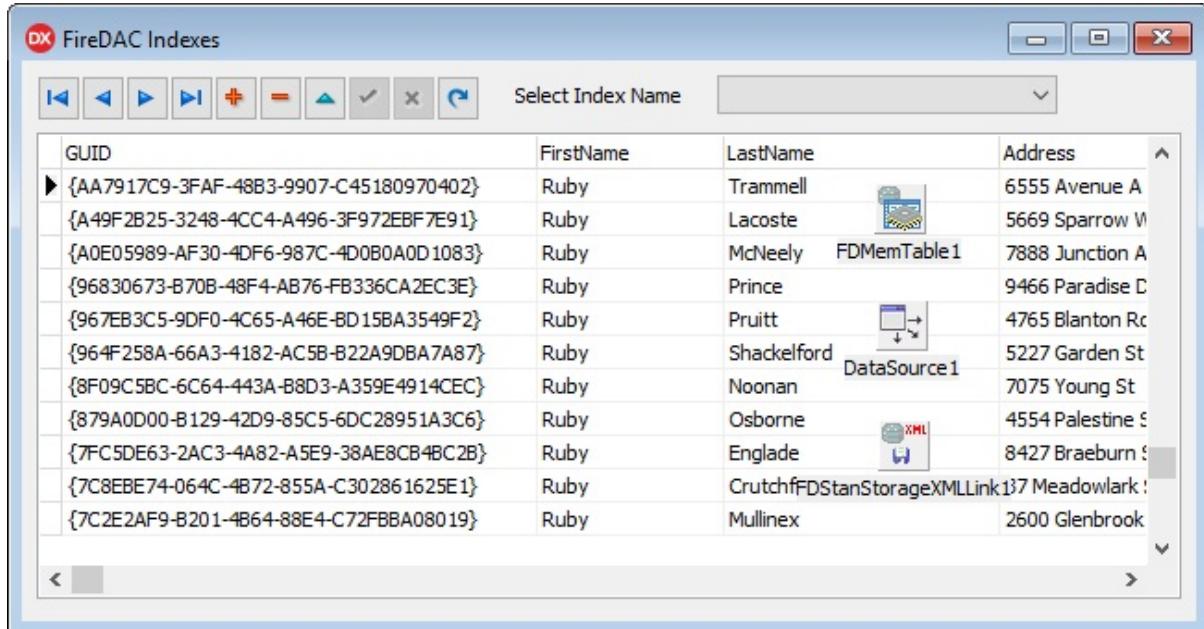
The most straightforward indexes, and the ones that most developers think about when creating indexes, are those based on one or more fields. You create field-based indexes by creating an FDIndex and then specifying the index using the Fields property. These types of indexes may also make use of the DescFields property, or the :D modifier in the Fields property if you want to create a multi-field index where some fields are sorted in ascending order and others are sorted in descending order.

### CREATING A FIELD-BASED INDEX AT DESIGN TIME

The following steps will walk you through the creation of a field-based index at design time.

1. Select the FDMemTable on the main form. Using the Object Inspector, select the Indexes property and then click the ellipsis that appears to display the Indexes collection editor. If you have followed the instructions given above to remove the existing indexes, this collection editor should be empty.
2. Click the Add New button (or press Ins) to create a new FDIndex instance in the Index collection. The collection editor should now look something like this:
3. Using the Object Inspector, set the Fields property to FirstName;Guid. You can either type or choose the fields from the Field List Editor by clicking the ellipsis. Note that when using the Fields List Editor, the order in which you choose the fields is important.
4. Now enter Guid in the DescFields property of the FDIndex.
5. Next, create a name for the index. In the Name field, enter

FirstGuidDTIdx. The DT part is meant to refer to this as a design-time created index, and I like to add the suffix Idx to indexes as a convention.



## Chapter 7: Creating Indexes 179

6. The final step is to set the Active property of this index to True. This will inform FireDAC that it can select this index. If you try to use the index without setting its Active property to True, FireDAC will throw an exception if you try to set the FDMemTable's IndexName property to the name of this index.

### **SELECTING AN INDEX AT DESIGN TIME**

Once you have defined an index, you can assign the name of that index to the IndexName property of the FireDAC dataset. If the dataset is not active, that index will not be created, and applied, until you subsequently make the dataset active.

On the other hand, if you have made the FireDAC dataset active at design time, assigning the name of the Index to the IndexName property of the dataset will cause that index to be constructed and then applied. This is demonstrated in the following steps:

1. Select the FDMemTable.
2. Use the Object Inspector to set the IndexName property of the FDMemTable to FirstGuidDTIdx. After a brief moment, FireDAC will display the records in a sort order based on the index, where the

FirstName field is first sorted in ascending order, following by GUID in descending order. Your form might look something like that shown in Figure 7-5.

**Figure 7-5: The FirstGuidDTIdx index sorts by first name, in ascending order, followed by GUID, in descending order**

180 Delphi in Depth: FireDAC

## **CREATING A FIELD-BASED INDEX AT RUNTIME**

The process of creating an index at runtime is essentially the same as that used at design time. You create a new FDIndex instance, define the necessary properties to make it a valid index definition, and then set its Active property to True. After that, it is possible to set the IndexName property of the corresponding FireDAC dataset to this index name to make it the current index.

You create a new FDIndex instance by calling the Add method of the Indexes property of the FireDAC dataset. This method returns an FDIndex instance. You then proceed in the same manner as you did at design time. This is demonstrated in the following code segment, which creates an index at runtime:

```
var  
  Index: TFDIndex;  
begin  
  // A field-based index  
  Index := FDMemTable1.Indexes.Add;  
  Index.Name := 'LastGuidRTIdx';  
  Index.Fields := 'Last_Name;GUID:D';  
  Index.Active := True;
```

In this case, I did not use the DescFields property, but instead decorated the GUID field name in the Fields property with the :D modifier. Both techniques produce the same results, which is to sort the GUID field in descending order.

## **SELECTING AN INDEX AT RUNTIME**

You select an active FDIndex at runtime using the same technique as you do at design time. Specifically, you assign the name of the index to the IndexName property of the FireDAC dataset. To assist you, the FDIndex class

includes a small collection of methods that can help you find your index. These include the `FindIndex`, `IndexByName`, and `FindIndexForFields` methods, as well as the

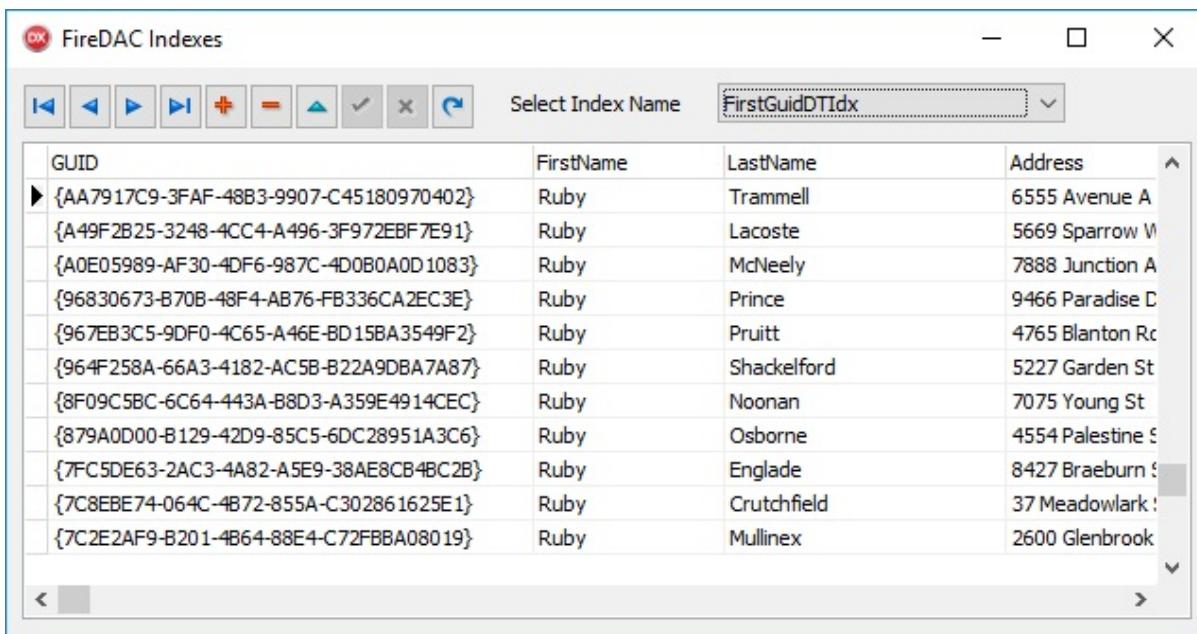
Items property.

The FDIndexes project includes a little code that iterates through all currently defined indexes at runtime, populating a combobox with the names of the available, active indexes. This code, which is called from the `OnCreate` event handler of the form, is shown here:

```
procedure TForm1.ListIndexes;
```

```
var
```

```
i: Integer;
```



## Chapter 7: Creating Indexes 181

```
begin
```

```
for i := 0 to FDMemTable1.Indexes.Count - 1 do
```

```
if FDMemTable1.Indexes[i].Active then
```

```
  cbxIndexes.Items.Add( FDMemTable1.Indexes.Items[i].Name );
```

```
  cbxIndexes.ItemIndex := -1;
```

```
end;
```

Here is the `OnChange` event handler of the index combobox:

```
procedure TForm1.cbxIndexesChange(Sender: TObject);
```

```
begin  
if cbxIndexes.ItemIndex <> -1 then  
  FDMemTable1.IndexName := cbxIndexes.Text;  
end;
```

Figure 7-6 shows is how the form looks at runtime after one of the active indexes has been selected.

### **Figure 7-6: An active index has been selected from the combobox, resulting in a sorting of the record shown on the form**

In the remaining sections of this chapter, I demonstrate three more index types: Expression indexes, distinct indexes, and filter indexes. In each of these

182 Delphi in Depth: FireDAC

sections, I walk you through the process of creating an index at design time, and then show the code that creates the corresponding index at runtime.

### **Creating Expression Indexes**

Unlike field-based indexes, which are largely based on the data in the columns of the FireDAC dataset, an expression index is based on an expression. That expression can include field names, constants, expression engine functions, FireDAC scalar functions, and complex expressions involving one or more of these combined with operators. While a simple expression might include only a field name, say for example, FirstName, the real power of an expression index is that the expression is calculated separately for each record, after which, the dataset is ordered by the expression.

FireDAC provides support for a large number of scalar functions that you can use in expressions, in filters, and in the SQL command preprocessor. These include string functions, such as SUBSTRING(), UPPER CASE(), and LENGTH(), and math and arithmetic functions, such as SIN(), LOG(), and RAND() (random number). Date and time functions include CURRENT\_DATE() and DAYOFWEEK(), and system functions include DATABASE() (the name of the database to which you are connected), NEWGUID(), and IIF (immediate if).

*Note: In order to use scalar functions from the FireDAC expression engine at runtime, you must add the FireDAC.Stan.ExprFuncs unit to at least one of*

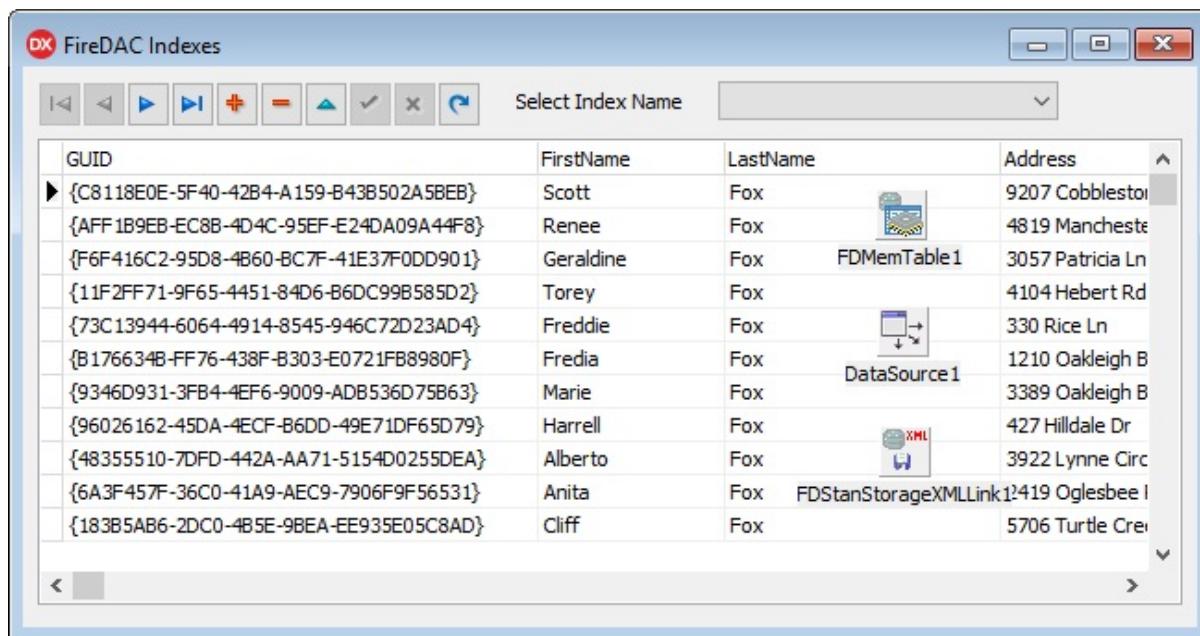
your project's uses clauses. Also, if you find that some of these functions, such as LENGTH, raise an exception when you use them against the InterBase database, see Appendix A for information on correcting those errors.

For a complete list of the FireDAC scalar functions, please refer to Tables 14-3

through 14-6 in *Chapter 14, The SQL Command Preprocessor*.

Use the following steps to create an index that will sort the FDMemTable based on the length of the LastName field:

1. Open the FDMemTable's Indexes collection editor and add a new index.
2. With this new index selected in the Indexes collection editor, set Expression to the following string: LENGTH(LastName).
3. Set IndexName to LengthLastNameDTIdx.
4. Set the Active property to True.



## Chapter 7: Creating Indexes 183

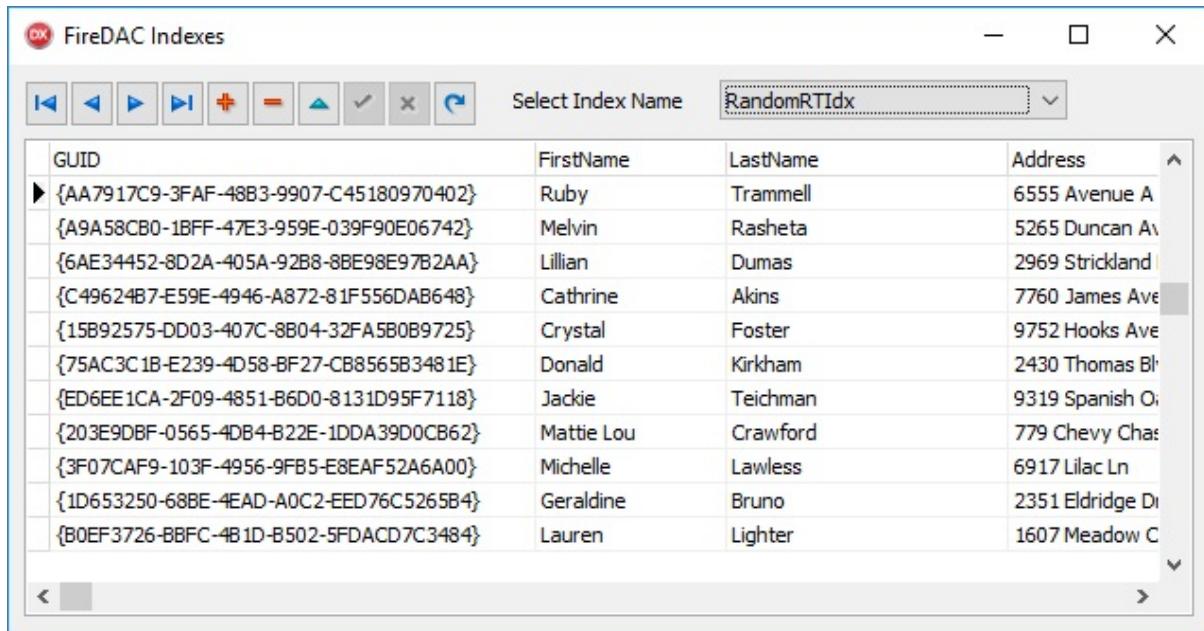
If you now select the FDMemTable and set its IndexName property to LengthLastNameDTIdx, you will see the data sorted by the length of the LastName field, as shown in Figure 7-7.

**Figure 7-7: The data is sorted by the length of the LastName field**

To make things interesting, the following code shows how to create a runtime index that sorts the records randomly. This code will define an expression

index where the rand() expression engine function is seeded with a random number between 0 and MaxInt. Here is the code that creates this index. Figure 7-8 shows the data sorted by this index.

```
// An expression index  
  
Index := FDMemTable1.Indexes.Add;  
  
Index.Name := 'RandomRTIdx';  
  
Index.Expression := 'RAND( ' + RandomRange(0, MaxInt) + ')';  
  
Index.Active := True;
```



The screenshot shows the FireDAC Indexes dialog box. At the top, there is a toolbar with various icons for managing indexes. To the right of the toolbar is a dropdown menu labeled "Select Index Name" with "RandomRTIdx" selected. Below the toolbar is a table with four columns: "GUID", "FirstName", "LastName", and "Address". The table contains 15 rows of data, each with a unique GUID and a different first name, last name, and address. The data is sorted by the "Address" column, which is highlighted in yellow.

GUID	FirstName	LastName	Address
{AA7917C9-3FAF-48B3-9907-C45180970402}	Ruby	Trammell	6555 Avenue A
{A9A58CB0-1BFF-47E3-959E-039F90E06742}	Melvin	Rasheta	5265 Duncan Av
{6AE34452-8D2A-405A-92B8-8BE98E97B2AA}	Lillian	Dumas	2969 Strickland
{C49624B7-E59E-4946-A872-81F556DAB648}	Cathrine	Akins	7760 James Ave
{15B92575-DD03-407C-8B04-32FA5B0B9725}	Crystal	Foster	9752 Hooks Ave
{75AC3C1B-E239-4D58-BF27-CB8565B3481E}	Donald	Kirkham	2430 Thomas Bl
{ED6EE1CA-2F09-4851-B6D0-8131D95F7118}	Jackie	Teichman	9319 Spanish O
{203E9DBF-0565-4DB4-B22E-1DDA39D0CB62}	Mattie Lou	Crawford	779 Chevy Chas
{3F07CAF9-103F-4956-9FB5-E8EAF52A6A00}	Michelle	Lawless	6917 Lilac Ln
{1D653250-68BE-4EAD-A0C2-EED76C5265B4}	Geraldine	Bruno	2351 Eldridge Di
{B0EF3726-BBFC-4B1D-B502-5FDACD7C3484}	Lauren	Lighter	1607 Meadow C

## 184 Delphi in Depth: FireDAC

**Figure 7-8: An expression index is used to randomize a dataset**

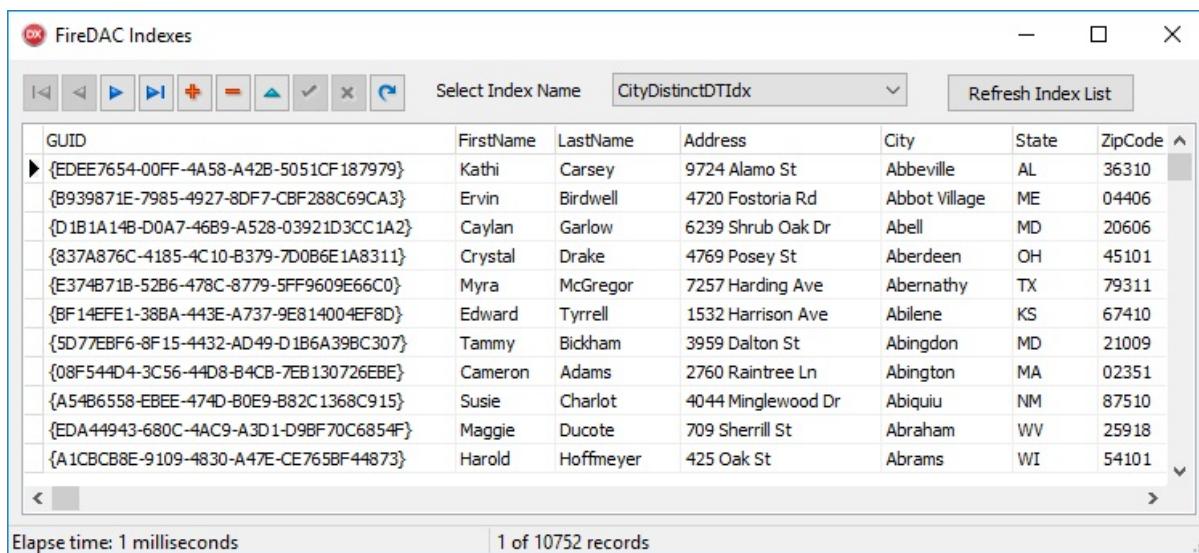
### Creating Distinct Indexes

A distinct index is one that, once applied, results in the display of only unique values in the fields of the index. For example, regardless of how many records there are in the result set, a distinct index placed on the AccountNumber field will result in only one record per account number being displayed. While this might include all records, especially in tables where account numbers must be unique, there are many instances where a distinct index will result in far fewer records being displayed than there are records in the underlying result set.

For example, imagine that you have an FDQuery whose SQL select statement select all Cities, States/Provinces, and Countries from a large database of customers. A distinct index on the Country field will display only one record per country. As a result, even though the SQL statement returns 900 records,

once the distinct index is applied, there may appear to be only 50 records in the result set, one for each country represented in the original result set. A distinct index on both state/province and country is likely to include many more records, but far fewer than 900.

In essence, a distinct index is one that produces a result similar to a SELECT DISTINCT SQL query (or whatever is equivalent in the SQL dialect that you use). Which records will actually represent the *distinct* values in the index fields? That depends on the order of records in the database, as well as any restrictive clauses in the query or stored procedure that produced the result set,



GUID	FirstName	LastName	Address	City	State	ZipCode
{EDEE7654-00FF-4A58-A42B-5051CF187979}	Kathi	Carsey	9724 Alamo St	Abbeville	AL	36310
{B939871E-7985-4927-8DF7-CBF288C69CA3}	Ervin	Birdwell	4720 Fostoria Rd	Abbot Village	ME	04406
{D1B1A14B-D0A7-46B9-A528-03921D3CC1A2}	Caylan	Garlow	6239 Shrub Oak Dr	Abell	MD	20606
{837A876C-4185-4C10-B379-7D0B6E1A8311}	Crystal	Drake	4769 Posey St	Aberdeen	OH	45101
{E374B71B-52B6-478C-8779-5FF9609E66C0}	Myra	McGregor	7257 Harding Ave	Abernathy	TX	79311
{BF14FEE1-38BA-443E-A737-9E814004EF8D}	Edward	Tyrrell	1532 Harrison Ave	Abilene	KS	67410
{5D77EBF6-8F15-4432-AD49-D1B6A39BC307}	Tammy	Bickham	3959 Dalton St	Abingdon	MD	21009
{08F544D4-3C56-44D8-B4CB-7EB130726EBE}	Cameron	Adams	2760 Raintree Ln	Abington	MA	02351
{A54B6558-EBEE-474D-B0E9-B82C1368C915}	Susie	Charlot	4044 Minglewood Dr	Abiquiu	NM	87510
{EDA44943-680C-4AC9-A3D1-D9BF70C6854F}	Maggie	Ducote	709 Sherrill St	Abraham	WV	25918
{A1C8CB8E-9109-4830-A47E-CE765BF44873}	Harold	Hoffmeyer	425 Oak St	Abrams	WI	54101

Elapsed time: 1 milliseconds

1 of 10752 records

## Chapter 7: Creating Indexes 185

such as predicates in an SQL SELECT WHERE clause that returns only those records that match some criteria.

Use the following steps to create a distinct index using the FDIndexes project:

1. Create a new index for the FDMemTable.
2. With your newly created index selected in the Indexes collection editor, use the Object Inspector to set Fields to City.
3. Next, set the Name property of the FDIndex to CityDistinctDTIdx.
4. Now set the Distinct property of the FDIndex to True.
5. Finally, set the Active property to True.

If the FDMemTable were active, and you were to set the IndexName property to CityDistinctDTIdx, the FDMemTable would display only one record for each

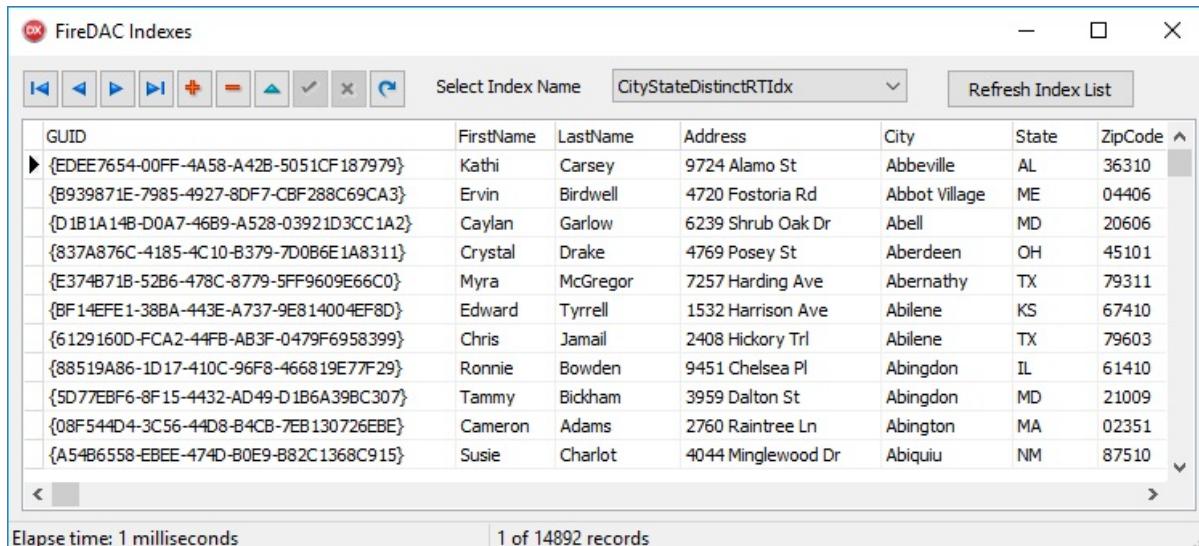
different City field value in the dataset, as shown in Figure 7-9.

### Figure 7-9: A distinct index is displaying 10752 unique city names in the BigMemTable database

The FDIndexes project includes code that creates a distinct index at runtime.

This code creates an index that displays unique city/state combinations, as you can see in the following code segment:

```
// A distinct index  
Index := FDMemTable1.Indexes.Add;  
Index.Name := 'CityStateDistinctRTIdx';
```



The screenshot shows a Windows application window titled "FireDAC Indexes". At the top, there is a toolbar with various icons for navigating through the index list. To the right of the toolbar is a dropdown menu labeled "Select Index Name" which is currently set to "CityStateDistinctRTIdx". Further to the right is a "Refresh Index List" button. Below the toolbar is a table with columns: GUID, FirstName, LastName, Address, City, State, and ZipCode. The table contains 14,892 records. The first few records are listed below:

GUID	FirstName	LastName	Address	City	State	ZipCode
{EDEE7654-00FF-4A58-A42B-5051CF187979}	Kathi	Carsey	9724 Alamo St	Abbeville	AL	36310
{B939871E-7985-4927-8DF7-CBF288C69CA3}	Ervin	Birdwell	4720 Fostoria Rd	Abbot Village	ME	04406
{D1B1A14B-D0A7-46B9-A528-03921D3CC1A2}	Caylan	Garlow	6239 Shrub Oak Dr	Abell	MD	20606
{837A876C-4185-4C10-B379-7D0B6E1A8311}	Crystal	Drake	4769 Posey St	Aberdeen	OH	45101
{E374B71B-52B6-478C-8779-5FF9609E66C0}	Myra	McGregor	7257 Harding Ave	Abernathy	TX	79311
{BF14EFE1-38BA-443E-A737-9E814004EF8D}	Edward	Tyrrell	1532 Harrison Ave	Abilene	KS	67410
{6129160D-FCA2-44FB-AB3F-0479F6958399}	Chris	Jamail	2408 Hickory Trl	Abilene	TX	79603
{88519A86-1D17-410C-96F8-466819E77F29}	Ronnie	Bowden	9451 Chelsea Pl	Abingdon	IL	61410
{5D77EBF6-8F15-4432-AD49-D1B6A39BC307}	Tammy	Bickham	3959 Dalton St	Abingdon	MD	21009
{08F544D4-3C56-44D8-B4CB-7EB130726EBE}	Cameron	Adams	2760 Raintree Ln	Abington	MA	02351
{A54B6558-EBEE-474D-B0E9-B82C1368C915}	Susie	Charlot	4044 Minglewood Dr	Abiquiu	NM	87510

At the bottom left of the window, it says "Elapse time: 1 milliseconds". At the bottom center, it says "1 of 14892 records".

### 186 Delphi in Depth: FireDAC

```
Index.Fields := 'City;State';
```

```
Index.Distinct := True;
```

```
Index.Active := True;
```

Figure 7-10 shows this index selected in the running project. Since there is the possibility that two or more states share cities with the same names, this view contains more records than the previous figure, almost 15,000, compared to just under 11,000 in Figure 7-9.

### Figure 7-10: A distinct index is displaying only one record for each unique city/state field combination

#### Creating a Filter-Based Index

I am going to discuss filters in more detail in a following chapter, *Chapter 9, Filtering Data*. Many searches and filters require indexes, and this type of index employs a filter. Since I have decided to discuss indexes before filters, I

will discuss the filter aspect of these indexes only in passing here. If you find this topic particularly interesting, or have never used filters before, you may want to skip ahead to Chapter 9 to read the discussion of filters before continuing with this topic.

As described earlier in this chapter, an index is a structure that contains an ordered set of pointers to the records of your FireDAC dataset, and it is often used to sort, search, and filter the records in your dataset. A filter-based index contains a structured set of record pointers, but like the distinct indexes discussed in the previous section, not every record is included in the structure.

## Chapter 7: Creating Indexes 187

In short, a filter limits which records are displayed to some subset of records based on the data appearing in the fields of the dataset. For example, a filter may be used to show only records where account holders have credit limits that exceed some value, or whose invoices are past due greater than some number of days. As a result, filters permit you to select some larger set of data from your underlying database, and then manipulate which records are displayed or

navigable without have to re-execute the query that originally obtained the data.

When an index employs a filter expression, selecting that index does two things.

It orders the records based on the index, and it limits which records are accessible based on the filter.

Filter-based indexes are unique to FireDAC. In most other Delphi data-access frameworks, when you want to place a filter, you either do so without an index, which can be relatively slow, or you first apply an index and then use one of the filtering mechanisms that are index based. This second approach is almost

universally faster than filters that do not require an index.

Since FireDAC has introduced filter-based indexes, it provides for very fast filtering based on the index. As a result, the filters are applied with fewer lines of code and with ultra-fast results.

Use the following steps to create a filter-based index:

1. Create a new index for the FDMemTable.
2. With your newly created index selected in the Indexes collection editor, use the Object Inspector to set Fields to FirstName.

3. Select Filter and enter the following expression: [FirstName] = 'K\*'.
4. Expand the FilterOptions property and check the ekPartial flag.
5. Set the DescFields property to FirstName.
6. Finally, set the Name property to KNamesFilterDTIdx.

This filter, when applied, will display only records where the FirstName field begins with the letter K, and the names will be in descending order. Figure 7-11

shows how the FDIndexes project looks when this filter is selected at design time.

GUID	FirstName	LastName	Address	City
{BF365130-5DD6-46B8-ABD0-4D156C01CF58}	Kal	Thornton	4921 Dinah Dr	Fouke
{50416FE5-6CCA-4B42-A1D8-91F2882C1A42}	Kal	Maes	294 Arbor Ct	Boston
{C19EDA36-D819-44BF-97F4-F710544EB1AC}	Kal	Seeley	2591 Ranchland Dr	Newburg
{5F83CF64-4560-4340-A2BE-8253927DCD30}	Kal	Swanson	5676 Meadowridge Dr	El Dorado Spr
{12975C0A-ACC2-41C1-8DC2-39242CB49EB5}	Kal	Tucker	9820 Jefferson Dr	Lincoln Park
{06BE7D0D-F511-46BD-8198-9ABBD570E98D}	Kal	Shores	9473 Atlanta Ave	Sagle
{10DE82AF-D832-41AF-BE74-BC2A184888F8}	Kal	Creed	549 Inverness Dr	Gail
{D23C76EA-1237-4F27-8019-393001C1F371}	Kal	Eddington	2649 King George Dr	Regan
{5A392325-072D-45AB-AA36-28F2CCB800B7}	Kal	Sockler	925 Bedford Dr	Garrison
{0353AD96-E313-444D-AB4F-09D7DA75DD7A}	Kal	Trant	1859 Verde St	Ridge Manor
{86851458-187F-4FCD-8AC3-1913B8804046}	Kal	Sumpter	2161 Oakleigh Blvd	Merricourt

## 188 Delphi in Depth: FireDAC

**Figure 7-11: A filter-based index at design time**

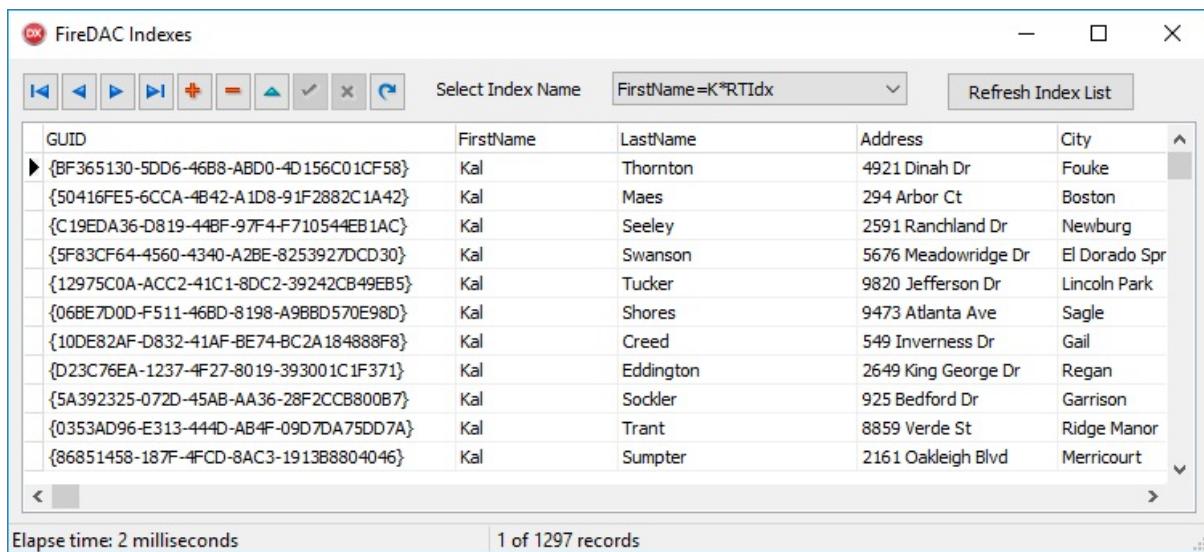
*Note: Unfortunately, I encountered issues when selecting this index at runtime using the combobox I provided in this sample project. FireDAC reported that the index was not a sorted view. This is clearly a bug, since we can both select the index at design time, as well as create and use a similar index at runtime (see the following discussion). Hopefully this bug has been fixed in the version of FireDAC that you are using.*

My example of a runtime filter-based index is more involved, in that it queries the user for the pattern on which to filter. To make things simple, I am using the InputQuery method to display a dialog into which the user can enter data. The following code segment shows how this filter is constructed, and Figure 7-12

shows how the FDIndexes project looks after I entered K\* in response to the InputQuery dialog box. This index essentially reproduces the design time index you created in the preceding section, with the exception that it actually

works here.

```
procedure TForm1.cbxIndexesChange(Sender: TObject);
var
  s: string;
  Index: TFDIndex;
begin
  if cbxIndexes.ItemIndex <> -1 then
  begin
    if cbxIndexes.Text =
      'Create a runtime filter-based index' then
```



The screenshot shows a software interface titled "FireDAC Indexes". At the top, there is a toolbar with various icons for navigating through the index list. To the right of the toolbar is a search bar labeled "Select Index Name" containing the text "FirstName=K\*RTIdx". Next to it is a "Refresh Index List" button. Below the toolbar is a table with five columns: "GUID", "FirstName", "LastName", "Address", and "City". The table contains 1297 records, with the first few rows visible. The data includes names like Kal Thornton, Maes, Seeley, Swanson, Tucker, Shores, Creed, Eddington, Sockler, Trant, and Sumpter, along with their addresses and cities. At the bottom of the interface, there is a status bar showing "Elapse time: 2 milliseconds" and "1 of 1297 records".

GUID	FirstName	LastName	Address	City
{BF365130-5DD6-46B8-ABD0-4D156C01CF58}	Kal	Thornton	4921 Dinah Dr	Fouke
{50416FE5-6CCA-4B42-A1D8-91F2882C1A42}	Kal	Maes	294 Arbor Ct	Boston
{C19EDA36-D819-44BF-97F4-F710544EB1AC}	Kal	Seeley	2591 Ranchland Dr	Newburg
{5F83CF64-4560-4340-A2BE-8253927DCD30}	Kal	Swanson	5676 Meadowridge Dr	El Dorado Spr
{12975C0A-ACC2-41C1-8DC2-39242CB49EB5}	Kal	Tucker	9820 Jefferson Dr	Lincoln Park
{06BE7D0D-F511-46BD-8198-A9BBD570E98D}	Kal	Shores	9473 Atlanta Ave	Sagle
{10DE82AF-D832-41AF-BE74-BC2A184888F8}	Kal	Creed	549 Inverness Dr	Gail
{D23C76EA-1237-4F27-8019-393001C1F371}	Kal	Eddington	2649 King George Dr	Regan
{5A392325-072D-45AB-AA36-28F2CCB800B7}	Kal	Sockler	925 Bedford Dr	Garrison
{0353AD96-E313-444D-AB4F-09D7DA75DD7A}	Kal	Trant	8859 Verde St	Ridge Manor
{86851458-187F-4FCB-8AC3-1913B8804046}	Kal	Sumpter	2161 Oakleigh Blvd	Merricourt

## Chapter 7: Creating Indexes 189

```
begin
  s := '';
  if InputQuery('Enter pattern for FirstName', 'Pattern', s) then
  begin
    Index := FDMemTable1.Indexes.Add;
    Index.Name := 'FirstName=' + s + 'RTIdx';
    Index.Fields := 'FirstName';
    Index.Filter := '[FirstName] = "' + s + '"';
    Index.FilterOptions := [ekPartial];
```

```
Index.Active := True;  
cbxIndexes.Items.Add( Index.Name );  
cbxIndexes.ItemIndex := cbxIndexes.Items.Count - 1;  
end  
else  
exit;  
end;  
Start;  
FDMemTable1.IndexName := cbxIndexes.Text;  
Complete;  
end;  
end;
```

**Figure 7-12: A runtime filter-based index has sorted and filtered the dataset**

190 Delphi in Depth: FireDAC

## **Two Runtime Index Examples: Sorting a DBGrid**

### **On-The-Fly**

A frequently requested feature in a database application is the ability to sort the data displayed in a DBGrid by clicking on the column title. The FDIndexes

project demonstrates how you can add this feature to any DBGrid that displays data from a FireDAC dataset.

This code is roughly based on some code that I published in my previous book: *Delphi in Depth: ClientDataSets, 2nd Edition, 2015*, and which was beautifully refactored by Bruce McGee, who reviewed a number of draft chapters of this book. However, FireDAC makes this process so much easier. In the

ClientDataSet version, I had to resort to RTTI (runtime type information) and had to create persistent indexes on the fly. With FireDAC's support for FDIndexes and support for both ascending and descending indexes using temporary indexes, this code is a fraction of the length of the code I wrote for my ClientDataSet book.

Actually, I am going to present two versions of this code, one that uses persistent indexes and a second that uses temporary indexes. While the temporary index version is shorter, what it produces is different, and I will discuss that difference at the conclusion of this section.

Let's begin with the code that permits sorting using FDIndex instances. This function, named SortFDDDataSetWithIndex, takes two parameters and returns a Boolean value indicating success or failure to sort the data. The first parameter is a FireDAC dataset, which means that you can pass an FDMemTable, an

FDQuery, an FDTable, or an FDStoredProc component, so long as it's an instance of that type that contains a result set. The second parameter is the name of the field on which you want to sort. If you pass the same FireDAC dataset and field name in two consecutive calls to this method, the first call will produce an ascending sort, and the second will produce a descending sort.

The code for this method is shown here:

```
function SortFDDDataSetWithIndex(DataSet: TFDDDataSet;
```

```
  const FieldName: String): Boolean;
```

```
  var
```

```
    Field: TField;
```

```
    newIndex: TFDIndex;
```

```
  type
```

```
    iDirection = (idAscending, idDescending);
```

Chapter 7: Creating Indexes 191

```
  function GetIndexName(const FieldName: string;
```

```
    Direction: iDirection): string;
```

```
  begin
```

```
    if Direction = idAscending then
```

```
      Result := FieldName + '_AIdx'
```

```
    else
```

```
      Result := FieldName + '_DIdx';
```

```
    end;
```

```
function CreateIndex(const FieldName: string;
Direction: iDirection): TFDIndex;
begin
Result := DataSet.Indexes.Add;
Result.Fields := FieldName;
Result.Name := GetIndexName(FieldName, Direction);
if Direction = idDescending then
Result.DescFields := FieldName;
Result.Active := True;
end;
function FindOrCreateIndex(const FieldName: string;
Direction: iDirection): TFDIndex;
begin
Result := DataSet.Indexes.FindIndex(
GetIndexName(FieldName, Direction));
if not Assigned(Result) then
Result := CreateIndex(FieldName, Direction);
Result.Active := True;
end;
begin
Result := False;
Field := DataSet.Fields.FindField(FieldName);
if not Assigned(Field) then
exit;
// if field type is not sortable, exit.
if (Field is TObjectField) or (Field is TBlobField) or (Field is TVariantField)
or (Field is TBinaryField) or not (Field.FieldKind in [ fkData, fkInternalCalc
]) then exit;
if SameText(DataSet.IndexName,
GetIndexName(FieldName, idAscending)) then
```

## 192 Delphi in Depth: FireDAC

```
NewIndex := FindOrCreateIndex.FieldName, idDescending)
else
  NewIndex := FindOrCreateIndex.FieldName, idAscending);
DataSet.IndexName := NewIndex.Name;
Result := True;
end;
```

As you can see, this code starts by verifying that the field name that was passed is a field in the FireDAC dataset defined in the first parameter, and also that it is a field type that is sortable. Next, the code checks to see if the dataset is already sorted using the ascending index name pattern, which is *FieldName*\_AIdx. If a match is found, the descending index name pattern ( *FieldName*\_DIdx) is either selected or created and selected. Otherwise, the ascending pattern index name is selected or created and selected.

The second sorting mechanism does not rely on FDIndexes, but instead makes use of the IndexFieldNames property of FireDAC datasets. Like the SortFDDDataSetWithIndex method, SortFDDDataSetWithFieldName takes two parameters, a FireDAC dataset and a field name. It also returns a Boolean result indicating success of failure to sort. This method is shown here:

```
function SortFDDDataSetWithFieldName(DataSet: TFDDDataSet;
const FieldName: String): Boolean;
var
  Field: TField;
begin
  Result := False;
  Field := DataSet.Fields.FindField(FieldName);
  if not Assigned(Field) then
    exit;
  // if field type is not sortable, exit.

  if (Field is TObjectField) or (Field is TBlobField) or (Field is TVariantField)
  or (Field is TBinaryField) or not (Field.FieldKind in [ fkData, fkInternalCalc
  ]) then exit;
```

```

if SameText(DataSet.IndexFieldNames, FieldName) or
SameText(DataSet.IndexFieldNames, FieldName + ':A') then
  DataSet.IndexFieldNames := FieldName + ':D'
else
  DataSet.IndexFieldNames := FieldName + ':A';
Chapter 7: Creating Indexes 193
Result := True;
end;

```

The first few lines of this method match that of the previous method. Both the field name and suitability for sorting is verified. The remainder of the code is similar in philosophy, but much simpler in execution. If the current value of IndexFieldNames matches either the field name or the field name with the A (ascending) modifier, then the IndexFieldNames property is assigned the name of the field with the D (descending) modifier. Otherwise, IndexFieldNames is assigned the field name with the D modifier.

You call one of these methods from the OnTitleClick method of a DBGrid (or some other method if you like, but then you'll have to determine yourself which values to pass in the two parameters). OnTitleClick executes when the user has clicked the header of a DBGrid column and the DBGrid's Options include the dgTitleClick flag. Here is the OnClick event handler found in the FDIndexes project, in which I have commented out the call to

SortFDDDataSetWithFieldName:

```

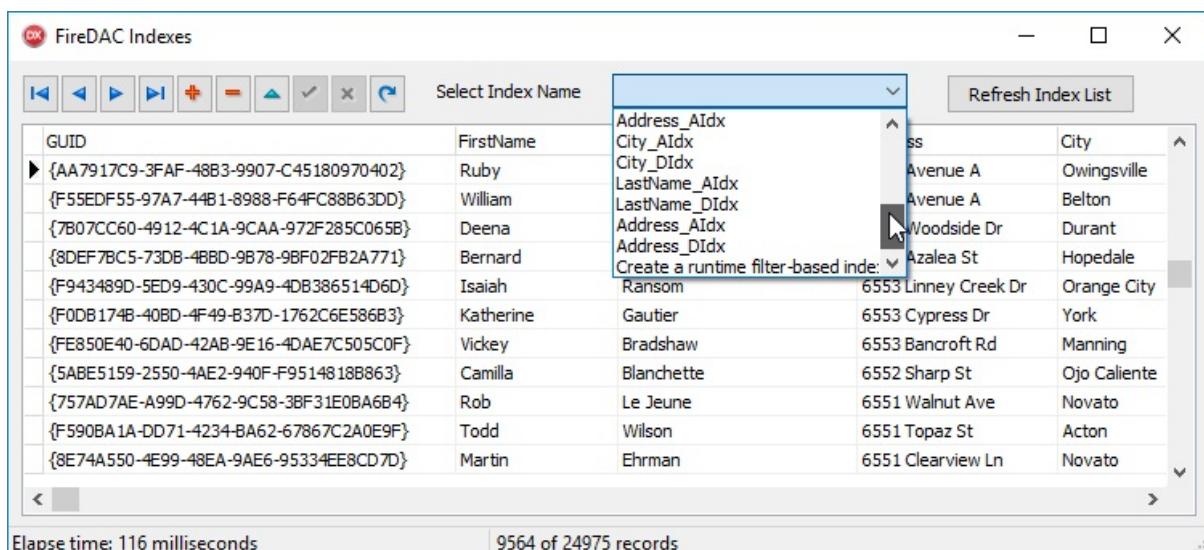
procedure TForm1.DBGrid1TitleClick(Column: TColumn);
begin
  Start;
  SortFDDDataSetWithFieldName( TFDDDataSet(Column.Field.DataSet),
    Column.FieldName );
  // SortFDDDataSetWithIndex( TFDDDataSet(Column.Field.DataSet),
  // 
  Column.FieldName );
  Complete;
end;

```

Referring back to the source code for these two sorting methods, when you

inspect these methods, you will conclude that the first method is more complicated, which is true. Why, then, would you consider using the first method? The answer is that although it is more complicated, it produces a different result, other than simply sorting the dataset. The first method also creates an FDIndex instance.

This is a subtle distinction, and you will probably rarely really care, since both methods produce similar results from the user's perspective. But the difference is real. Consider Figure 7-13. In this figure, I am using the first method, SortFDDDataSetWithIndex, and I have clicked on a number of different column headings, after which, I refreshed the index list and then have dropped down the



## 194 Delphi in Depth: FireDAC

index-selecting combobox and scrolled to the bottom. In the combobox, you can see the FDIndex names that have been created by SortFDDDataSetWithIndex.

**Figure 7-13: SortFDDDataSetWithIndex creates actual FDIndex definitions, as seen in the combobox**

In Figure 7-14, I am calling SortFDDDataSetWithFieldNames. Here I have done exactly the same thing as in Figure 7-13, I have run the project and sorted a number of different fields. Here, however, there are no indexes created, as you can see in the combobox.

FireDAC Indexes

The screenshot shows a software interface titled "FireDAC Indexes". On the left, there is a list of index names with a search bar labeled "Select Index Name" and a dropdown menu containing options like "FirstGuidDTIdx", "CityDistinctDTIdx", etc. On the right, there is a table with columns "Address", "City", and a dropdown menu with the option "Create a runtime filter-based index". Below the table, there is a message "Elapse time: 132 milliseconds" and a status bar indicating "8030 of 24975 records".

Address	City
Avenue A	Owingsville
Windsor Ct	Owenton
olivar St	Owenton
ansy Dr	Owenton
Nolan	5121 McAnelly Dr
Madden	8799 61st St
Hobson	2652 Summerwood Dr
Vail	2484 Oak St
Thornhill	8618 Parry St
Hayes	2523 Longwood St
Sylvester	6575 Riverbend St

## Chapter 7: Creating Indexes 195

**Figure 7-14: The SortFDDataSetWithFieldNames sorts, but does not create indexes**

In the next two chapters, I will take an in-depth look at searching and filtering.

## Chapter 8: Searching Data 197

# Chapter 8

## Searching Data

In the context of this chapter, *searching* means attempting to locate a record based on specific data that it contains. For example, attempting to find a record for a particular customer based on their customer id number is considered

searching. Likewise, finding an invoice based on the date of the invoice and the customer id number associated with that invoice is also considered a search operation. Importantly, searching often results in changing which record is the current record.

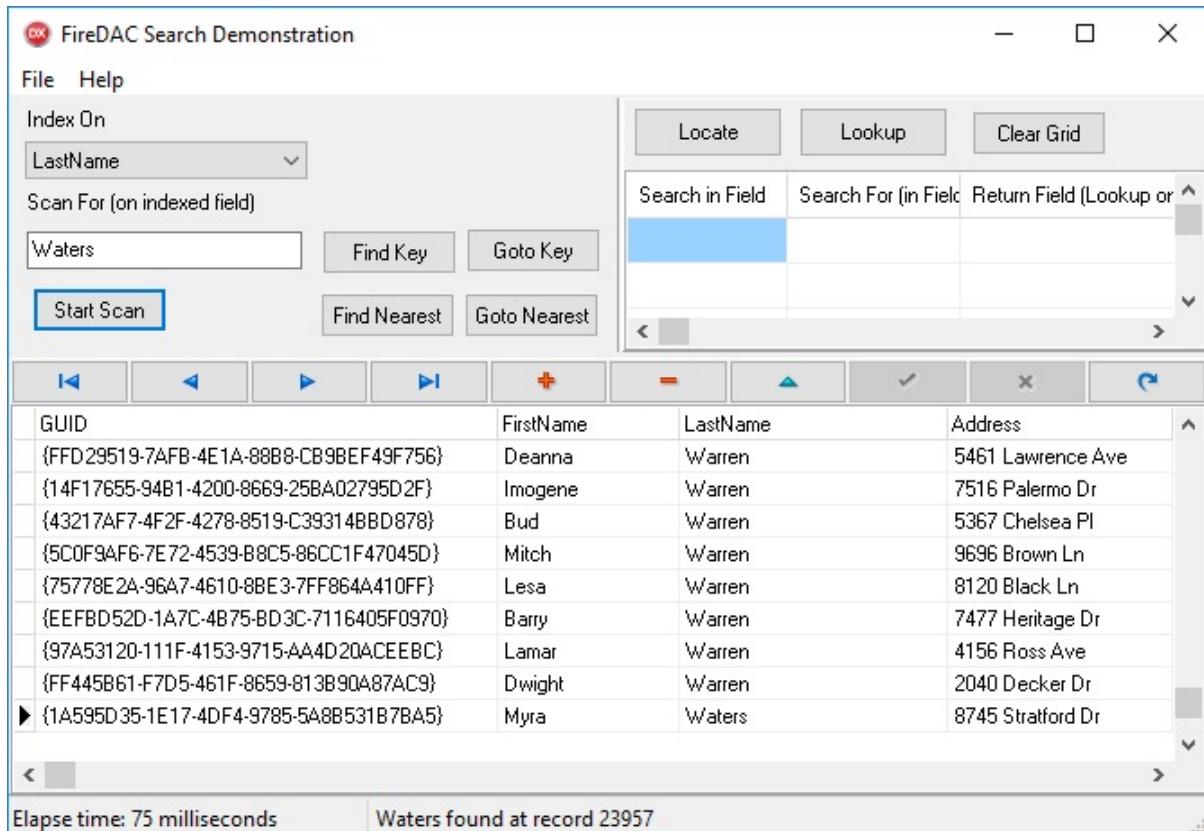
There is a somewhat similar operation that you can perform with FireDAC datasets called *filtering*. Filtering, which shares some characteristics with searching, involves selecting subsets of the records in a dataset based on the data. In other words, while searching may change the current record, filtering often results in a change in the number of records that are available within the dataset. As a result, I decided to cover filtering separately in *Chapter 9, Filtering Data*.

### Searching FireDAC DataSets

FireDAC datasets support a number of search operations, and each has its strengths and weaknesses. Some are relatively easy to employ, while others require some setup, such as ensuring that an appropriate index exists and is selected. Depending on your application's needs, one of the most important factors may be the speed of the search.

*Code: The FDSearch project is available from the code download. See Appendix A for details.*

In order to compare the speed of the various search options, we need to have some means of measuring the speed of searching. As demonstrated in the preceding chapter, I do this using the TStopWatch record and two custom methods: Start and Complete.



## 198 Delphi in Depth: FireDAC

Start initiates a stop watch, and Complete captures the elapsed time and displays the results in the status bar, as shown in Figure 8-1. In this figure, a search operation has been completed, and information about the search speed appears in the StatusBar.

**Figure 8-1: The speed of a search operation is displayed in the StatusBar of the FDSearch project**

Before discussing the individual search mechanisms, I also want to mention that we are once again using the BigMemTable.xml FDMemTable file, just as we

did in the preceding chapter. As you may recall, this saved FDMemTable file contains almost 25,000 records, making it a good candidate for comparing the speed of the various search mechanisms.

*Code: The file BigMemTable.xml is included in the code download.*

Because this project uses an FDMemTable, search operations on the data are very fast, since all of the data is in memory. If we used an FDQuery, the data

project.

In short, if you ensure that an FDQuery has loaded its entire result set into memory, search operations should be optimized. You use the FetchOptions.Mode property to control whether FireDAC datasets load all of their data at once, or on an as-needed basis.

### **Simple Record-by-Record Searches**

The simplest, and typically slowest, mechanism for searching is performed by scanning. As I described in *Chapter 6, Navigating and Editing Data*, you can scan a table by moving to either the first or last record in the current index order, and then navigating record-by-record. When you use scanning for a search

operation, you read each record programmatically as you search, comparing the record's data to the search criteria. When you find a record that contains the data you are looking for, you stop scanning.

An example of a scan-based search can be found on the OnClick event handler associated with the button labeled Start Scan in the FDSearch project. In fact, Figure 8-1 depicts the results of a search operation where a record-by-record scan located the first record in which the name Waters appears in the LastName field.

```
procedure TForm1.ScanBtnClick(Sender: TObject);  
var  
  Found: Boolean;  
begin  
  Found := False;  
  FDMemTable1.DisableControls;  
  Start;  
  try  
    FDMemTable1.First;  
    while not FDMemTable1.Eof do  
      begin  
        if FDMemTable1.Fields[IndexOnComboBox.ItemIndex].Value =  
          ScanForEdit.Text then  
          begin
```

```

Found := True;
Break;
end;
FDMemTable1.Next;
end;
Complete;
200 Delphi in Depth: FireDAC
finally
FDMemTable1.EnableControls;
end;
if Found then StatusBar1.panels[1].Text :=
ScanForEdit.Text + ' found at record '
+ IntToStr(FDMemTable1.RecNo)
else
StatusBar1.panels[1].Text :=
ScanForEdit.Text + ' not found';
end;

```

As you can see from the StatusBar shown in Figure 8-1, the last name Waters was found in record number 23,957, and this search took approximately 75 milliseconds. That's pretty remarkable, if you think about it. Specifically, the code in the preceding event handler evaluated almost 24,000 records in under a tenth of a second before finding a match. And of course, if there were many more records in the dataset, and the record you were searching for was much later in the dataset, the search operation would take longer.

While the scan appears to be very fast, it is nearly always the slowest search you can perform. Many of the remaining search mechanisms discussed in this chapter make use of indexes to perform the search, and this speeds things up significantly.

*Note: The scan executed in the preceding code was performed with updates disabled. If we had not invoked the DisableUpdates method on the FDMemTable before scanning, the operation would have taken a significantly longer time.*

## Searching with Indexes

Indexes, which I described in detail in *Chapter 7, Creating Indexes*, provide you with a number of features, and speeding up the search process is one of the more significant of these features. As you recall, indexes contain information about one or more fields of the dataset. And since these indexes are ordered by the fields on which they are based, they permit algorithms internal to the dataset to locate a record in which a search value appears very quickly.

There are two search mechanisms that rely entirely on the presence of indexes.

These are FindKey and GotoKey, as well as their close cousins, FindNearest and GotoNearest. FindKey and GotoKey perform their searches based on the fields of the currently selected index. Importantly, it does not matter whether this

### Chapter 8: Searching Data 201

index is a temporary index created using IndexFieldNames, or a permanent index defined using an FDIndex.

## FINDING DATA

One of the oldest mechanisms for searching a dataset in Delphi was introduced in the first release of Delphi (we often refer to this version as Delphi 1, but in reality, it was simply called Delphi). This method, FindKey, permits you to search one or more fields of the current index for a particular value. FindKey, and its relative, FindNearest, both make use of the current index to perform the search. As a result, the search is always index-based, and is always very fast.

Both FindKey and FindNearest take a single array of const parameter. (An array of const is a set of one or more comma separated values appearing within square braces.) You include in this array the values for which you want to search on the fields of the index, with the first element in the array being searched for in the first field of the index, the second field in the array (if provided) being searched for in the second field of the index, and so forth. For example, if you have an index based on CustomerNumber, LastName, and FirstName, in that order, you can use FindKey or FindNearest to search on just CustomerNumber,

CustomerNumber and LastName, or CustomerNumber and LastName and FirstName.

This pattern cannot be violated. For example, with a

CustomerNumber;LastName;FirstName index in place, you cannot use FindKey or FindNearest to search for just LastName, FirstName, or LastName and FirstName. And the search fields must be in the same order as the index, meaning that you cannot search for LastName and CustomerNumber, in that order. Finally, the number of fields being searched cannot exceed the number of fields in the index.

In the FDSearch project, the only indexes available are temporary indexes associated with a single field in the DataSet. The current temporary index is based on the fields listed in the IndexOnComboBox, which you can use to change which index the FDMemTable is using. Changing the current index also has the side effect of changing the order in which the records appear in the project's main form. The following code segment is found on the OnChange

event handler for this combobox:

```
procedure TForm1.IndexOnComboBoxChange(Sender: TObject);  
begin  
  FDMemTable1.IndexFieldNames :=  
    202 Delphi in Depth: FireDAC  
  IndexOnComboBox.Text;  
end;
```

The following single line populates this combobox. This line appears in the AfterOpen event handler of FDMemTable1 (which must be active in order for this code to perform its magic):

```
FDMemTable1.Fields.GetFieldNames( IndexOnComboBox.Items );
```

The use of FindKey is demonstrated by the code that appears in the OnClick event handler for the button labeled FindKey, shown in the following code segment:

```
procedure TForm1.FindKeyBtnClick(Sender: TObject);  
begin  
  Start;
```

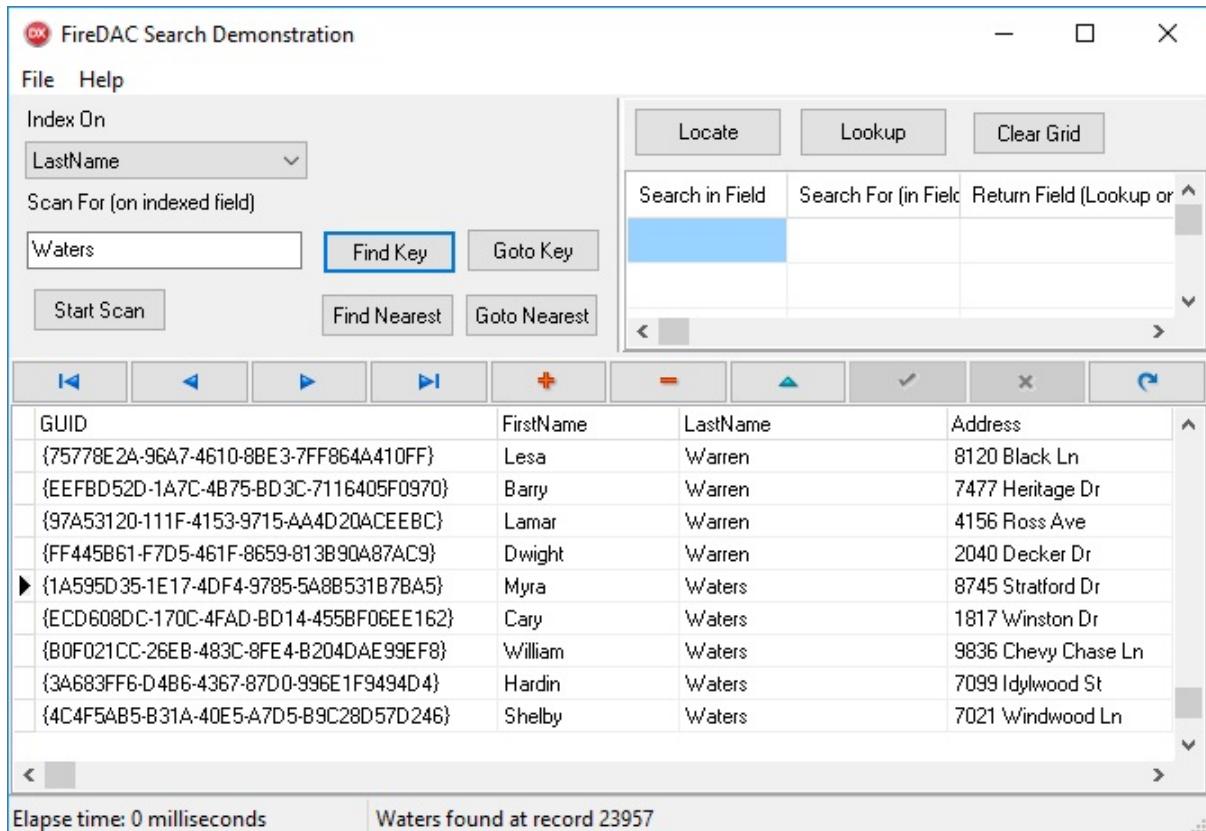
```

if FDMemTable1.FindKey([ScanForEdit.Text]) then
begin
  Complete;
  StatusBar1.panels[1].Text := ScanForEdit.Text +
    ' found at record ' +
    IntToStr(FDMemTable1.RecNo);
end
else
begin
  Complete;
  StatusBar1.panels[1].Text :=
    ScanForEdit.Text + ' not found';
end;
end;

```

Figure 8-2 shows how the main form of the FDSearch project might look following the use of FindKey to locate the first customer whose last name is Waters. As you can see in this figure, the search took less than a millisecond, even though the located record was close to the end of the 25,000 record table.

FindKey and FindNearest are identical in syntax. There is, however, a very big difference in what they do. FindKey is a Boolean function method that returns True if a matching record is located. In that case, the cursor is repositioned in the dataset to the found record, which, if there is more than one match, is the first match that is located based on the current index order. If FindKey fails, it returns False and the cursor remains on the current record.



## Chapter 8: Searching Data 203

**Figure 8-2: FindKey is used to search for the first customer named Waters.**

**This search, which is index-based, is much faster than a scan**

Unlike FindKey, which is a function, FindNearest is a procedural method.

Technically speaking, FindNearest always succeeds, moving the cursor to the record that most closely matches the search criteria. For example, in Figure 8-3, a search for the city named San Mateo has located a record. This record,

however, is not San Mateo. Instead, it is the closest match, San Pablo, which is alphabetically closer to San Mateo than the preceding record, San Marino.

Here is the code associated with the button labeled Find Nearest:

```
procedure TForm1.FindNearestBtnClick(Sender: TObject);
```

```
begin
```

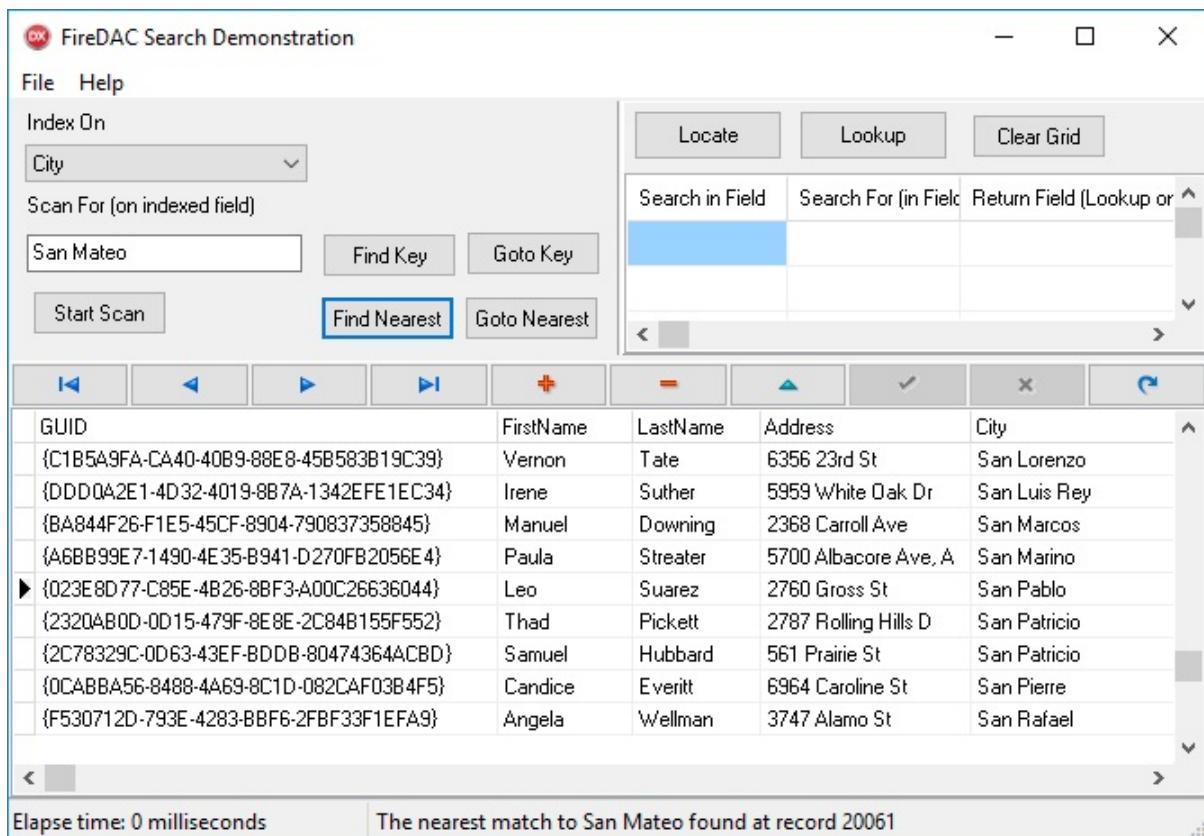
```
Start;
```

```
FDMemTable1.FindNearest([ScanForEdit.Text]);
```

```
Complete;
```

```
StatusBar1.panels[1].Text := 'The nearest match to ' +
```

ScanForEdit.Text + ' found at record ' +



## 204 Delphi in Depth: FireDAC

```
IntToStr(FDMemTable1.RecNo);  
end;
```

Although FindNearest always succeeds, and always moves to a new current record, the located *nearest* record may actually be the record on which the search started. Specifically, it is possible that the new current record was the old current record, but that would be completely circumstantial.

### **Figure 8-3: FindNearest always succeeds, even if it does not find an exact match**

## GOING TO DATA

GotoKey and GotoNearest provide the same searching features as FindKey and FindNearest, respectively. The only difference between these two sets of methods is how you define your search criteria. As you have already learned, FindKey and FindNearest are passed a constant array as a parameter, and the search criteria are contained in this array.

Both GotoKey and GotoNearest take no parameters. Instead, their search criteria are defined using the *search key buffer*. The search key buffer contains

one field

## Chapter 8: Searching Data 205

for each field in the current index. For example, if the current index is based on the field LastName, the search key buffer contains one field: LastName. On the other hand, if the current index contains three fields, the search key buffer also contains three fields.

Just as you do not have to define data for each field in the current index when using FindKey and FindNearest, you do not have to define data for each field in the search key buffer. However, those fields that you do define must be

associated with the left-most fields in the index definition. For example, if your index is based on LastName and FirstName, you can use the search key buffer to define only the LastName in the search key buffer, or both LastName and FirstName. Using this same index, you cannot define only the FirstName in the search key buffer.

Once the search key buffer has been populated with the values that you want to search for, you call GotoKey or GotoNearest. At this point, these methods perform the same search, with the same results, as FindKey and FindNearest, respectively.

Fields in the search key buffer can only be modified when the dataset is in a special state called the dsSetKey state. You call the dataset's SetKey method to clear the search key buffer and enter the dsSetKey state. If you have previously assigned one or more values to the search key buffer, you can enter the

dsSetKey state without clearing the search key buffer's contents by calling the dataset's EditKey method.

Once the dataset is in the dsSetKey state, you assign data to Fields in the search key buffer as if you were assigning data to the dataset's Fields. For example, assuming that the current index is based on the LastName and FirstName fields, the following lines of code assign the value Selman to the LastName field of the search key buffer, and the value Minnie to the FirstName field of the search key buffer:

```
FDQuery1.SetKey;
```

```
FDQuery1.FieldByName('LastName').Value := 'Selman';
```

```
FDQuery1.FieldByName('FirstName').Value := 'Minnie';
```

As should be apparent, using GotoKey or GotoNearest requires more lines of code than FindKey and FindNearest. For example, once again assuming that

the current index is based on the LastName and FirstName fields, consider the following statement:

206 Delphi in Depth: FireDAC

```
FDQuery1.FindKey(['Selman', 'Minnie']);
```

Achieving the same result using GotoKey requires four lines of code since you must first enter the dsSetKey state and edit the search key buffer. The following lines of code, which uses GotoKey, perform precisely the same search as the preceding line of code:

```
FDQuery1.SetKey;  
FDQuery1.FieldName('FirstName').Value := 'Minnie';  
FDQuery1.FieldName('LastName').Value := 'Selman';  
FDQuery1.GotoKey;
```

*Note: In the preceding code segment, I intentionally assigned a value to the FirstName field in the set key buffer before setting a value to the LastName field. Once in the dsSetKey state, it does not matter in which order you assign values to the fields, so long as you respect the other requirements, such as providing values to all of the left-most fields in the index order, without gaps.*

The following event handlers are associated with the buttons labeled Goto Key and Goto Nearest in the FDSearch project:

```
procedure TForm1.GotoKeyBtnClick(Sender: TObject);  
begin  
Start;  
FDMemTable1.SetKey;  
FDMemTable1.Fields[IndexOnComboBox.ItemIndex].AsString :=  
Trim(ScanForEdit.Text);  
if FDMemTable1.GotoKey then  
begin  
Complete;  
StatusBar1.panels[1].Text := ScanForEdit.Text +  
' found at record ' +  
IntToStr(FDMemTable1.RecNo);  
end
```

```
else
begin
  Complete;
  StatusBar1.panels[1].Text :=  

    ScanForEdit.Text + ' not found';
end;
Chapter 8: Searching Data 207
end;
procedure TForm1.GotoNearestBtnClick(Sender: TObject);
begin
  Start;
  FDMemTable1.SetKey;
  FDMemTable1.Fields[IndexOnComboBox.ItemIndex].AsString :=  

    ScanForEdit.Text;
  FDMemTable1.GotoNearest;
  Complete;
  StatusBar1.panels[1].Text := 'The nearest match to ' +
    ScanForEdit.Text + ' found at record ' +
    IntToStr(FDMemTable1.RecNo);
end;
```

Since GotoKey and GotoNearest perform essentially the same tasks as FindKey and FindNearest, though in a more verbose syntax, you might wonder why

anyone would use these methods when FindKey and FindNearest are available.

There is an answer, and it has to do with EditKey.

EditKey is a method that places the dataset in the dsSetKey state, but without clearing the search key buffer. As a result, EditKey permits you to change a single value or a subset of values in the search key buffer without affecting those values you do not want to change. As a result, there are times when

GotoKey provides you with a more convenient way to define and change your

search criteria. You may never need GotoKey or GotoNearest, but if you do, you'll be glad that these options exist.

## Searching with Variants

FireDAC datasets, like most TDataSets, provide two additional searching mechanisms, and these involve the use of variants. Unlike FindKey, FindNearest, and their Goto counterparts, these variant-using search mechanisms do not require an index.

For those search mechanisms that require an index, it means that in order to use them, you must first set the index. While this might not sound like a big deal, setting an index on a FireDAC dataset has the side effect of changing the sort order of the records. While you can sidestep the potential change of record order in the FireDAC dataset if you employ a cloned cursor, that too requires

additional code. I discuss using cloned cursors in detail in *Chapter 13, More FDMemTables: Cloned Cursors and Nested DataSets*.

208 Delphi in Depth: FireDAC

## LOCATING DATA

Locate, like FindKey and GotoKey, makes the located record the current record if a match is found. In addition, Locate is a function method and returns a Boolean True if the search produces a match. Lookup is somewhat different in that it returns requested data from a located record, but never moves the current record pointer. Lookup is described separately later in this chapter.

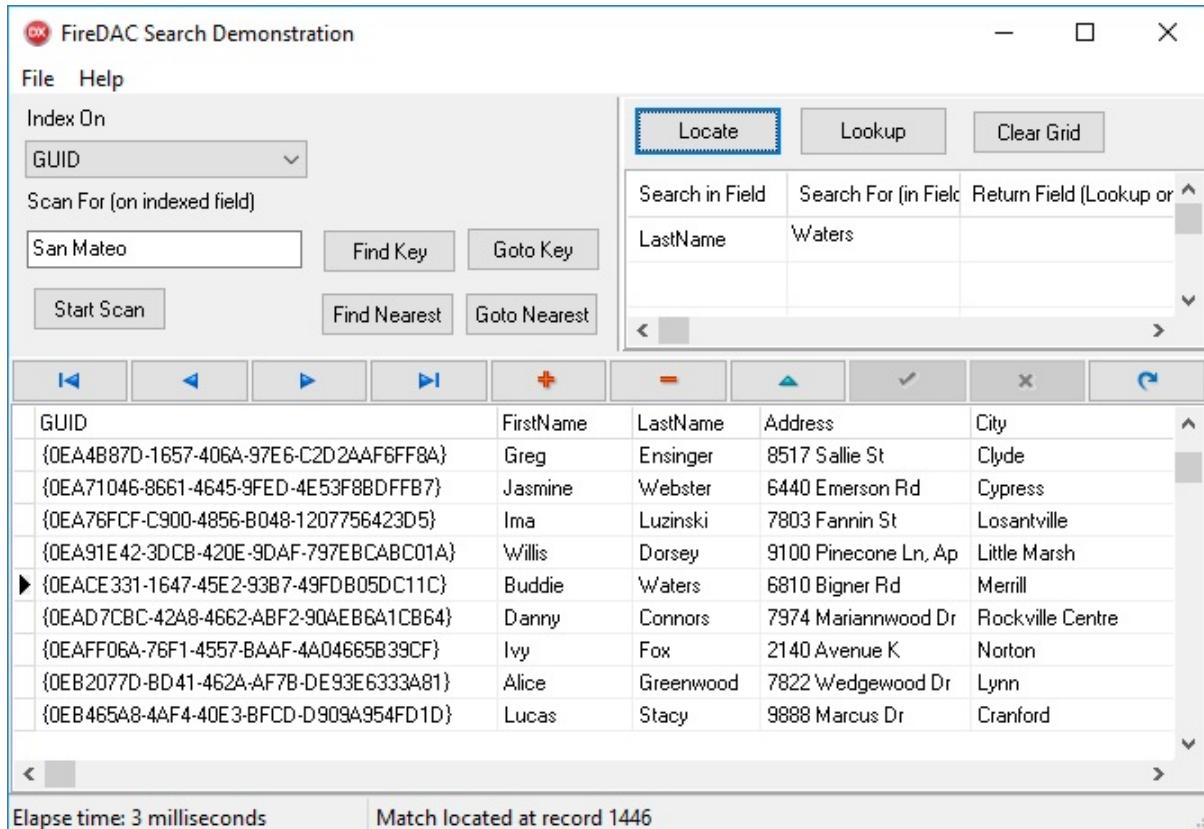
What makes Locate and Lookup so special is that they do not require you to create or switch indexes, but still provide much faster performance than scanning. In a number of tests that I have conducted, Locate is always faster than scanning, but generally slower than FindKey. Figure 8-4 displays a representative search, in this case, searching for a customer named Waters.

At first glance, Locate looks pretty good, as the preceding search results are close to that of the FindKey search shown in Figure 8-2, and significantly faster than scanning. However, upon closer inspection, a wrinkle appears. If you

compare Figure 8-4 with Figure 8-1 and Figure 8-2, you will notice that they did not locate the same record. Yes, they located a record with the last name, Waters, but it is not the same record.

Another aspect that is not obvious, unless you spend some time testing various uses of Locate, is that the speed of the search is dependent on the current order of records in the FDMemTable. For example, in Figure 8-4 the current index is based on the GUID field. If a different index was used, and there is more than one entry where the last name is Waters, a different record may have been

located.



## Chapter 8: Searching Data 209

**Figure 8-4: A customer with the last name Waters is found using Locate**

However, if you were to select the LastName index, where Waters is much later in the order, the Duration measure would likely increase. Should you care? That depends. Locate is fast — FindKey is faster, but Locate and Lookup do not

require an index switch. On the other hand, FindKey is fastest, though it requires an index switch or some cloned cursor mojo. But I think that you get what I'm trying to say. So let's learn how to use Locate.

Locate has the following syntax:

```
function Locate(const KeyFields: string;
  const KeyValues: Variant; Options: TLocateOptions): Boolean;
```

If you are locating a record based on a single field, the first argument is the name of that field and the second argument is the value you are searching for.

To search on more than one field, pass a semicolon-separated string of field names in the first argument, and a variant array containing the search values corresponding to the field list in the second argument.

## 210 Delphi in Depth: FireDAC

The third argument of Locate is a TLocateOptions set. This set can contain zero or more of the following flags: loCaseInsensitive and loPartialKey. Include loCaseInsensitive to ignore case in your search and loPartialKey to match any value that begins with the values you pass in the second argument.

If the search is successful, Locate makes the located record the current record and returns a value of True. If the search is not successful, Locate returns False and the cursor does not move.

Imagine that you want to find a customer with the last name Waters. This can be accomplished with the following statement:

```
FDQuery1.Locate('LastName', 'Waters', []);
```

The following is an example of a partial match, searching for a record where the LastName field begins with the letter W or w.

```
FDQuery1.Locate('LastName', 'w', [loCaseInsensitive, loPartialKey]);
```

Searching for two or more fields is more complicated in that you must pass the search values using a variant array. The following lines of code demonstrate how you can search for a record where the FirstName field contains Minnie and the LastName field contains Selman:

**var**

```
SearchList: Variant;
```

**begin**

```
SearchList := VarArrayCreate([0, 1], VarVariant);
```

```
SearchList[0] := 'Minnie';
```

```
SearchList[1] := 'Selman';
```

```
FDMemTable1.Locate('FirstName;LastName',
```

```
SearchList, [loCaseInsensitive]);
```

Instead of using VarArrayCreate, you can use VarArrayOf. VarArrayOf takes an array of const from which to create the variant array. This means that you must know at design time how many elements you will need in your variant

array. By comparison to VarArrayCreate, the dimensions of the variant array created

using VarArrayOf can include variables, which permit you to determine the  
Chapter 8: Searching Data 211

array size at runtime. The following code performs the same search as the preceding code, but makes use of an array created using VarArrayOf:

**var**

SearchList: Variant;

**begin**

SearchList := VarArrayOf(['Minnie','Selman']);

FDMemTable1.Locate('FirstName;LastName',SearchList,

[loCaseInsensitive]);

If you refer back to the FDSearch project main form shown in the earlier figures in this section, you will notice a StringGrid in the upper-right corner. Data entered into the first two columns of this grid are used to create the KeyFields and KeyValues arguments of Locate, respectively. The following methods,

found in the FDSearch project, generate these parameters:

**function** TForm1.GetKeyFields(var FieldStr: String): Integer;

**const**

FieldsColumn = 0;

**var**

i : Integer;

Count: Integer;

**begin**

Count := 0;

**for** i := 1 to 20 **do**

**begin**

**if** StringGrid1.Cells[FieldsColumn,i] <> " **then**

**begin**

**if** FieldStr = " **then** FieldStr :=

```

StringGrid1.Cells[FieldsColumn,i]
else
  FieldStr := FieldStr + ';' +
  StringGrid1.Cells[FieldsColumn,i];
  inc(Count);
end
else
  Break;
end;
Result := Count;
end;
function TForm1.GetKeyValues(Size: Integer): Variant;
const
  SearchColumn = 1;
  212 Delphi in Depth: FireDAC
var
  i: Integer;
begin
  Result := VarArrayCreate([0,Pred(Size)], VarVariant);
  for i := 0 to Pred(Size) do
    Result[i] := StringGrid1.Cells[SearchColumn, Succ(i)];
  end;

```

The following code is associated with the OnClick event handler of the button labeled Locate in the FDSearch project. As you can see in this code, the Locate method is invoked based on the values returned by calling GetKeyFields and GetKeyValues:

```

procedure TForm1.LocateBtnClick(Sender: TObject);
var
  FieldList: String;
  Count: Integer;

```

```

SearchArray: Variant;
begin
  FieldList := "";
  Count := GetKeyFields(FieldList);
  SearchArray := GetKeyValues(Count);
  Start;
  if FDMemTable1.Locate(FieldList, SearchArray, []) then
    begin
      Done;
      StatusBar1.Panels[3].Text :=
        'Match located at record ' +
        IntToStr(FDMemTable1.RecNo);
    end
  else
    begin
      Done;
      StatusBar1.Panels[3].Text := 'No match located';
    end;
  end;

```

## Chapter 8: Searching Data 213

*Note: Instead of passing a constant array in the call to VarArrayOf, you might be able to get away with passing an array of TVarRec. Using an array of TVarRec in place of an array of const is demonstrated in Chapter 9, Filtering Data , in the discussion of SetRange.*

## USING LOOKUP

Lookup is similar in many respects to Locate, with one very important difference. Instead of moving the current record pointer to the located record, Lookup returns a variant containing data from the located record without moving the current record pointer. The following is the syntax of Lookup:

**function** Lookup(**const** KeyFields: string;

**const** KeyValues: Variant; **const** ResultFields: string): Variant; The KeyFields and KeyValues parameters of Lookup are identical in purpose to those in the Locate method. The ResultFields parameter is a semicolon-separated string of field names whose values you want returned.

If Lookup fails to find the record being search for, it returns a null variant.

Otherwise, it returns a variant containing the field values requested in the ResultFields parameter.

The event handler associated with the Lookup button in the FDSearch project makes use of the GetKeyFields and GetKeyValues custom methods for defining the KeyFields and KeyValues parameters of the call to Lookup, based again on the first two columns of the StringGrid in the FDSearch project. In addition, this event handler makes use of the GetResultFields custom method to construct the ResultFields parameter from the third column of the grid. The following is the code associated with the GetResultFields method:

```
function TForm1.GetResultFields: String;  
const  
  ReturnColumn = 2;  
var  
  i: Integer;  
begin  
  for i := 1 to Succ(StringGrid1.RowCount) do  
    if StringGrid1.Cells[ReturnColumn, i] <> "" then  
      if Result = "" then  
        214 Delphi in Depth: FireDAC  
        Result := StringGrid1.Cells[ReturnColumn, i]  
      else  
        Result := Result + ';' +  
        StringGrid1.Cells[ReturnColumn, i]  
    else  
      Break;  
  end;
```

The following is the code associated with the OnClick event handler of the

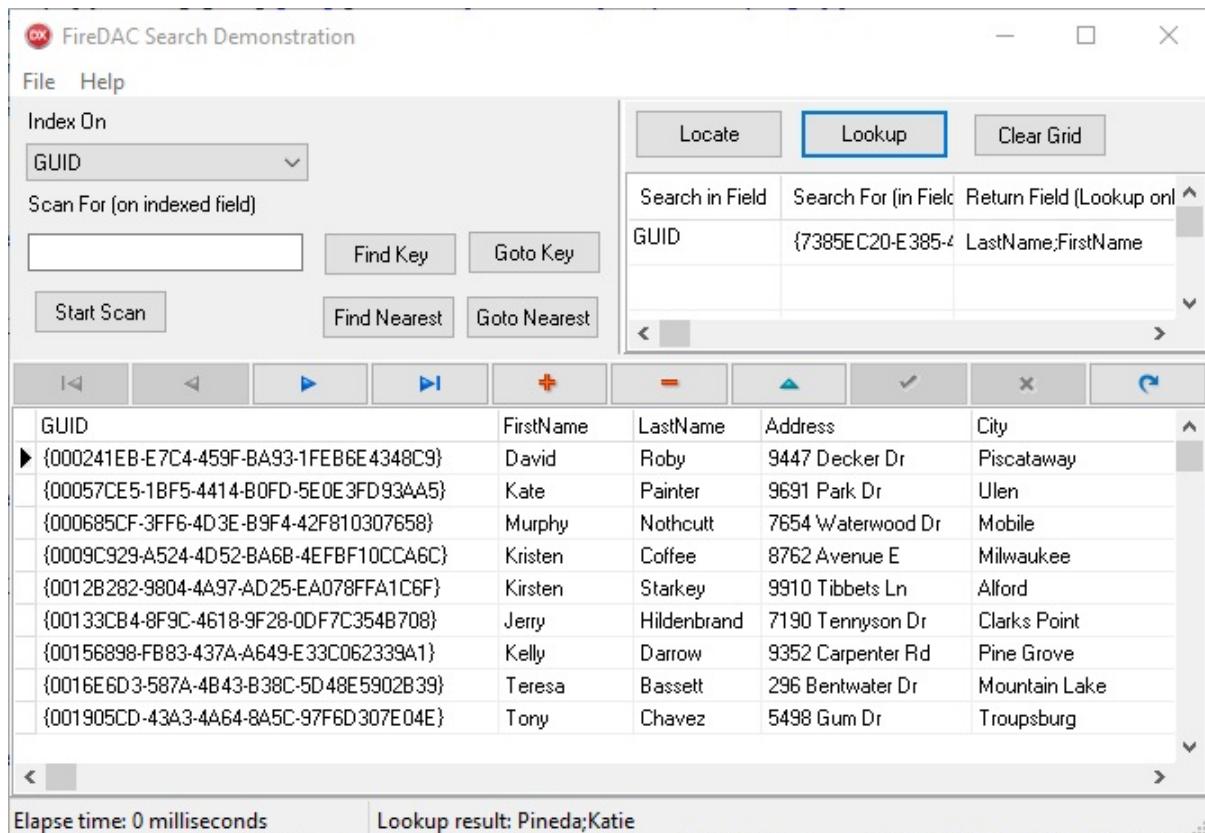
button labeled Lookup:

```
procedure TForm1.LookupBtnClick(Sender: TObject);  
var  
  ResultFields: Variant;  
  KeyFields: String;  
  KeyValues: Variant;  
  ReturnFields: String;  
  Count, i: Integer;  
  DisplayString: String;  
begin  
  Count := GetKeyFields(KeyFields);  
  DisplayString := ”;  
  KeyValues := GetKeyValues(Count);  
  ReturnFields := GetResultFields;  
  Start;  
  ResultFields := FDMemTable1.Lookup(KeyFields,  
    KeyValues, ReturnFields);  
  Complete;  
  if VarIsNull(ResultFields) then  
    DisplayString := ‘Lookup record not found’  
  else  
    if VarIsArray(ResultFields) then  
      for i := 0 to VarArrayHighBound(ResultFields,1) do  
        if i = 0 then  
          DisplayString := ‘Lookup result: ‘ +  
          VarToStr(ResultFields[i])  
        else  
          DisplayString := DisplayString +  
          ‘;’ + VarToStr(ResultFields[i])
```

```

else
  DisplayString := VarToStr(ResultFields);
  StatusBar1.Panels[3].Text := DisplayString
end;

```



## Chapter 8: Searching Data 215

Figure 8-5 shows the main form of the FDSearch project following a call to Locate. Notice that the current record is still the first record in the FDMemTable (as indicated by the DBNavigator buttons), even though the data returned from the call to Locate was found much later in the current index order.

*Note: Even though Lookup does not cause the current record to change, if the current record is in edit mode, a call to Lookup will cause that record to post, or raise an exception if posting fails.*

## Figure 8-5: Lookup is a relatively high-speed way to get data from a record without changing the current record of a dataset

In the next chapter, you will learn how to filter FireDAC datasets.

## Chapter 9: Filtering Data 217

# Chapter 9

## Filtering Data

When you filter a dataset, you restrict access to a subset of records contained in the dataset. For example, imagine that you have an FDQuery that includes one record for each of your company's customers, worldwide. Without filtering, all customer records are accessible in the result set. That is, it is possible to navigate, view, and edit any customer in the dataset.

Through filtering, you can make the FDQuery appear to include only those customers who live in the United States or in London, England, or who live on a street named 6th Avenue. This example, of course, assumes that there is a field in the dataset that contains country names, or fields containing City and Country names, or a field holding street names. In other words, a filter limits which records in a dataset are accessible based on data that is stored in the dataset.

While a filter is similar to a search, it is also different in a number of significant ways. For example, when you apply a filter, it is possible that the current record in the dataset will change. This will happen if the record that was current before the filter was applied no longer exists in the filtered dataset. When performing a search, the current record may change as well, specifically if the record you are searching for is not the current record. However, the record that was the current record prior to the search operation is still accessible in the dataset.

Another difference is that a search operation never changes the number of records in the FireDAC dataset, as reflected by the dataset's RecordCount property. By comparison, if at least one record in the dataset does not match the filter criteria, RecordCount will be lower following the application of the filter, and will change again when the filter is dropped. (Recall that RecordCount is also affected by the FetchOptions.RecordCountMode, as well as the

FetchOptions.LiveWindowParanoic properties.)

### Filters

A FireDAC datasets supports two fundamentally different mechanisms for creating filters. The first of these involves a *range*, which is an index-based

## 218 Delphi in Depth: FireDAC

filtering mechanism. The second, called a *filter*, is more flexible than a range, but is slower to apply and cancel. Both of these approaches to filtering are covered in the following sections.

But before addressing filtering directly, there are a couple of additional points that I need to make. The first is that filtering is a client-side operation.

Specifically, the filters discussed here are applied to the data that is loaded into the dataset. For example, you may load 10,000 records into an FDMemTable

(every customer record, for instance), and then apply a filter that limits access to only those customers located in Philadelphia.

Once applied, the filter may make the dataset appear to contain only 300 records (assuming that 300 of your customers are located in Philadelphia). Although the filtered dataset provides access only to these 300 records (and the RecordCount property returns 300), all 10,000 records remain in the dataset. In other words, a filter does not reduce the overhead of your dataset — it simply restricts your access to a subset of the dataset's records, those that match the filter criteria.

The second point is that instead of using a filter on the data loaded into your FireDAC dataset, you may be better off limiting how many records are available in your result set. Consider the preceding example where a query might return 10,000 customer records. Instead of loading all 10,000 records, it might be better to load only those customer records associated with customers who live in Philadelphia. Specifically, you might want to use a WHERE clause predicate that limits the result set records to those associated with Philadelphia-based customers. For example, consider a query similar to the following:

```
SELECT * FROM Customer WHERE City = 'Philadelphia'
```

While the preceding query seems rather limiting in that it only allows the selection of customers from Philadelphia, a better approach would be to define the query using a parameter, similar to the SQL shown here:

```
SELECT * FROM Customer WHERE City = :City
```

With this query in play, and with a single FDParam named City defined in the FDQuery's Params property, you could use code similar to the following to allow the end user to select into the dataset customer records from any city:

```

var
  CityName: String;
begin
  if InputQuery('Select Customers from which City',
    'City', CityName) then
    begin
      FDQuery1.SQL.Text := 'SELECT * FROM Customer ' +
      ' WHERE City = :c;';
      FDQuery1.Params[0].AsString := CityName;
      FDQuery1.Open;
    end;
  end;

```

*Important!: The use of a parameter to include a value entered by the end user to be employed in the predicate part of the SQL statement is very important in order to avoid a vulnerability known as an SQL injection hack. For more information about SQL injection, refer to the section Prevention of SQL Injection in Chapter 5, More Data Access .*

Nonetheless, from the perspective of this discussion, these techniques are not technically dataset filtering since they do not limit access within the dataset to a subset of its loaded records.

So, when do you use filtering as opposed to loading only selected records into a FireDAC dataset? The answer boils down to four issues: bandwidth, the source of data, the amount of data, and client-side features.

When loading data from a DataSnap server, bandwidth is a concern. In distributed applications like these (we are talking DataSnap here), it is usually best to load only selected records when bandwidth is low. In this situation, loading records that are not going to be displayed would consume bandwidth unnecessarily, affecting the performance of your application as well as that of others that share the bandwidth. On the other hand, if bandwidth is plentiful and the entire result set is relatively small, it is often easier to load all data and filter on those records that you want displayed.

The second consideration is data location. If you are loading data from a previously saved FireDAC dataset (using LoadFromFile or

`LoadFromStream),`

you have no choice. Filtering is the only option for showing just a subset of records. Only when you are loading data through a query or stored procedure call do you have a choice between using a filter and selectively loading the data.

See *Chapter 17, Understanding Local SQL* for information on querying data loaded through a call to `LoadFromFile` or `LoadFromStream`.

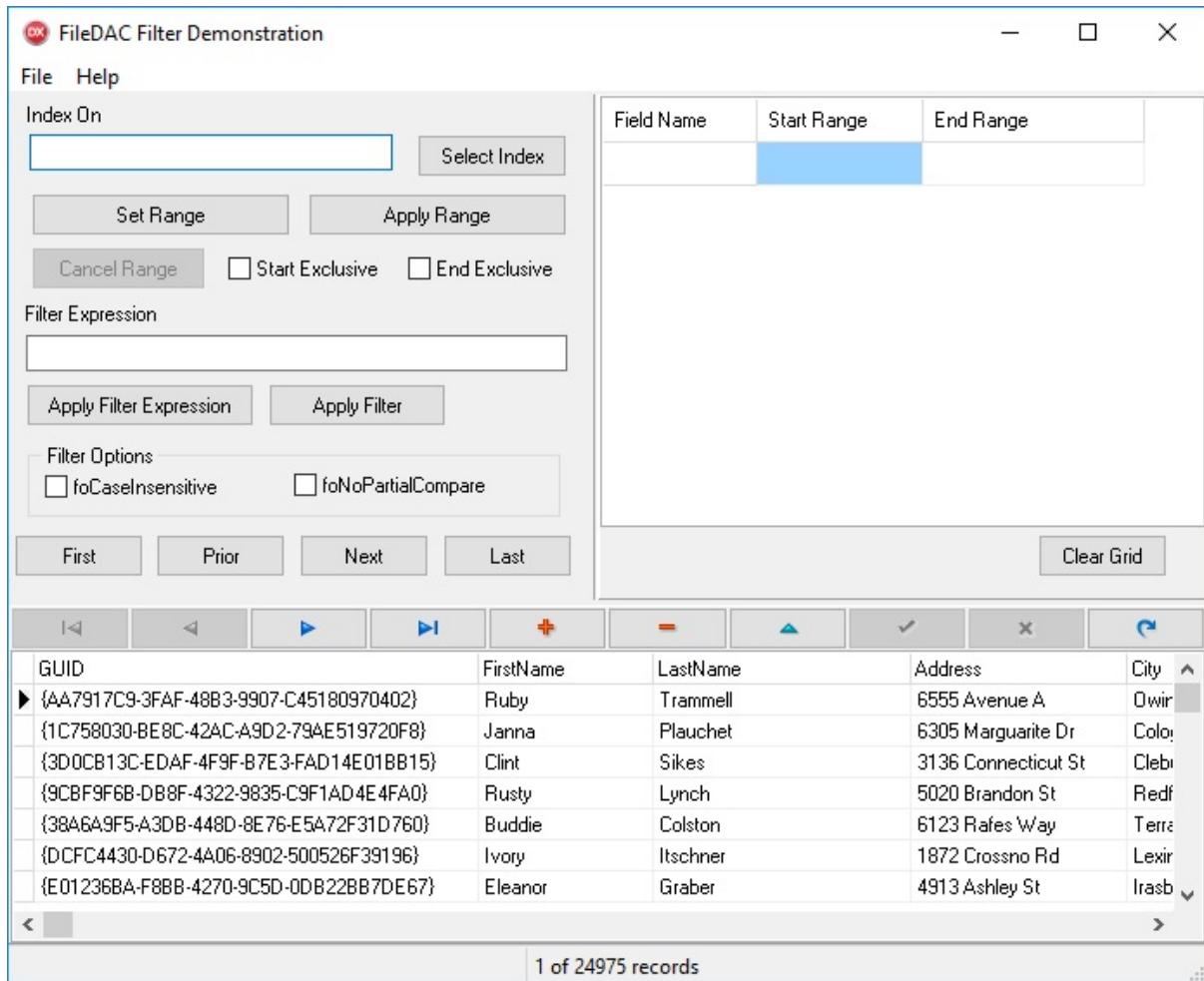
## 220 Delphi in Depth: FireDAC

The third consideration is the amount of data. If your SQL queries a database table that has a very large amount of data, it may not even be possible to load all of that data into the FDQuery. For example, if the query returns millions of records, or contains a number of large BLOB (Binary Large OBject) fields, it may not be possible to load all of that data into memory at the same time. In these cases, you must use some technique, such as using a well-considered

WHERE clause, to load only that data you need or can handle.

The final consideration is related to client-side features, the most common of which is speed. Once data is loaded into memory in the client, most filters are applied very quickly, even when a large amount of data needs to be filtered. As a result, filtering permits you to rapidly alter which subset of records are displayed. A simple click of a button or a menu selection can almost instantly switch your dataset from displaying customers from Philadelphia to displaying customers from Dallas, without a network roundtrip.

The use of filters is demonstrated in the FDFilter project. The main form for this project is shown in Figure 9-1.



## Chapter 9: Filtering Data 221

**Figure 9-1: The main form of the FDFilter project**

*Code: The code project FDFilter is available from the code download. See Appendix A for details.*

Several types of filters are demonstrated in the FDFilter project. This project also makes use of BigMemTable.xml, a saved FDMemTable file that contains almost 25,000 records. This file, which is assumed to be located in a directory named BigMemTable in a folder parallel to the one in which the FDFilter project is located, is loaded from the OnCreate event handler of the main form.

This event handler will also display an error message if BigMemTable.xml cannot be located.

### 222 Delphi in Depth: FireDAC

*Code: The file BigMemTable.xml is available from the code download.*

As mentioned earlier, there are two basic approaches to filtering: ranges and

filters. Let's start by looking at ranges.

## Using a Range

Ranges, although less flexible than filters, provide the fastest option for displaying a subset of records from a FireDAC dataset. In short, a range is an index-based mechanism for defining the low and high values of records to be displayed in the dataset. For example, if the current index is based on the customer's last name, a range can be used to display all customers whose last name is Jones. Or, a range can be used to display only customers whose last name begins with the letter J. Similarly, if a FireDAC dataset is indexed on an integer field called Credit Limit, a range can be used to display only those customers whose credit limit is greater than (USD) \$1,000, or between \$0 and \$1000.

## SETTING RANGES

Setting ranges with a FireDAC dataset bears a strong resemblance to the index-based search mechanisms covered in *Chapter 8, Searching Data*. First, both of these mechanisms require that you set an index, and that index can be a

temporary index or FDIndex-based. The range, like the search, is then based on one or more fields in the current index. A second similarity between ranges and index-based searches is that there are two ways to set a range. One technique involves a single method call, similar to FindKey, while the other employs the set key buffer, like GotoKey.

Let's start by looking at the easiest mechanism for setting a range — the SetRange method. SetRange defines a range using a single method invocation.

In FireDAC, SetRange has the following syntax:

```
procedure SetRange(const StartValues, EndValues: array of const;  
AStartExclusive: Boolean = False;  
AEndExclusive: Boolean = False );
```

As you can see from this syntax, at a minimum you pass two arrays of const when you call SetRange. The first array contains the low values of the range for the fields of the index, with the first element in the array being the low value of the range for the first field in the index, the second element being the low value of the range for the second field in the index, and so on. The second array

contains the high end values for the index fields, with the first element in the second array being the high end value of the range on the first field of the index, the second element being the high value of the range on the second field of the index, and so forth. These arrays can contain fewer elements than the number of fields in the current index, but cannot contain more.

The SetRange call includes two optional parameters. Pass a Boolean value in the third parameter to control whether the lower end values provided in the first parameter are included in the range or not. If you pass False, the low end of the range will be included in the range. Pass True to exclude the lower end values from the Range. The fourth parameter works the same, but with respect to the upper end parameters. The default values for these parameters is False, in which case the range is inclusive.

Consider again our example of an FDQuery that returns all customer records.

Given that there is a field in this result set named City, and you want to display only records for customers who live in Pleasantville, you can use the following statements:

```
FDQuery1.IndexFieldNames := 'City';  
FDQuery1.SetRange(['Pleasantville'], ['Pleasantville']);
```

The first statement creates a temporary index on the City field, while the second sets the range. Of course, if the dataset is already using an index where the first field of the index is the City field, you can omit the first line in the preceding code segment.

Now, consider this example:

```
FDQuery1.IndexFieldNames := 'City';  
FDQuery1.SetRange(['Pleasantville'], ['Pleasantville'], True, True);
```

In this case, we have chosen to exclude both the low end and the high end values. Since we were filtering only on a single value, the dataset would display no records, since records whose City field contains Pleasantville will be

excluded. What this example shows is that AStartExclusive and AEndExclusive are only meaningful when your range includes more than one value.

The preceding example sets the range on a single field, but it is often possible to set a range on two or more fields of the current index. For example, imagine that

you want to display only those customers whose last name is Waters and who live in Pleasantville, New York. The following statements show you how:

```
FDMemTable1.IndexFieldNames := 'LastName;City;State';
```

```
FDMemTable1.SetRange(['Waters', 'Pleasantville', 'NY'],  
['Waters', 'Pleasantville', 'NY']);
```

Each of these examples sets the range to a single value in all fields, and in both cases, setting the third and fourth parameters to True would have produced a record count of zero. It is sometimes possible to set a range that includes a range of values. For example, imagine that you want to find all customers who live in California and in a city whose name begins with A. You could achieve this with the following statements:

```
FDMemTable1.IndexFieldNames := 'State;City';
```

```
FDMemTable1.SetRange(['CA', 'Aa'],  
['CA', 'Az']);
```

In this case, AStartExclusive and AEndExclusive will not necessarily produce an empty dataset.

The use of SetRange is demonstrated in the OnClick event handler associated with the button labeled Set Range. However, before you can define a range, you must define an index.

You define an index by clicking the Select Index button. The event handler on this button was described in some detail in *Chapter 7, Creating Indexes*, so I won't go into a lot of detail here. But suffice it to say that this event handler ensures that the user can enter a valid set of fields for the current dataset (which it assigns to the FDMemTable's IndexFieldNames property). Once the index is set, it also initializes the StringGrid that appears in the upper-right side of the main form, displaying the name of each index field, and providing fields for you to enter the low and high values for one or more fields of the index.

That the first two parameters of SetRange are arrays of const make calls to SetRange somewhat inflexible. Specifically, you typically have to know exactly how many elements you need to include in the array of const in advance. In most applications of SetRange, this is not an issue as the number of elements is known.

## Chapter 9: Filtering Data 225

When I originally wrote an example of SetRange back in 1996, I wanted to accommodate from one to five elements in the range, which meant that I had

to include one call to SetRange for each combination of fields on which I wanted to filter. This code looked something like the following, where

GetMaxRangeItems is a custom method that determines how many elements are

in the range:

```
procedure TForm1.SetRangeBtnClick(Sender: TObject);
var
  MaxItems: Integer;
begin
  if FDMemTable1.IndexFieldNames = "" then
    begin
      ShowMessage('Set index field names before trying to set a range');
      Exit;
    end;
  MaxItems := GetMaxRangeItems;
  case MaxItems of
    0: ShowMessage('Enter a range');
    1: begin
      Start;
      FDMemTable1.SetRange(
        [StringGrid1.Cells[1,1]],
        [StringGrid1.Cells[2,1]]);
      Done;
    end;
    2: begin
      Start;
      FDMemTable1.SetRange(
        [StringGrid1.Cells[1,1]],
        StringGrid1.Cells[1,2]),
        [StringGrid1.Cells[2,1]]);
```

```
StringGrid1.Cells[2,2]]);
```

```
Done;
```

```
end;
```

```
3: begin
```

```
/// And so on
```

Fortunately, the Delphi language has improved significantly over the years, and now instead of passing an array of const, we have the option of using an array of TVarRec. As a result, it is easier to create code that can accommodate a variable number of range elements. This can be seen in the OnClick event handler for the button labeled Set Range. This event handler is shown here:

226 Delphi in Depth: FireDAC

```
procedure TForm1.SetRangeBtnClick(Sender: TObject);
```

```
var
```

```
MaxItems: Integer;
```

```
arLow: array of TVarRec;
```

```
arHigh: array of tvarrec;
```

```
i: Integer;
```

```
begin
```

```
if FDMemTable1.IndexFieldNames = ” then
```

```
begin
```

```
ShowMessage(‘Set index field names before ‘ +
```

```
‘trying to set a range’);
```

```
exit;
```

```
end;
```

```
MaxItems := GetMaxRangeItems;
```

```
SetLength( arLow, MaxItems);
```

```
SetLength( arHigh, MaxItems);
```

```
for i := 1 to MaxItems do
```

```
begin
```

```
arLow[i-1].vtype := vtUnicodeString;
```

```

arLow[i-1].VUnicodeString :=  

Pointer((StringGrid1.Cells[ 1, i]));  

arHigh[i-1].vtype := vtUnicodeString;  

arHigh[i-1].VUnicodeString :=  

Pointer((StringGrid1.Cells[ 2, i]));  

end;  

FDMemTable1.SetRange( arLow, arHigh,  

cbxStartExclusive.Checked,  

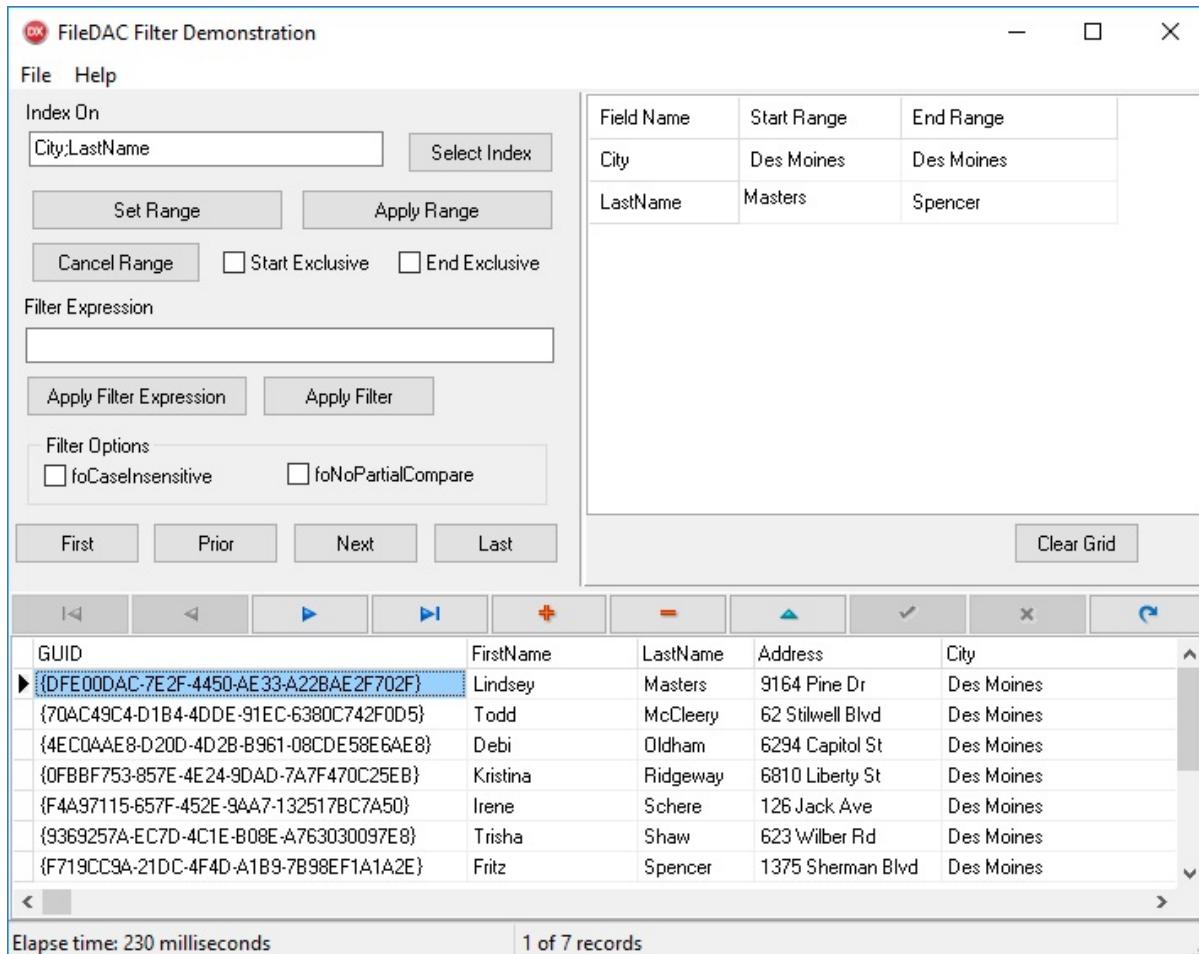
cbxEndExclusive.Checked );  

CancelRangeBtn.Enabled := True;  

end;

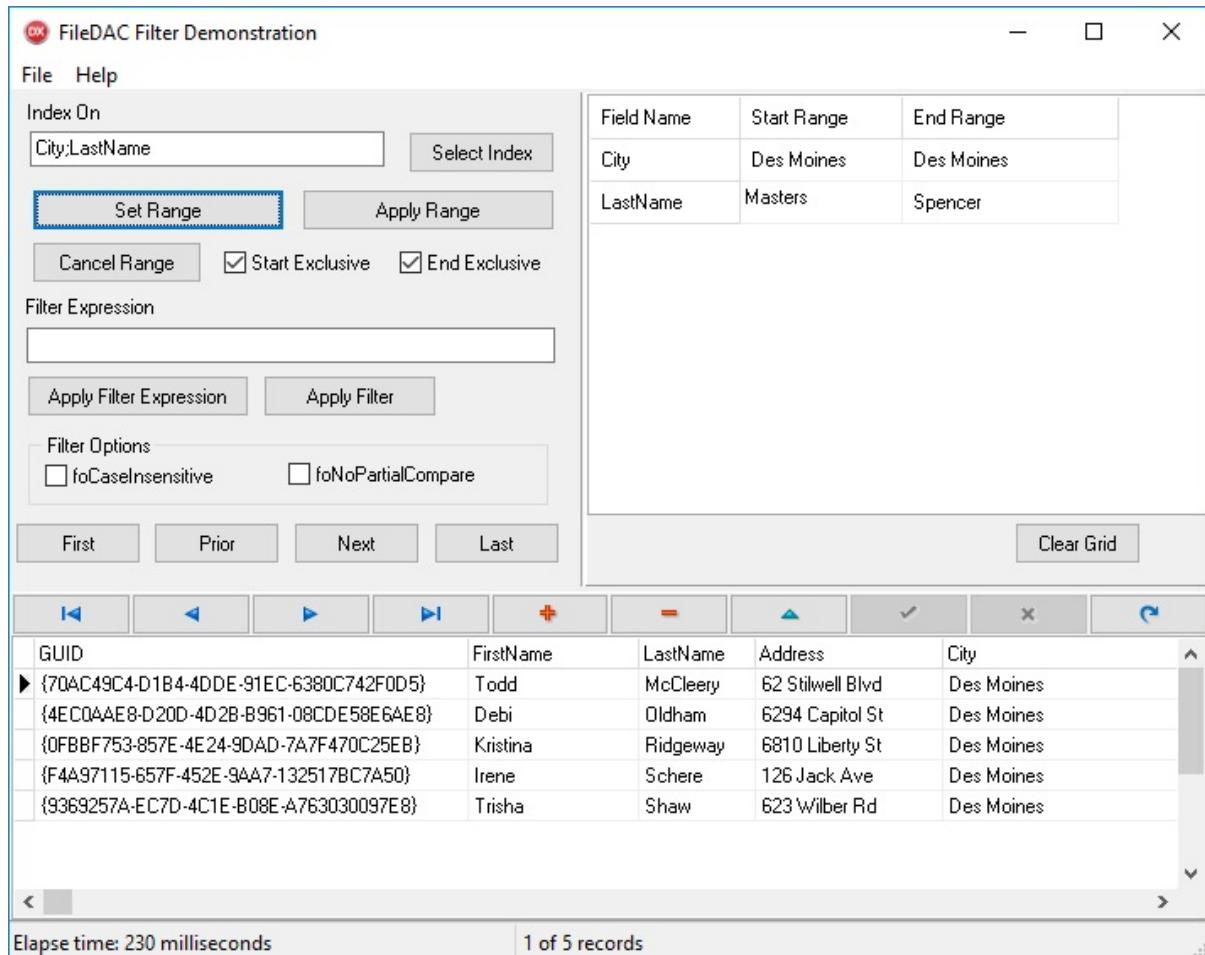
```

Figure 9-2 shows how the FDFilter project looks after a range has been set on two fields. This range displays all records associated with people from Des Moines whose last name ranges from Masters to Spencer.



## Figure 9-2: A range on two fields has been applied to the dataset

Now see what happens when you use the exclusive parameters. Figure 9-3 shows the same filter, except that this time both the Start Exclusive and End Exclusive checkboxes have been checked. The subsequent call to SetRange passed True in the third and fourth parameters, and as a result, the names Masters and Spencer are no longer in the range.



## 228 Delphi in Depth: FireDAC

### Figure 9-3: Start Exclusive and End Exclusive have resulted in the starting and ending values of the range being omitted from the range

#### USING APPLYRANGE

ApplyRange is the range equivalent to the GotoKey search. To use ApplyRange, you begin by calling SetRangeStart (or EditRangeStart). Doing so places the dataset in the dsSetKey state. While in this state, you assign values to one or more of the Fields involved in the current index to define the low values of the range. As is the case with SetRange, if you define a single low value, it will be used to define the low end of the range on the first field

of the current index. If you define the low values of the range for two fields, they must necessarily be the first two fields of the index.

After setting the low range values, you call SetRangeEnd (or EditRangeEnd).

You now assign values to one or more fields of the current index to define the high values for the range. Once both the low values and high values of the range have been set, you call ApplyRange to filter the dataset on the defined range.

## Chapter 9: Filtering Data 229

For example, the following statements use ApplyRange to display only customers who live in Philadelphia:

```
FDQuery1.IndexFieldNames := 'City';
FDQuery1.SetRangeStart;
FDQuery1.FieldName('City').Value := 'Philadelphia';
FDQuery1.SetRangeEnd;
FDQuery1.FieldName('City').Value := ' Philadelphia ';
FDQuery1.ApplyRange;
```

Just like SetRange, ApplyRange can be used to set a range on more than one field of the index, as shown in the following example:

```
FDQuery1.IndexFieldNames := 'LastName;City;State';
FDQuery1.SetRangeStart;
FDQuery1.FieldName('LastName').Value := 'Waters';
FDQuery1.FieldName('City').Value := 'Pleasantville';
FDQuery1.FieldName('State').Value := 'NY';
FDQuery1.SetRangeEnd;
FDQuery1.FieldName('LastName').Value := 'Waters';
FDQuery1.FieldName('City').Value := 'Pleasantville';
FDQuery1.FieldName('State').Value := 'NY';
FDQuery1.ApplyRange;
```

Both of the preceding examples make use of SetRangeStart and SetRangeEnd.

In some cases, you can use EditRangeStart and/or EditRangeEnd instead. In

short, if you have already set low and high values for a range, and want to modify some, but not all, values, you can use EditRangeStart and EditRangeEnd. Calling SetRangeStart clears any previous values in the set key buffer. By comparison, if you call EditRangeStart, the previously defined low values remain in the set key buffer. If you want to change some, but not all, of the low range values, call EditRangeStart and modify only those fields whose low values you want to change. Likewise, if you want to change some, but not all, of the high range values, do so by calling EditRangeEnd.

For example, the following code segment will display all records in which the customer's credit limit is between (USD) \$1,000 and (USD) \$5,000:

```
FDQuery1.IndexFieldNames := 'CreditLimit';
```

```
FDQuery1.SetRange([1000],[5000]);
```

230 Delphi in Depth: FireDAC

If you then want to change the range to between \$1,000 and \$10,000, you can do so using the following statements:

```
FDQuery1.EditRangeEnd;
```

```
FDQuery1.FieldName('CreditLimit').Value := 10000;
```

```
FDQuery1.ApplyRange;
```

At first glance, ApplyRange sounds like it is more verbose than SetRange, and this is true when you know in advance how many fields on which your range is based. Ironically, when you do not know how many fields you need to set a

range on in advance, and cannot use an array of TVarRec, ApplyRange can actually be more concise.

Consider the OnClick event handler associated with the button labeled Apply Range. This event handler, shown in the following code segment, performs the same task as my original (circa 1996) call to Set Range, a partial segment of which I showed you. As you can see, the next event handler is much shorter. It can also handle any number of fields in the index without requiring more code: **procedure TForm1.ApplyRangeBtnClick(Sender: TObject);**

**var**

MaxItems: Integer;

i: Integer;

**begin**

```

if FDMemTable1.IndexFieldNames = " then
begin
ShowMessage('Set index field names before trying to set a range');
Exit;
end;

MaxItems := GetMaxRangeItems;
if MaxItems = 0 then
begin
ShowMessage('Enter a range');
Exit;
end;

Start;
FDMemTable1.SetRangeStart;
for i := 1 to MaxItems do
FDMemTable1.FieldName(StringGrid1.Cells[0,i]).Value :=
Trim(StringGrid1.Cells[1,i]);
FDMemTable1.SetRangeEnd;
for i := 1 to MaxItems do
Chapter 9: Filtering Data 231
FDMemTable1.FieldName(StringGrid1.Cells[0,i]).Value :=
Trim(StringGrid1.Cells[2,i]);
Complete;
if MaxItems < 6 then
CancelRangeBtn.Enabled := True;
end;

```

## **CANCELING A RANGE**

Whether you have created a range using SetRange or ApplyRange, you cancel that range by calling the dataset's CancelRange method. Canceling a range is demonstrated by the OnClick event handler associated with the button labeled Cancel Range, as shown in the following code:

```
procedure TForm1.CancelRangeBtnClick(Sender: TObject);  
begin  
  FDMemTable1.CancelRange;  
  CancelRangeBtn.Enabled := False;  
end;
```

## A COMMENT ABOUT RANGES

Earlier I mentioned that it is *sometimes* possible to set a range on two or more fields where a range of values is included in the result (all customers with a credit limit above 1000) compared to setting a range where all records in the range have the same value (all customers from Pleasantville). The implication of this statement is that sometimes it is not possible to set a range where the records have a range of values, and that is intentional since it is a correct conclusion.

When setting a range on two or more fields, only the last field of the range can specify a range of values; all other fields must have the same value for both the low and high ends of the range. For example, the following range will display all records in which the credit limit is between \$1,000 and \$5,000 for customers living in Des Moines:

```
FDMemTable1.IndexFieldNames := 'City;CreditLimit';  
FDMemTable1.SetRange(['Des Moines', 1000], ['Des Moines', 5000]);
```

### 232 Delphi in Depth: FireDAC

By comparison, the following statement will display all records for customers whose credit limit is between \$1,000 and \$5,000, regardless of which city they live in:

```
FDMemTable1.IndexFieldNames := 'CreditLimit;City';  
FDMemTable1.SetRange([1000, 'Des Moines'], [5000, 'Des Moines']);
```

The difference between these two range examples is that in the first example, the low and high value in the first field of the range is a constant value, Des Moines. In the second, a range appears (1000-5000). Because a range of values, instead of the same value, appears in the first field of the range, the second field of the range is ignored.

The bottom line is this. If you want a range of values between a low value and a high value to appear, it must be defined only in the last element of the array of const parameters passed to SetRange, or defined in ApplyRange.

## Using Filters

Because ranges rely on indexes, they are applied very quickly. For example, using the BigMemTable.xml table with an index on the FirstName field, setting a range to show only records for customers where the first name is Billy was applied in less than a millisecond on my computer.

Filters, by comparison, do not use indexes. Instead, they operate by evaluating the records of the FireDAC dataset, displaying only those records that pass the filter. Since filters do not use indexes, they are not as fast. (Filtering on the first name Billy took about 40 milliseconds in my tests.). On the other hand, filters are much more flexible.

*Note: You might recall from Chapter 7, Creating Indexes , that you can create a Filter-based index. This discussion of a lack of performance using filters does not apply to filter-based indexes, which use both filters and an index to produce very fast filtered views.*

FireDAC datasets, like most other TDataSet descendants, have four properties that apply to filters. These are: Filter, Filtered, FilterOptions, and

OnFilterRecord (an event property). In its simplest case, a filter requires that you use two of these properties: Filter and Filtered. Filtered is a Boolean

Chapter 9: Filtering Data 233

property that you use to turn filtering on and off. If you want to filter records, set Filtered to True. Otherwise, set Filtered to False (the default value).

## Basic Filters

When Filtered is set to True, the dataset uses the value of the Filter property to identify which records will be displayed. You assign to this property a Boolean expression containing at least one comparison operator involving at least one field in the dataset. You can use any comparison operators, including =, >, <,

>=, <=, and <>. As long as the field name does not include any spaces, you include the field name directly in the filter expression without delimiters. For example, if your dataset includes a field named City, you can set the Filter property to the following expression to display only customers living in Dayton (when Filtered is True):

City = 'Dayton'

Note that the single quotes are required here, since Dayton is a string literal. If you want to assign a value to the Filter property at runtime, you must include the single quotes in the string that you assign to the property. The following is one example of how to do this:

```
FDMemTable1.Filter := 'City = ' + QuotedStr('Philadelphia');
```

The preceding code segment used the QuotedStr function, which is located in the System.SysUtils unit. The alternative is to use something like the following: FDMemTable1.Filter := ‘City = ”Philadelphia”’;

In the preceding examples, the field name of the field in the filter does not include spaces. If one or more fields that you want to use in a filter include spaces in their field names, or characters that would otherwise be interpreted to mean something else, such as the > (greater than) symbol, enclose those field names in square braces. (Square braces can also be used around field names that do not include spaces or special characters.) For example, if your dataset contains a field named ‘Last Name,’ you can use a statement similar to the following to create a filter:

## 234 Delphi in Depth: FireDAC

```
FDMemTable1.Filter := ‘[Last Name] = ‘ + QuotedStr(‘Williams’);
```

These examples have demonstrated only simple expressions. However, complex expressions can be used. Specifically, you can combine two or more comparisons using the AND, OR, and NOT logical operators. Furthermore, more than one field can be involved in the comparison. For example, you can use the following filter to limit records to those where the City field is San Francisco and the Last Name is Martinez:

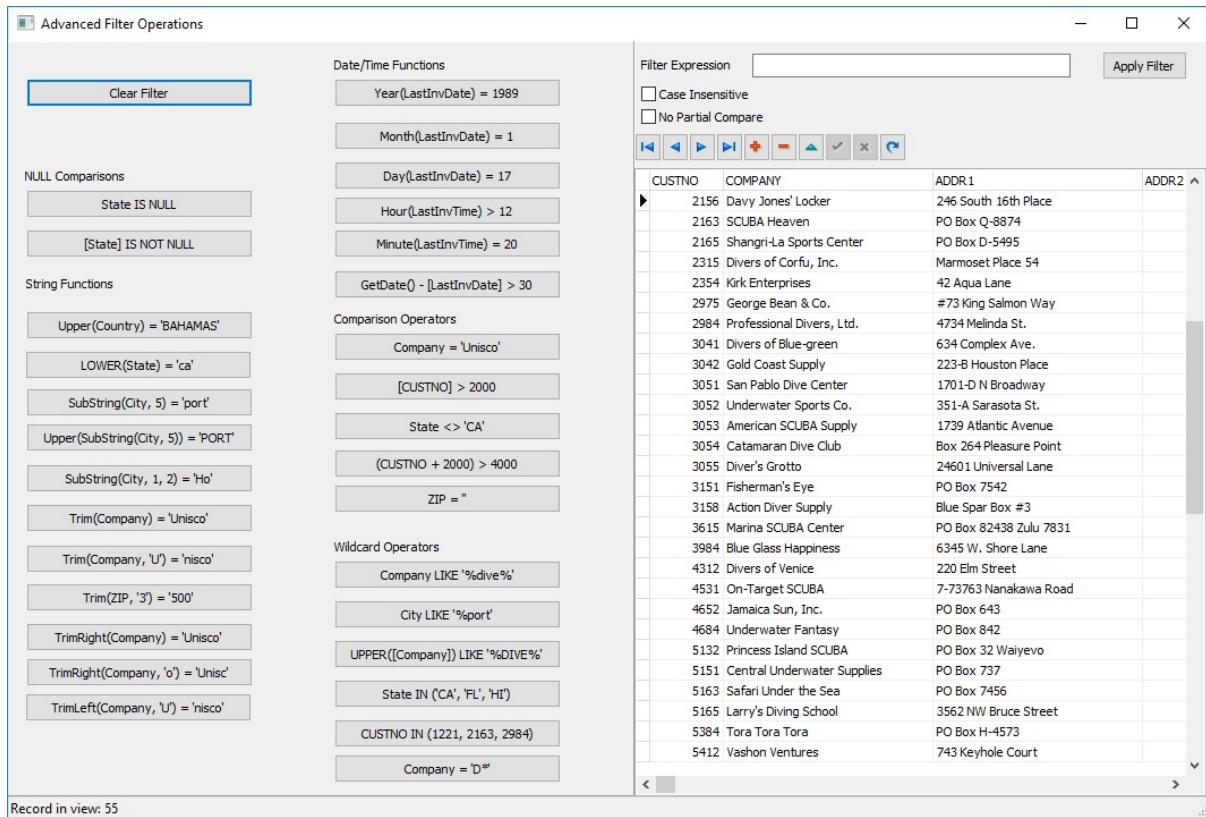
```
FDMemTable1.Filter := ‘[City] = ‘ + QuotedStr(‘San Francisco’) +  
‘and [Last Name] = ‘ + QuotedStr(‘Martinez’);
```

## Special Filter Expressions

In addition to field references, literal values, and operators, there are two categories of special functions that you can use in filters. Some of these, including some NULL comparisons, string functions, date/time functions, and miscellaneous functions, are available for use by all TDataSets, not just

FireDAC datasets. The use of these Delphi functions, as well as some of the other filter properties (such as FilterOptions) are demonstrated in the

FDAdvancedFilters project, the main form of which is shown in Figure 9-4.



Record in view: 55

## Chapter 9: Filtering Data 235

### Figure 9-4: The main form of the FDAdvancedFilters project

In addition, FireDAC introduces a large number of additional functions that you can use in filters, and these are the FireDAC scalar functions. In order to use FireDAC's scalar functions in a filter, you must add the

FireDAC.Stan.ExprFuncs unit to your uses clause. An in-depth discussion of these scalar functions really belongs in the chapter on the SQL command

preprocessor, so I am going to limit my discussion of using them in filters, saving a more detailed discussion for *Chapter 14, The SQL Command Preprocessor*. For more information about these functions, refer to the section *FireDAC Scalar Functions* in Chapter 14.

*Code: You can find the FDAdvancedFilters project in the code download.*

The FDAdvancedFilters project contains a series of buttons whose captions contain valid filter expressions. When clicked, the Caption property of the clicked button is assigned to the Filter property of an FDQuery (which has its Filtered property set to True). More is said about Filtered later in this chapter).

## 236 Delphi in Depth: FireDAC

In fact, there are very few OnClick event handlers used in this project. One of these event handlers is assigned to every button on which the Caption

property is set to a valid filter. This event handler, named AssignFilter in this project, casts the Sender parameter as a TButton instance, and then assigns the Caption property of this TButton reference to the Filter property of the FDQuery. This event handler is shown here:

```
procedure TForm1.AssignFilter(Sender: TObject);  
begin  
  if Sender is TButton then  
    FDMemTable1.Filter := TButton(Sender).Caption  
  else  
    ShowMessage('Invalid event handler assigned to ' +  
    Sender.Name);  
end;
```

The Button whose caption reads Clear Filter has an event handler that assigns an empty string to the Filter property, which, even though the Filtered property is set to True, has the effect of dropping the filter. This event handler is shown here:

```
procedure TForm1.ClearFilter(Sender: TObject);  
begin  
  FDMemTable1.Filter := '';  
end;
```

This project also includes an Edit, into which you can enter a filter expression, after which you can click the accompanying button labeled Apply Filter. The Apply Filter button assigns the Text property of the Edit to the FDQuery's Filter property and, since the Filtered property of the FDQuery is already set to True, results in filtering. (As with the buttons whose captions contain a valid filter expression, you clear this filter by clicking the button labeled Clear Filter.) The FDAAdvancedFilters project main form is pretty large, so in order to make the remaining figures in this chapter readable, I will use the Edit and its Apply Filter button to demonstrate the filters.

## NULL COMPARISONS

Null comparisons permit you to compare a dataset's fields to a null value. Unlike the other comparison filter operators, these comparisons use a special syntax that prevents the dataset from interpreting the keyword NULL as a field name.

The screenshot shows a Windows application window titled 'FDAdvancedFilters'. At the top, there is a 'Filter Expression' input field containing 'State IS NULL' and a 'Apply Filter' button. Below the input field are two checkboxes: 'Case Insensitive' and 'No Partial Compare'. Underneath these are several small icons for filtering operations. The main area is a grid displaying customer data from a dataset. The columns are labeled 'CUSTNO', 'COMPANY', 'ADDR1', 'ADDR2', 'CITY', and 'STATE'. The data rows are:

CUSTNO	COMPANY	ADDR1	ADDR2	CITY	STATE
1231	Unisco	PO Box Z-547		Freeport	
1351	Sight Diver	1 Neptune Lane		Kato Paphos	
1354	Cayman Divers World Unlimited	PO Box 541		Grand Cayman	
1513	Fantastique Aquatica	Z32 999 #12A-77 A.A.		Bogota	
1984	Adventure Undersea	PO Box 744		Belize City	
2163	SCUBA Heaven	PO Box Q-8874		Nassau	

## Chapter 9: Filtering Data 237

The use of NULL comparisons are demonstrated by the two buttons that appear under the label NULL comparisons.

For example, to filter the State field on records where the State field is a null value, use the following filter expression:

State IS NULL

To include only records where the State field contains data, use the following:  
State IS NOT NULL

Figure 9-5 shows the FDAdvancedFilters project after the State IS NULL filter has been applied.

### **Figure 9-5: The State IS NULL filter expression has been applied**

Delphi's documentation states that it is possible to use the keyword BLANK to test for empty fields. I have never been able to get this to work, and believe that the documentation is incorrect. For example, if you assign the following filter expression to a FireDAC's dataset Filter property, nothing happens (or maybe FireDAC tells you that you need to use the FireDAC.Stan.ExprFuncs unit):

```
FDMemTable1.Filter := 'Company = BLANK'; // Does not work
```

238 Delphi in Depth: FireDAC

## **STRING FUNCTIONS**

Delphi supports six string functions. These are Upper, Lower, SubString, Trim, TrimRight, and TrimLeft.

Upper and Lower are the simplest of these to use. Upper converts a single string expression to uppercase, and Lower converts a string expression to lowercase.

Here is an example of a string expression that converts the contents of the City field to uppercase, after which it compares that value to ‘KAPAA KAUAI’:

Upper(CITY) = ‘KAPAA KAUAI’

The SubString function is more complicated, in that it can accept either two or three arguments. When used with two arguments, the first argument is a string expression (typically a string field reference) and the second is the position of the first character from which to begin the comparison (1-based, not 0-based).

For example, the following expression compares the City field to the string **port**, beginning with the fifth character of the City field. Records where the City field contains the value **Freepo**rt match this filter:

SubString(City, 5) = ‘port’

When three arguments are passed to SubString, the first is a string expression (again, typically a string field), the second is the position within that string to begin the comparison, and the third is the number of characters to compare.

As a result, the following filter expression will display only those records whose City field contains the letter **r** in the second position. Figure 9-6 shows the FDAdvancedFilters project filtering on this filter expression:

SubString(CITY, 2, 1) = ‘r’

The screenshot shows a software interface for filtering data. At the top, there is a 'Filter Expression' input field containing 'SubString(CITY, 2, 1) = 'r''. Below it are two unchecked checkboxes: 'Case Insensitive' and 'No Partial Compare'. Underneath these are several small icons for navigating through filters. A main table displays customer information with columns: CUSTNO, COMPANY, ADDR1, ADDR2, CITY, and STATE. The data shows five rows of customers, all of whom have 'r' in their city names. The table has scroll bars on the right and bottom.

CUSTNO	COMPANY	ADDR1	ADDR2	CITY	STATE
1231	Unisco	PO Box Z-547		Freepo	
1354	Cayman Divers World Unlimited	PO Box 541		Grand Cayman	
2165	Shangri-La Sports Center	PO Box D-5495		Freepo	
3151	Fisherman's Eye	PO Box 7542		Grand Cayman	
5163	Safari Under the Sea	PO Box 7456		Grand Cayman	

Chapter 9: Filtering Data 239

### Figure 9-6: The SubString function is used to filter records

The Trim, TrimRight, and TrimLeft functions all take either one or two arguments. When a single string argument is passed, these functions will remove blank spaces from either the right side (TrimRight), the left side

(TrimLeft), or both left and right sides of the string argument (Trim). For example, the following filter expression will trim white space (blank characters) from both the right and left of the Company field, and compare the result to the string Unisco:

Trim(Company) = ‘Unisco’

When the optional second parameter is employed, you pass a single character in the second argument, and it is that character (instead of a blank character) that is trimmed. Consequently, the following filter expression limits the dataset to displaying only records where the Company name field contains the value nisco once all of the leading U characters have been trimmed off:

TrimRight(Company, ‘U’) = ‘nisco’

## DATE/TIME FUNCTIONS

Delphi’s date/time filter expression functions permit you to make comparisons based on partial date values, such as a particular year, month, hour, or second. In addition, they permit you to compare against the date part or time part of a timestamp expression, and even compare using the current date.

The screenshot shows the FDAdvancedFilters component's filter configuration window. At the top, there is a 'Filter Expression' input field containing 'Year(LastInvDate) = 1989'. Below it are two unchecked checkboxes: 'Case Insensitive' and 'No Partial Compare'. Underneath these are several small icons for filtering operations like AND, OR, NOT, and others. The main area is a grid displaying customer data. The columns are labeled CUSTNO, COMPANY, ADDR1, ADDR2, CITY, STATE, ZIP, COUNTRY, and LASTINVOICEDATE. The data rows are:

CUSTNO	COMPANY	ADDR1	ADDR2	CITY	STATE	ZIP	COUNTRY	LASTINVOICEDATE	PHK
2135	Frank's Divers Supply	1455 North 44th St.		Eugene	OR	90427	US	6/24/1989 10:29:46 AM	503
2165	Shangri-La Sports Center	PO Box D-5495		Freeport			Bahamas	7/3/1989 3:32:49 PM	013
5132	Princess Island SCUBA	PO Box 32 Waiyeho		Taveuni			Fiji	12/30/1989 9:20:17 AM	679
6312	Aquatic Drama	921 Everglades Way		Tampa	FL	30643	US	3/25/1989 4:42:52 AM	613

## 240 Delphi in Depth: FireDAC

In order to demonstrate the date/time filter expression functions in the FDAdvancedFilters project, I had to make some adjustments to my query. For example, a SELECT \* FROM Customer query returns a TSQLTimestampField

for the LastInvoiceDate column of the Customer table. Apparently, you cannot use date/time filter expression functions on this type of field. Therefore, I used FireDAC’s SQL preprocessing capabilities, specifically, its CONVERT

function, to create TDateField and TTimeField instances, upon which I then applied the date/time function. The following is the SQL statement used by the FDQuery that appears in the FDAdvancedFilters project:

```
SELECT c.*,
{fn CONVERT(c.LastInvoiceDate, DATE) } AS LastInvDate,
{fn CONVERT(c.LastInvoiceDate, Time) } AS LastInvTime
FROM Customer c
```

Using these two new fields, named LastInvDate and LastInvTime, it is possible to demonstrate the use of Delphi's date/time filter expression functions.

The following filter expression will cause the dataset to display only those records where the LastInvDate field contains dates from the year 1989:

Year(LastInvDate) = 1989

The effect of this filter is shown in Figure 9-7.

### **Figure 9-7: The Year function is used to filter on records from 1989**

Chapter 9: Filtering Data 241

Similarly, the following filter expression causes Delphi to display records whose LastInvTime value later in the day than 12:00 pm (noon):

Hour(LastInvTime) > 12

The Date() function returns the date portion of a date/time value, and Time() returns the time portion. Also, GetDate() returns the current date. As a result, the following filter will return all records whose InvoiceDueDate is more than 30

days old, and for which the InvoicePaid field is null (though no records matching this filter exist in the Customer table of Delphi's dbdemos.gdb InterBase database):

GetDate() - 30 > InvoiceDueDate AND InvoicePaidDate IS NULL

## **MISCELLANEOUS FUNCTIONS**

There are two miscellaneous Delphi functions supported by filter expressions, and these are LIKE and IN. The LIKE statement operates similar to how the

SQL-92 LIKE statement works, in that you compare a string expression with a pattern that can include the % and \_ wildcard characters. Within this pattern, the

% wildcard stands for zero or more characters, and the \_ character stands for exactly one character.

The following filter expression demonstrates a use of LIKE, and will search for any Company name that includes the characters Dive:

Company LIKE '%Dive%'

Likewise, the following filter expression will display any record where the City field ends with the characters port:

State LIKE '%port'

Of course, special filter expressions can be combined. Here is a filter that will display only records where the letters 'shop' (in any combination of upper or lowercase letters) appear, followed by exactly two more characters. (Note that

A screenshot of a Delphi IDE interface. At the top, there is a filter dialog box with the following content:

- Filter Expression: `Upper(Company) LIKE '%SHOP_.'`
- Case Insensitive
- No Partial Compare
- Toolbar with various filter operators (e.g., AND, OR, NOT, etc.).
- An "Apply Filter" button.

Below the filter dialog is a query results grid displaying the following data:

CUSTNO	COMPANY	ADDR1	ADDR2	CITY	STATE
1221	Kauai Dive Shoppe	4-976 Sugarloaf Hwy	Suite 103	Kapaa Kauai	HI

## 242 Delphi in Depth: FireDAC

we have had to insert a slight space between the two underscore characters in the following expression for readability. Had we not done this, the printed copy would appear to include a single underscore character.) The effect can be seen in Figure 9-8:

`Upper(Company) LIKE '%SHOP_ _'`

### Figure 9-8: Special filter expressions can be combined

Similar to LIKE, the IN keyword works similar to that found in SQL-92. When using IN, you follow an expression with the keyword IN followed by a comma-separated set of possible values enclosed in parentheses. The following filter expression displays only companies whose mailing address is in CA

(California), FL (Florida), or HI (Hawaii):

`State IN ('CA', 'FL', 'HI')`

## ESCAPING WILDCARD CHARACTERS

As you learned, you can use the % and \_ wildcard characters to filter using the LIKE operator. If the pattern that you are searching for includes a character that otherwise would be interpreted as a wildcard character, you must precede that character with the \ (backslash) escape character.

For example, if you want to filter on records where the Company field includes at least one underscore character (\_), you can use the following filter expression:

The screenshot shows a Delphi IDE interface. At the top, there is a filter dialog with the expression `LENGTH([Company]) = 12`. Below it, a dataset viewer displays a single record from a table with columns: CUSTNO, COMPANY, ADDR1, ADDR2, CITY, STATE, ZIP, COUNTRY, and PH. The record shown is: CUSTNO 2163, COMPANY SCUBA Heaven, ADDR1 PO Box Q-8874, CITY Nassau, COUNTRY Bahamas. The dataset viewer has navigation buttons (first, previous, next, last) and a scroll bar.

Chapter 9: Filtering Data 243

Company LIKE '%\\_%'

If you want to use a backslash in a LIKE expression, you escape it with another backslash. In other words, two consecutive backslashes filter on a single

backslash.

## FIREDAC SCALAR FUNCTIONS

As I mentioned at the outset of this section on special filter expressions, FireDAC adds its own collection of scalar functions that you can use in filters (and other expressions in Delphi). There are many more functions available through FireDAC's scalar functions than offered from Delphi's TDataSet

implementation, and you will find tables of these function in *Chapter 14, The SQL Command Preprocessor*.

The one difference between using these functions in filter expressions and using them in SQL statements, is that you do not enclose the functions in curly braces when using them in filter expressions, but you must do so in SQL statements.

`LENGTH()` is a FireDAC scalar string function, and it returns the number of characters in a string. Figure 9-9 shows the `LENGTH()` function being used in

a filter expression in the FDAdvancedFilters project. But let me remind you once more, if you want to use these FireDAC scalar functions in filter expressions, you must include the FireDAC.Stan.ExprFuncs unit in your uses clause.

### Figure 9-9: A FireDAC scalar function being used in a filter expression Other Filter-Related Properties

Assigning a value to the Filter property does not automatically mean that records will be filtered. Only when the Filtered property is set to True does the

The screenshot shows a Delphi IDE interface. At the top, there is a 'Filter Expression' dialog box containing the text 'Company LIKE '%dive%''. Below it are two checkboxes: 'Case Insensitive' (checked) and 'No Partial Compare' (unchecked). Below the dialog is a toolbar with various icons for filtering and sorting. The main area is a dataset grid with columns: CUSTNO, COMPANY, ADDR1, ADDR2, CITY, and STATE. The data in the grid is as follows:

CUSTNO	COMPANY	ADDR1	ADDR2	CITY	STATE
1221	Kauai Dive Shoppe	4-976 Sugarloaf Hwy	Suite 103	Kapaa Kauai	HI
1351	Sight Diver	1 Neptune Lane		Kato Paphos	
1354	Cayman Divers World Unlimited	PO Box 541		Grand Cayman	
1384	VIP Divers Club	32 Main St.		Christiansted	St. Croix
1551	Marmot Divers Club	872 Queen St.		Kitchener	Ontario
2135	Frank's Divers Supply	1455 North 44th St.		Eugene	OR

## 244 Delphi in Depth: FireDAC

Filter property actually produce a filtered dataset. Furthermore, if the Filter property contains an empty string, setting Filtered to True has no effect.

By default, filters are case sensitive and perform a partial match to the filter criteria. You can influence this behavior using the FilterOptions property. This property is a set property that can contain zero or more of the following two flags: foCaseInsensitive and foNoPartialCompare.

Using the default settings for FilterOptions, the following filter will result in no records being displayed in the FDAdvancedFilters project:

Company LIKE '%dive%'

The reason for this is that, due to case sensitivity, no records contain the string dive. However, if you click the Case Insensitive checkbox, the

foCaseInsensitive flag gets added to the FilterOptions property, after which this filter expression will include any company whose name contains the letters dive, regardless of case. This can be seen in Figure 9-10.

### Figure 9-10: Include the foCaseInsensitive flag in FilterOptions to employ case-insensitive filters

When foNoPartialCompare is included in the FilterOptions property, partial matches are excluded from the filtered dataset. When foNoPartialCompare is absent from the FilterOptions property (the default), partial matches are identified by an asterisk (\*) in the last character of your filter criteria. All fields

CUSTNO	COMPANY	ADDR1	ADDR2	CITY	STATE
2156	Davy Jones' Locker	246 South 16th Place		Vancouver	BC
2315	Divers of Corfu, Inc.	Marmoset Place 54		Ayios Matthaios	Corfu
3041	Divers of Blue-green	634 Complex Ave.		Pelham	AL
3055	Diver's Grotto	24601 Universal Lane		Downey	CA
4312	Divers of Venice	220 Elm Street		Venice	FL
5432	Divers-for-Hire	G.O. P Box 91		Suva	

## Chapter 9: Filtering Data 245

whose contents begin with the characters to the left of the asterisk are included in the filter. For example, consider the following filter:

Company = 'D\*'

As long as foNoPartialCompare is absent from the FilterOptions property, this filter will include any company whose name begins with the letter D (and that would include a lowercase D if the foCaseInsensitive flag appears in FilterOptions). This can be seen in Figure 9-11.

**Figure 9-11: By default, the asterisk character (\*) is a wildcard that can appear at the end of a string comparison**

If you add the foNoPartialCompare flag to the FilterOptions property, search comparisons that include the \* character will be taken literally.

### Using the OnFilterRecord Event Handler

There is a second, somewhat more flexible way to define a filter. Instead of using the Filter property, you can attach code to the OnFilterRecord event handler of a FireDAC dataset. When Filtered is set to True, this event handler triggers for every record in the dataset. When called, this event handler is passed a Boolean parameter by reference, named Accept, which you use to indicate

whether or not the current record should be included in the filtered view.

From within this event handler, you can perform almost any test you can imagine. For example, you can verify that the current record is associated with a

## 246 Delphi in Depth: FireDAC

corresponding record in another table. If, based on this test, you wish to exclude the current record from the view, you set the value of the Accept formal

parameter to False. This parameter is True by default.

The Filter property normally consists of one or more comparisons involving values in fields of the dataset. OnFilterRecord event handlers, however, can include any comparison you want. And therein lies the danger. Specifically, if the comparison that you perform in the OnFilterRecord event handler is time consuming, the filter will be slow. In other words, you should try to optimize any code that you place in an OnFilterRecord event handler, especially if you need to filter a lot of records, since it is executed for each and every record in the dataset (with an exception noted in the final section of this chapter, *Using Ranges and Filters Together*).

The following is a simple example of an OnFilterRecord event handler:

```
procedure TForm1.FDMMemTable1FilterRecord(DataSet: TDataSet;  
var Accept: Boolean);  
begin
```

```
Accept := FDMMemTable1.Fields[1].AsString = 'Scarlett';
```

```
end;
```

If you have both a Filter expression and an OnFilterRecord event handler assigned when you set Filtered to True, both will be applied. Importantly, the Filter will not necessarily be applied first, meaning that OnFilterRecord will be executed for some records that do not pass the Filter expression. The bottom line is that you should use Filter or OnFilterRecord, but not both. On the other hand, you can use Filters and Ranges together, as discussed in *Using Ranges and Filters Together*.

## Navigating Using a Filter

Whether you have set Filtered to True or not, you can still use a Filter for the purpose of navigating selected records. For example, although you may want to view all records in a FireDAC dataset, you may want to quickly move among

records that meet specific criteria. For example, you may want to be able to quickly navigate among those records where the customer has a credit limit in excess of (USD) \$5,000.

FireDAC datasets, like other TDataSets, exposes four methods for navigating using a filter. These methods are FindFirst, FindLast, FindNext, and FindPrior.

When you execute one of these methods, the dataset will locate the requested record based on the current Filter property or OnFilterRecord event handler.

## Chapter 9: Filtering Data 247

This navigation, however, does not require that the Filtered property be set to True. In other words, while all records of the dataset may be visible, the filter can be used to quickly navigate among those records that match the filter.

When you execute the methods FindNext or FindPrior, the dataset sets a property named Found. If Found is True, a next record or a prior record was located, and is now the current record. If Found returns False, the attempt to navigate failed. However, all of the filtered navigation methods are function methods that return a Boolean True if the operation was successful. For example, after setting the filter expression, a call to FindFirst or FindLast returns False if no records match the filter expression.

The use of filtered navigation is demonstrated in the event handlers associated with the buttons labeled First, Prior, Next, and Last in the FDFilter project (shown earlier in this chapter). These event handlers are shown here:

```
procedure TForm1.FirstBtnClick(Sender: TObject);
begin
  Start;
  if not FDMemTable1.FindFirst then
    begin
      Complete;
      ShowMessage('There are no matching records');
    end
    else
      Complete;
  end;
```

```
procedure TForm1.PriorBtnClick(Sender: TObject);
begin
  Start;
  if not FDMemTable1.FindPrior then
    begin
      Complete;
      ShowMessage('No prior record found');
    end
  else
    Complete;
  end;
procedure TForm1.NextBtnClick(Sender: TObject);
begin
  Start;
  FDMemTable1.FindNext;
  248 Delphi in Depth: FireDAC
  Complete;
  if not FDMemTable1.Found then
    ShowMessage('No next record found');
  end;
procedure TForm1.LastBtnClick(Sender: TObject);
begin
  Start;
  if not FDMemTable1.FindLast then
    begin
      Complete;
      ShowMessage('There are no matching records');
    end
  else
```

**Complete;**

**end;**

Notice that the NextBtnClick method uses the Found property of the dataset to determine if the navigation was successful. By comparison, the PriorBtnClick simply uses the return value of FindPrior to measure success. The values

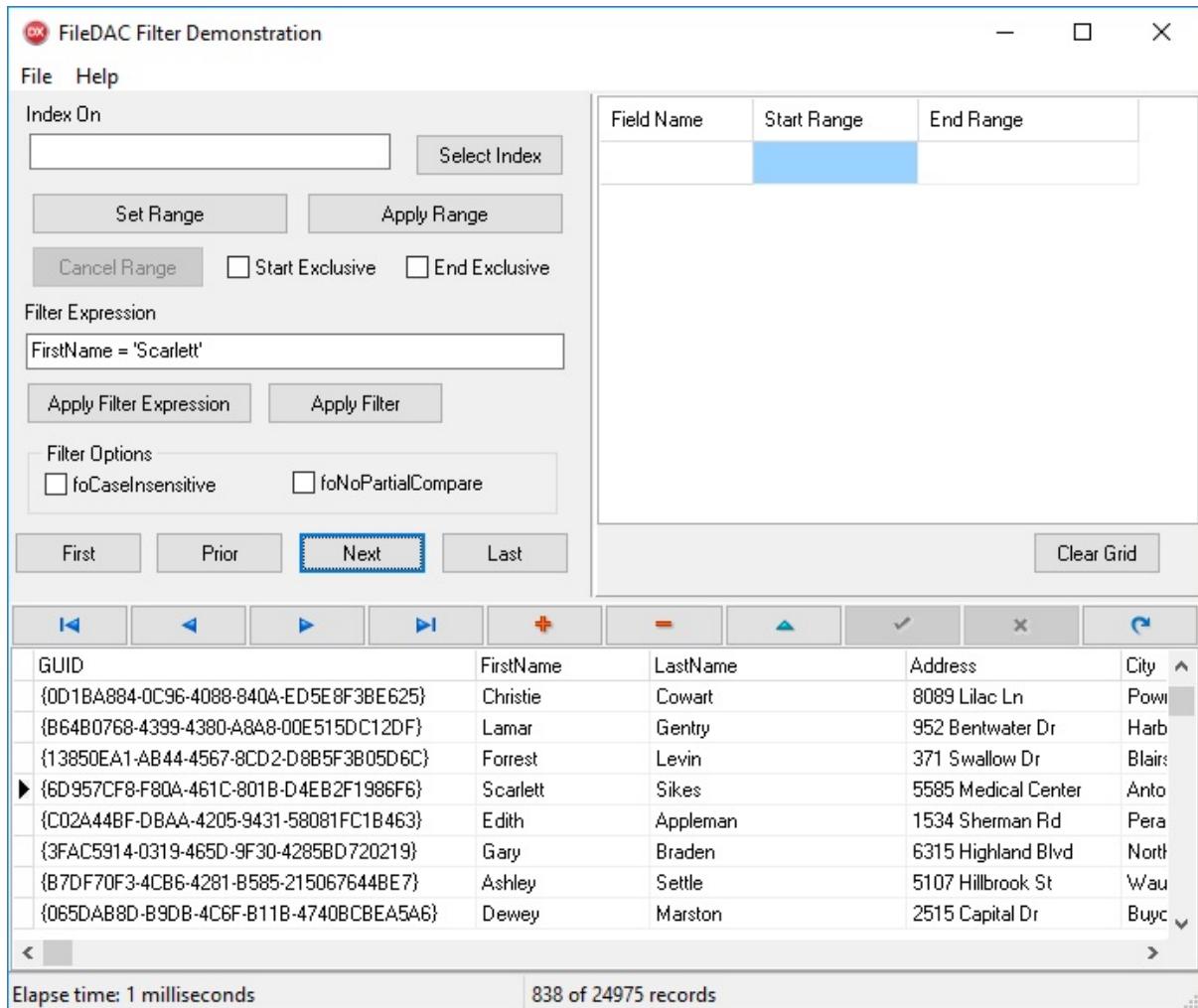
returned by the calls to FindFirst and FindLast are also used in this code to evaluate whether or not any records matched the expression entered into the provided Edit.

Figure 9-12 shows how the FDFilter project looks after a filtered navigation.

After entering the filter expression FirstName = ‘Scarlett’ into the Edit, and clicking the Apply Filter Expression button, the button labeled First was clicked.

This moved the cursor to record number 510. The Next button was then clicked, and the record at position 838 became the current record. The navigation is being performed on records where the first name field contains the value

Scarlett, even though all records from the FDMemTable are visible in the DBGrid.



## Chapter 9: Filtering Data 249

**Figure 9-12: Filtered navigation uses the filter expression to navigate, even when Filtered is set to False**

### Using Ranges and Filters Together

Ranges make use of indexes and are very fast. Filters are slower, but are more flexible. Fortunately, both ranges and filters can be used together. Using ranges with filters is especially helpful when you cannot use a range alone, and your filter is a complicated one that would otherwise take a long time to apply. In those situations, it is best to first set a range, limiting the number of records that need to be filtered to the smallest possible set that includes all records of the filter. The filter can then be applied on the resulting range. Since fewer records need to be evaluated for the filter, the combined operations will be faster than using a Filter alone.

### 250 Delphi in Depth: FireDAC

This advantage of using ranges and filters together is especially useful when you employ the **OnFilterRecord** event handler, which as pointed out earlier,

can negatively affect filter performance. By eliminating as many records initially from a view by using a range, OnFilterRecord executes for a smaller number of records, which in turn will improve the performance of the filter.

## **Master-Detail Views: Dynamically Filtered Detail**

### **Tables**

Most of the data that we work with is stored in relational databases, where related data is stored in more than one table. A good example of this is the employee.gdb database that has been used for many of the examples in this book. There are many tables in this database, and you often have to refer to several different tables to get a complete picture of the data.

For example, the Sales table contains information about sales, including a reference to the customer to whom the sale was made. However, the Sales table only includes the customer number for the customer, and you must refer to the customer table to get information about the customer, including the customer's name and address.

Turning this example on its head, when we look at the customer table we find the customer's name, address, and more, but that table contains no information about sales made to that customer. The Customer table and the Sales table are related, in this case, based on the values that appear in their respective CUST\_NO fields. And this relationship is often referred to as a master-detail relationship, but also referred to as a parent-child relationship or a one-to-many relationship. With the Customer table as the master, the detail records are the zero or more Sales table records associated with one particular customer.

While we can write a query that joins the data from these related tables, it is common for users to want to view these relationships without them being joined. A good example of this can be seen in Figure 9-13, in which the Customer records are displayed in one grid, and the sales associated with the current record in the customer table are displayed in another grid. In other words, the detail table records are filtered, based on data found in the current master table record. This is what is meant by a master-detail relationship.

Master-Detail Demonstration

Range-Based Parameter-Based

The screenshot shows a Delphi application window titled "Master-Detail Demonstration". It features two main sections: "Range-Based" and "Parameter-Based". The "Range-Based" section is active, displaying two tables. The top table, "Customer", has columns: CUST\_NO, CUSTOMER, CONTACT\_FIRST, CONTACT\_LAST, and PHONE\_NO. The bottom table, "Sales", has columns: PO\_NUMBER, CUST\_NO, SALES\_REP, ORDER\_STATUS, ORDER\_DATE, and SHIP\_DATE. A primary key constraint is visible between the CUST\_NO column of the Customer table and the CUST\_NO column of the Sales table. Navigation buttons for both tables are present at the top and bottom of each table's grid.

CUST_NO	CUSTOMER	CONTACT_FIRST	CONTACT_LAST	PHONE_NO
1001	Signature Design, Ltd.	Dale J.	Little	(619) 530-2710
1002	Dallas Technologies	Glen	Brown	(214) 960-2233
1003	Buttle, Griffith and Co.	James	Buttle	(617) 488-1864
1004	Central Bank	Elizabeth	Brocket	61 211 99 88

PO_NUMBER	CUST_NO	SALES_REP	ORDER_STATUS	ORDER_DATE	SHIP_DATE
V9324200	1001	72	shipped	8/9/2012	8/9/2012
V9324320	1001	127	shipped	8/16/2012	8/16/2012
V9320630	1001	127	open	12/12/2012	
V9420099	1001	127	open	1/17/2013	
V9427029	1001	127	shipped	2/7/2013	2/10/2013

## Chapter 9: Filtering Data 251

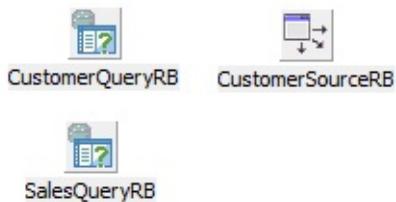
**Figure 9-13: Data from two related tables are displayed in a master-detail relationship**

*Code: The code project FDMasterDetail is available from the code download.*

What's especially interesting about the data displayed in Figure 9-13 is that this association between the datasets is defined using properties. There is no custom code being executed to produce and maintain the filtered view displayed in the detail table. If you run this project and navigate to a new current record in the Customer table the Sales table view will update automatically. This is what is referred to as a *dynamic master-detail relationship*.

There are two ways in Delphi to define a dynamic master-detail relationship between two datasets, and in this case, I am going to specifically describe how this is done using FireDAC datasets. Other datasets also support these techniques, but there may be minor differences in the specific properties that you use.

The two approaches to defining dynamic master-detail relationships are referred to as *range-based* and *parameter-based*. Range-base dynamic master-detail relationships can be implemented by any type of dataset in the detail table, while



## 252 Delphi in Depth: FireDAC

parameter-based dynamic master-detail relationships can only be used when the detail table is an FDQuery or FDStoredProc.

*Note: Master-detail relationships can also be created, sometimes very efficiently, through the use of nested datasets. Nested datasets are discussed in detail in Chapter 13: More FDMemTables: Cloned Cursors and Nested DataSets.*

### Defining a Range-Based Dynamic Master-Detail Relationship

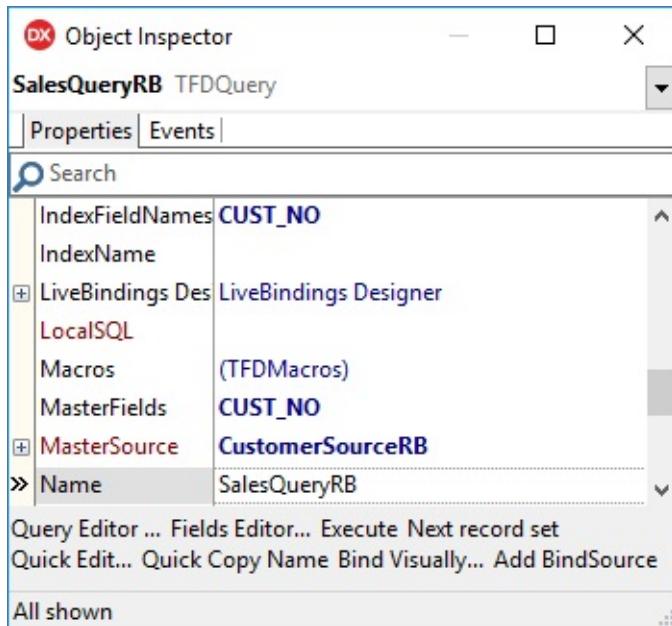
Range-based dynamic master-detail relationships can be defined at design time or at runtime, and doing so involves the same set of steps. These are:

- ▀ The master dataset and detail dataset must be related based on one or more fields. Typically the relationship is that the field or fields are the primary key field(s) in the master table, and the fields are a foreign key in the detail table.
- ▀ There must be a data source that points to the master dataset.
- ▀ The MasterSource property of the detail dataset must point to this data source.
- ▀ The MasterFields property of the detail dataset must define the fields on which the relationship is based.
- ▀ The detail dataset must be indexed on the fields in the MasterFields property.

The form shown in Figure 9-13 is found in the FDMasterDetail project. This project has two tabs on a page control, where the first tab is used to display data defined by a dynamic range-based master detail relationship. Here are the three components that participate in this relationship.

CustomerQueryRB is an FDQuery that holds the following SQL statement:

```
SELECT * FROM Customer
```



## Chapter 9: Filtering Data 253

And SalesQueryRB is an FDQuery that holds this SQL statement:

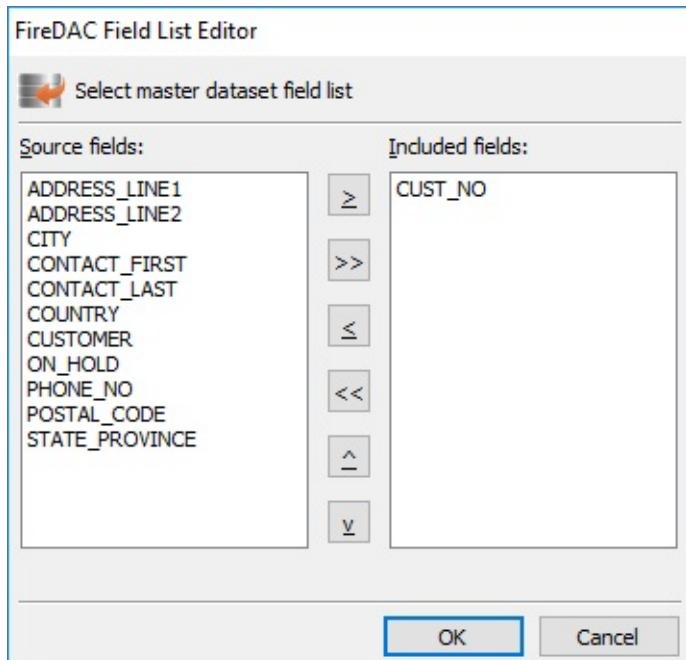
SELECT \* FROM Sales

Furthermore, CustomerSourceRB is a DataSource that points to CustomerQueryRB.

The relationship is defined by properties associated with SalesQueryRB, the detail dataset in this relationship. MasterSource points to CustomerSourceRB, MasterFields points to CUST\_NO (a foreign key on Sales and the primary key on Customer), and IndexFieldNames is set to CUST\_NO. These properties can

be seen here.

You select MasterSource using the dropdown list on the MasterSource property to select from the data-source objects in scope, and you enter the IndexFieldNames property manually (or select from an appropriate FDIndex that you have created). You can enter the value for the MasterFields property manually, or you can invoke the property editor for this property. The MasterFields property editor for FireDAC datasets is shown in Figure 9-14.



## 254 Delphi in Depth: FireDAC

**Figure 9-14: The MasterFields property editor has been used to select the primary key of the dataset to which the MasterSource property refers**

That's it. It really is that simple.

### Defining a Parameter-Based Dynamic Master-Detail

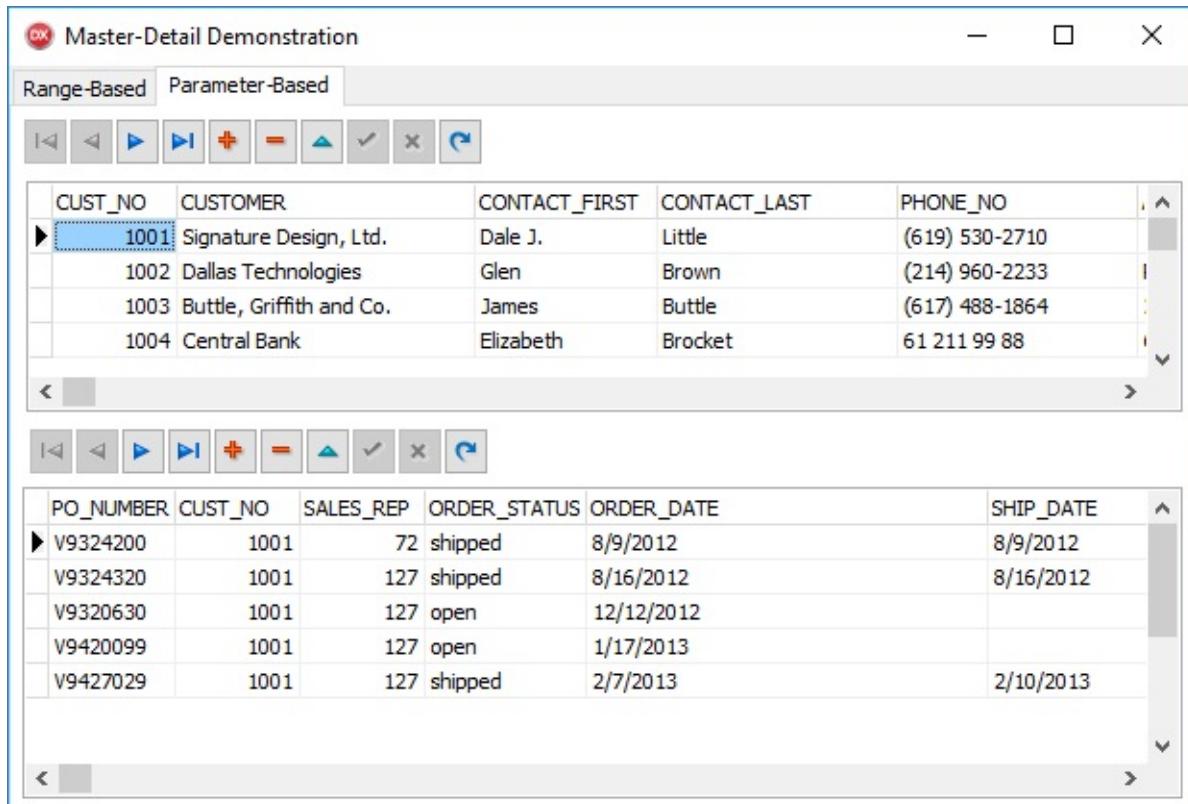
#### Relationship

While you can define a range-based dynamic master-detail relationship between any two FireDAC datasets (and more generally, between any FireDAC master

dataset and any Delphi detail dataset), a parameter-based dynamic master-detail relationship must have a detail dataset that can make use of a parameterized FireDAC dataset, in other words, an FDQuery or an FDStoredProc.

In many respects, defining a parameter-based dynamic master-detail relationship is easier than a range-based relationship. You perform the same steps that you do when defining a range-based master-detail relationship, with one wrinkle.

Your detail dataset must include a query that includes a WHERE clause that compares the fields of the detail table's foreign key fields to named parameters, where the names of the parameters correspond to the primary key fields of the



## Chapter 9: Filtering Data 255

master dataset. In the case of a relationship between the Customer and Sales tables, this query looks like the following:

```
SELECT * FROM Sales WHERE CUST_NO = :cust_no;
```

Since CUST\_NO is the name of the Customer table's primary key, the preceding query includes a parameter named :cust\_no. I have deliberately used lowercase for the parameter name to emphasize that the parameter name is case insensitive.

If you have defined the query prior to setting the MasterSource property of your detail FDQuery, FireDAC will fill in the MasterFields property for you, but you will still need to define an index based on the detail table's foreign key field or fields, which is done using IndexFieldNames in this case.

Figure 9-15 shows the running FDMasterDetail project with the Parameter-Based tab selected. The two grids display the data from two FDQueries that are configured using parameter-based dynamic master-detail properties.

**Figure 9-15: The Sales query values are dynamically filtered based on a parameterized query**

256 Delphi in Depth: FireDAC

If you run this project, and navigate between records of either of the master

datasets, you will see the detail datasets dynamically filter to display only those records associated with its current master record. And, this happens based on properties alone. There is no custom code in the project that participates in this operation, which you can see from the following, which shows the entire source code for the main form unit:

**unit** mainformu;

**interface**

**uses**

Winapi.Windows, Winapi.Messages, System.SysUtils,  
System.Variants, System.Classes, Vcl.Graphics,  
Vcl.Controls, Vcl.Forms, Vcl.Dialogs, FireDAC.Stan.Intf,  
FireDAC.Stan.Option, FireDAC.Stan.Param, FireDAC.Stan.Error,  
FireDAC.DatS, FireDAC.Phys.Intf, FireDAC.DApt.Intf,  
FireDAC.Stan.Async, FireDAC.DApt, Data.DB, Vcl.Grids,  
Vcl.DBGrids, Vcl.ExtCtrls, Vcl.DBCtrls, FireDAC.Comp.DataSet,  
FireDAC.Comp.Client, Vcl.ComCtrls;

**type**

TMainForm = class(TForm)  
PageControl1: TPageControl;  
TabSheet1: TTabSheet;  
TabSheet2: TTabSheet;  
CustomerQueryRB: TFDQuery;  
SalesQueryRB: TFDQuery;  
CustomerQueryPB: TFDQuery;  
SalesQueryPB: TFDQuery;  
DBNavigator1: TDBNavigator;  
DBGrid1: TDBGrid;  
DBNavigator2: TDBNavigator;  
DBGrid2: TDBGrid;  
DBNavigator3: TDBNavigator;

```
grdCustomerRB: TDBGrid;
DBNavigator4: TDBNavigator;
grdSalesRB: TDBGrid;
CustomerSourceRB: TDataSource;
CustomerSourcePB: TDataSource;
dscCustomerPB: TDataSource;
dscSalesPB: TDataSource;
Chapter 9: Filtering Data 257
dscCustomerRB: TDataSource;
dscSalesRB: TDataSource;
procedure FormCreate(Sender: TObject);
private
{ Private declarations }

public
{ Public declarations }

end;

var
MainForm: TMainForm;

implementation
{$R *.dfm}

uses SharedDMVclU;

procedure TMainForm.FormCreate(Sender: TObject);

begin
CustomerQueryRB.Open();
SalesQueryRB.Open();
CustomerQueryPB.Open();
SalesQueryPB.Open();

end;

end.
```

*Note: You may have noticed that I have used a total of six data source objects in this project, where two different data sources are pointing to each of the master datasets. While I could have used a single data source to point to the master datasets, and referenced it by both the detail dataset and the DBGrid, I used two, since each of them serve a different purpose. When creating a dynamic master-detail relationship, I like to use separate data sources for the MasterSource property and the wiring of data-aware controls to the datasets, since these two roles are distinctly different.*

258 Delphi in Depth: FireDAC

### **Detail Records and the `FetchOptions.DetailDelay` Property**

The employee.gdb database that is used in the FDMasterDetail project is a small one. The Customer table contains only 15 records, and the Sales table has but 33. As a result, we can scroll through the customer records, and the associated sales records are filtered very quickly.

If you create a dynamic master-detail view with very large tables, however, scrolling the master table records can be painfully slow as FireDAC attempts to load all (or some) of the detail records before being able to navigate to the next master table record.

Fortunately, FireDAC has a solution for this — the `FetchOptions.DetailDelay` property. You use this property to instruct FireDAC to wait some number of milliseconds before attempting to load the detail records after arriving on a master table record. If this delay is sufficiently long, an end user can scroll through master table records and no detail records will be loaded until the user stops scrolling.

The default value of `DetailDelay` is 0, in which case detail records are loaded as the user arrives at each master table record. Set `DetailDelay` to a positive integer to delay the loading of detail records until the user has stopped on a particular master table record. What value you assign to `DetailDelay` will depend on how quickly the user can scroll master table records, and you may even want this value to be configurable on a user-by-user basis.

In the next Chapter, I am going to discuss virtual fields.

Chapter 10: Virtual Fields 259

# Chapter 10

## Creating and Using

### Virtual Fields

I introduced the concept of fields in *Chapter 6, Navigating and Editing Data*. In that discussion, I demonstrated that fields could be used to read from, and sometimes write to, the columns returned in a dataset, as well as access metadata about those columns. Those fields were a particular type of field, referred to as a *Data field*. Data fields are always associated with the data returned from query or stored procedure calls, or data loaded from a file or stream, and there is always a one-to-one association between a particular field and a column in the dataset.

There is another category of field, called *virtual fields*. Unlike Data fields, virtual fields don't necessarily derive directly from an operation against a database or a file or stream. Instead, they result from some operation or calculation. For example, they result from the execution of your code that performs an assignment to the field, or from an operation that you have configured in FireDAC to produce a summary calculation.

More importantly, the values of virtual fields are never written to the underlying database. These values are virtual — they belong to the dataset in memory, but are not part of a result set in the same way that Data fields are. (InternalCalc fields, a type of Calculated field, are an exception, in that they can be persisted to a file or stream. This is discussed later in this chapter, as well in *Chapter 11, Persisting Data*.)

There are three general categories of virtual fields: Aggregate fields, Calculated fields, and Lookup fields. We will explore virtual fields in this chapter along with two related topics. Towards the end of this chapter, I discuss FieldOptions, a new property introduced in Delphi XE6 that gives you more control over Data fields and virtual fields. I also show you how to use the GroupState method, which is closely associated with Aggregate fields, the first type of virtual field that we will consider.

### 260 Delphi in Depth: FireDAC

Before I continue, I need to point out another way in which the various fields in Delphi differ. Fields can also be categorized as being either *dynamic fields* or *persistent fields*. Dynamic fields are always Data fields, meaning that they

are associated with a column of data referred to by a dataset. Dynamic fields are created dynamically, at runtime, when the dataset becomes active, and one dynamic field is created at runtime for each column in the corresponding result set.

Persistent fields, by comparison, are created at design time, and persist from the design-time environment to the runtime environment. Both Data and virtual

fields can be created as persistent fields, and when they are created, they appear in the published section of your form, frame, or data module, like that seen in the following partial type declaration:

**type**

```
TForm1 = class(TForm)
  Panel1: TPanel;
  DBGrid1: TDBGrid;
  DBNavigator1: TDBNavigator;
  DataSource1: TDataSource;
  DataSource2: TDataSource;
  FDQuery1: TFDQuery;
  FDQuery2: TFDQuery;
  FDConnection1: TFDCOnnection;
  FDQuery1ORDERNO: TFloatField;
  FDQuery1CUSTNO: TFloatField;
  FDQuery1SALEDATE: TSQLTimeStampField;
  FDQuery1SHIPDATE: TSQLTimeStampField;
  FDQuery1EMPNO: TIntegerField;
  FDQuery1SHIPTOCONTACT: TStringField;
  FDQuery1SHIPTOADDR1: TStringField;
```

At runtime, there is no functional difference between whether a given field is a dynamic field or a persistent field — you can use both to read data and sometimes write data, and both can be used to inspect metadata associated with the field. However, there are times when the distinction between persistent and dynamic fields becomes important, as you will learn later in this chapter.

## Aggregate Fields

An aggregate is an object that can automatically perform a simple descriptive statistical calculation across one or more records in a dataset. For example,

Chapter 10: Virtual Fields 261

imagine that you have an FDQuery that contains a list of all purchases made by your customers. If each record contains fields that identify the customer, the number of items purchased, and the total value of the purchase, an aggregate can calculate the sum of all purchases across all records in the table. Another aggregate can calculate the average number of items purchased by each

customer, and a third aggregate can calculate the average cost of a given customer's items.

FireDAC dataset aggregates support a total of five statistics. These are: Count, minimum, maximum, sum, and average.

There are two types of objects that you can use to create aggregates: Aggregates and AggregateFields. An Aggregate is a CollectionItem descendant, and an

AggregateField is a descendant of the TField class.

While these two aggregate types are similar in how you configure them, they differ in their use. Specifically, an AggregateField, because it is a TField descendant, can be associated with data-aware controls (and LiveBindings), permitting the aggregated value to be displayed automatically. By comparison, an Aggregate is an object whose value must be explicitly read at runtime.

One characteristic shared by both types of aggregates is that they require quite a few specific steps to configure them. If you have never used aggregation in the past, be patient. If your aggregates do not appear to work at first, you probably missed one or more steps in their configuration. However, after you get

comfortable configuring aggregates, you will find that they are relatively easy to use.

Because AggregateField instances are somewhat easier to use, I will consider them first. Using Aggregates is discussed later in this chapter.

*Note: Before the introduction of FireDAC, ClientDataSets were the only other TDataSet included with Delphi to support AggregateFields, Aggregates, and GroupState. For more information on ClientDataSets, please refer to my*

*last book*, Delphi in Depth: ClientDataSets, Second Edition, 2015.

## Creating Aggregate Fields

AggregateFields are virtual, persistent fields. While they are similar to other virtual, persistent fields, such as Data, Calculated and Lookup fields, they have one very important difference. Specifically, introducing one or more Aggregate fields does not automatically preclude the runtime creation of dynamic fields.

### 262 Delphi in Depth: FireDAC

By comparison, by default, creating at least one other type of persistent field, such as a persistent Data field, a Lookup field, or a Calculated field, prevents the dataset from creating dynamic fields for that dataset at runtime. While this description of the default behavior is correct, with the release of Delphi XE6, a new property named FieldOptions, was introduced which permits you to configure this default behavior, and in some cases, permits persistent fields and dynamically created fields to co-exist. This issue is explored in depth later in this chapter in the section titled *Understanding FieldOptions*.

*Note: The code project FDAGgregatesAndGroupState is available from the code download. See Appendix A for details.*

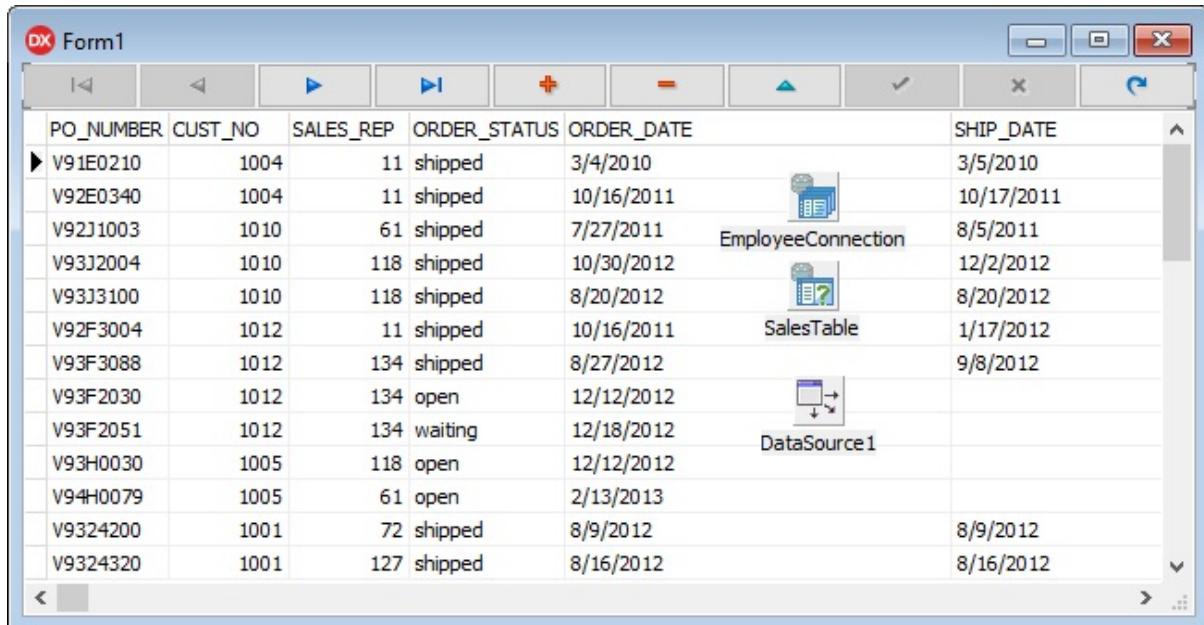
As mentioned earlier, adding an aggregate requires a number of specific steps in order to configure it correctly. These are:

- ▀ Add an AggregateField or an Aggregate collection item to a FireDAC dataset. AggregateFields can be added at design time using the Fields Editor, or at runtime using the AggregateField's constructor. Aggregate collection items are added using the Aggregates property editor at design time, or by calling the Aggregate's constructor at runtime.
- ▀ Set the aggregate's Expression property to define the calculation that the aggregate will perform.
- ▀ Set the aggregate's IndexName property to identify the index on which to base grouping level. If you have set the IndexName property for the dataset to which the aggregate belongs, you can likely skip this step.
- ▀ Set the aggregate's GroupingLevel property to identify across which records to perform the aggregation.
- ▀ Set the aggregate's Active property to True to activate it.

- ▀ Set the AggregatesActive property of the dataset to which the aggregate is associated to True.

Because there are so many steps here, and some of them must be performed before others, it is best to learn how to create an AggregateField by doing it. Use the following steps in Delphi to create a simple project to which you will add an AggregateField:

1. Create a new VCL Forms Application project.

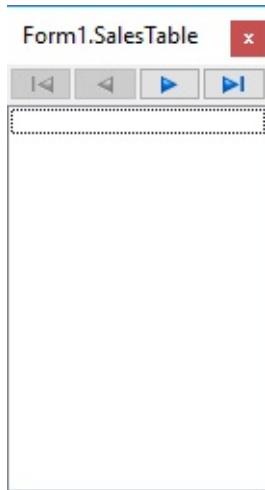


## Chapter 10: Virtual Fields 263

2. Using the Data Explorer, expand the FireDAC node, and then the InterBase node. Now expand the Employee node, and from there, the Tables node. Drag the node for the Sales table and drop it onto the form. Delphi should create a configured FDConnection on the form named EmployeeConnection, as well as a configured FDQuery named SalesTable.
3. Next, add to your main form a DBNavigator, a DBGrid, and a DataSource.
4. Set the Align property of the DBNavigator to alTop, and the Align property of the DBGrid to alClient.
5. Next, set the DataSource property of both the DBNavigator and the DBGrid to DataSource1.

6. Now set the DataSet property of DataSource1 to SalesTable.
7. Test that everything is working fine by setting the SalesTable's Active property to True. Your form should look something like that shown in Figure 10-1.
8. Delphi's documentation suggests that you configure your Aggregate field's expressions while your FireDAC dataset is inactive, so set the SalesTable's Active property to False before continuing.

**Figure 10-1: The initial application for creating an aggregate field**



## 264 Delphi in Depth: FireDAC

*Note: I have been able to create aggregate fields on an active FDQuery. However, I have also ran into difficulty when doing so. As a result, I follow the advice of the documentation and only create aggregate fields at design time with the FireDAC dataset inactive.*

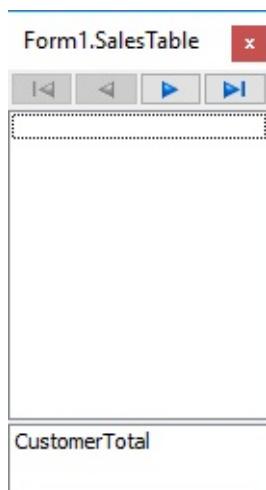
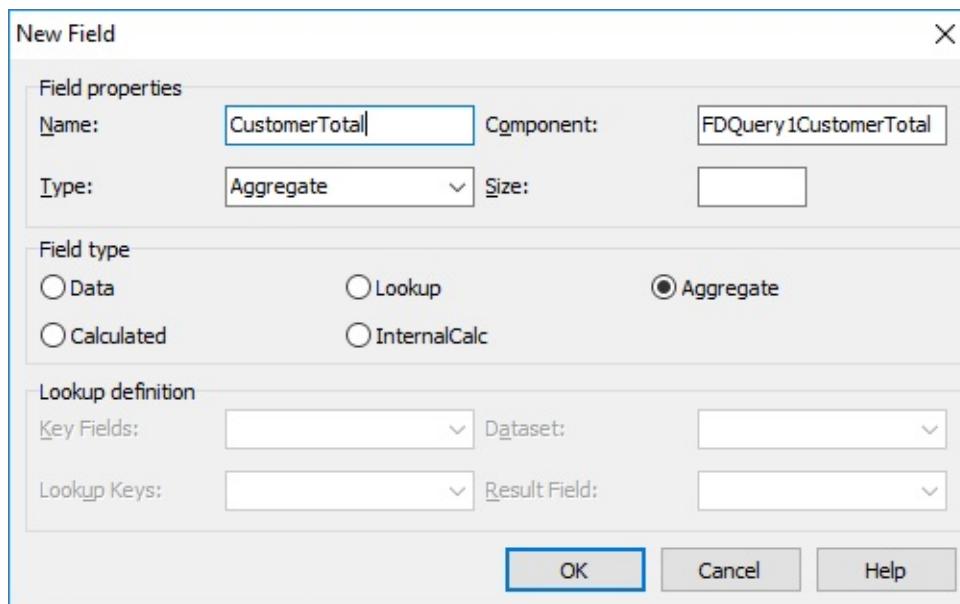
### ADDING THE AGGREGATE FIELD

You add an AggregateField to a FireDAC dataset at design time using the dataset's Fields Editor. Use the following steps to add an aggregate field:

1. Right-click SalesTable and select Fields Editor to display the Fields Editor, shown here:

2. Right-click the Fields Editor and select New Field (or press Ctrl-N). Delphi displays the New Field dialog box.
3. At Name, enter CustomerTotal and select the Aggregate radio button in the Field type area. Your New Field dialog box should now look

something like that shown in Figure 10-2.



## Chapter 10: Virtual Fields 265

**Figure 10-2: A new virtual AggregateField is being defined in the New Field Editor**

4. Click OK to close the New Field dialog box. You will see the newly added aggregate field in the Fields Editor, as shown here:

266 Delphi in Depth: FireDAC

Notice that the newly added CustomerTotal field appears in its own little window at the bottom of the Fields Editor. All AggregateFields appear in this window, which serves to separate AggregateFields from any other persistent fields, virtual or Data. As mentioned earlier in this chapter, this distinction is that the presence of persistent AggregateFields does not by default preclude the automatic creation of dynamic Data fields.

## DEFINING THE AGGREGATE EXPRESSION

The Expression property of an aggregate defines the calculation the aggregate will perform. This expression can consist of literals, field values, and aggregate functions. The aggregate functions are AVG, MIN, MAX, SUM, and COUNT.

For example, to define a calculation that will total the TOTAL\_VALUE field in the dataset, you use the following expression:

SUM(TOTAL\_VALUE)

The argument of the aggregate function can include two or more fields in an expression, if you like. For example, if you have two fields in your table, one named Quantity and the other named Price, you can use the following expression:

SUM(Quantity \* Price)

The expression can also include literals. For example, if the tax rate is 8.25%, you can create an aggregate that calculates the total plus tax, using something similar to this:

SUM(Total \* 1.0825)

You can also set the Expression property to perform an operation on two aggregate functions, as shown here:

MIN(SHIP\_DATE) - MIN(ORDER\_DATE)

Chapter 10: Virtual Fields 267

In addition, you can perform an operation between an expression function and a literal, as in the following:

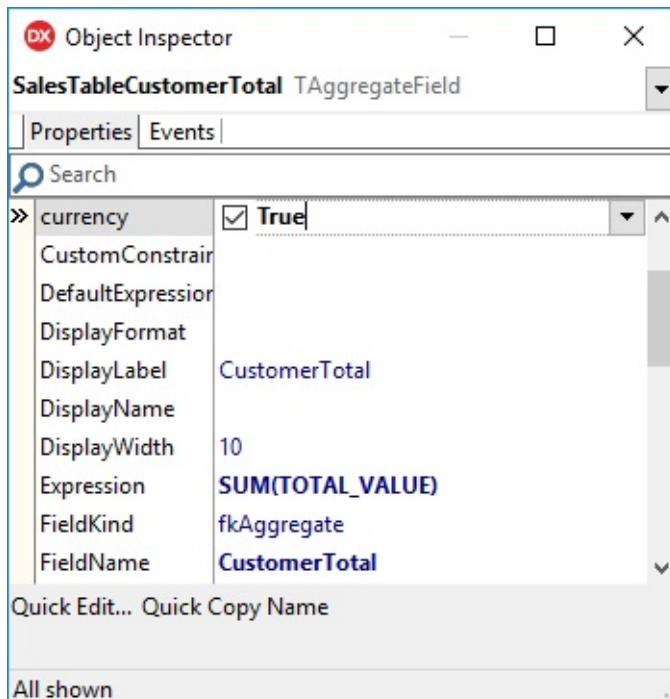
MAX(ShipDate) + 30

You cannot, however, include an aggregate function as the expression of another aggregate function. For example, the following is not a valid expression: SUM(AVG(AmountPaid)) //invalid

Nor can you use an expression that contains a calculation between an aggregate function and a field. For example, if Quantity is the name of a field, the following expression is invalid (since an aggregate is performed across one or more records, by definition, but the field reference applies to single records): SUM(Price) \* Quantity //invalid

In the case of the CustomerTotal AggregateField, we want to calculate the sum of the TOTAL\_VALUE field. To do this, use the following steps:

1. Select the AggregateField in the Fields Editor. By default, this field should have the name SalesTableCustomerTotal.
2. Using the Object Inspector, set the Expression property to SUM(TOTAL\_VALUE) and its Currency property to True. Your Object Inspector should now look something like that shown in Figure 10-3.



268 Delphi in Depth: FireDAC

**Figure 10-3: An AggregateField is being configured to calculate the total of the TOTAL\_VALUE field**

## SETTING AGGREGATE INDEX AND GROUPING LEVEL

An aggregate needs to know across which records it will perform the calculation. This is done using the IndexName and GroupingLevel properties of the aggregate. Actually, if you want to perform a calculation across all records in a dataset, you can leave IndexName blank and GroupingLevel set to 0.

If you want the aggregate to perform its calculation across groups of records, you must have a persistent index whose left-most fields define the group. For example, if you want to calculate the sum of the TOTAL\_VALUE field separately for each customer, and a customer is identified by a field name CUST\_NO, you must set IndexName to the name of a persistent index whose first field is CUST\_NO. If you want to perform the calculation for each

customer for each order date, and you have fields named CUST\_NO and ORDER\_DATE, you must set IndexName to the name of a persistent index that

has CUST\_NO and ORDER\_DATE as its first two fields (the order of these fields in the index is irrelevant).

## Chapter 10: Virtual Fields 269

The persistent index whose name you assign to the IndexName property can have more fields than the number of fields you want to group on. This is where the aggregate's GroupingLevel comes in. You set GroupingLevel to the number of fields of the index that you want to treat as a group.

The following steps walk you through the process of creating a persistent index on the CUST\_NO field, and then setting the AggregateField to use this index with a grouping level of 1:

1. Select the SalesTable in the Object Inspector and select its Indexes property. Click the ellipsis button of the Indexes property to display the FDIndex collection editor.
2. Click the Add New button in the FDIndex collection editor toolbar to add a new persistent index.
3. With the new index selected in the collection editor, use the Object Inspector to set the Name property of this index to CustIdx, its Fields property to CUST\_NO, and its Active property to True. You can now close the FDIndex collection editor.
4. With SalesTable still selected, set its IndexName property to CustIdx.
5. Next, using the Fields Editor, once again select the AggregateField. Since we've already set the IndexName property of the FDQuery to CustIdx, we do not have to touch the aggregate field's index property. Instead, set the GroupingLevel property to 1. You might also want to set the Alignment property of this field to taRightJustified, as it is a currency value, and this will make it look better.

## **MAKING THE AGGREGATE FIELD AVAILABLE**

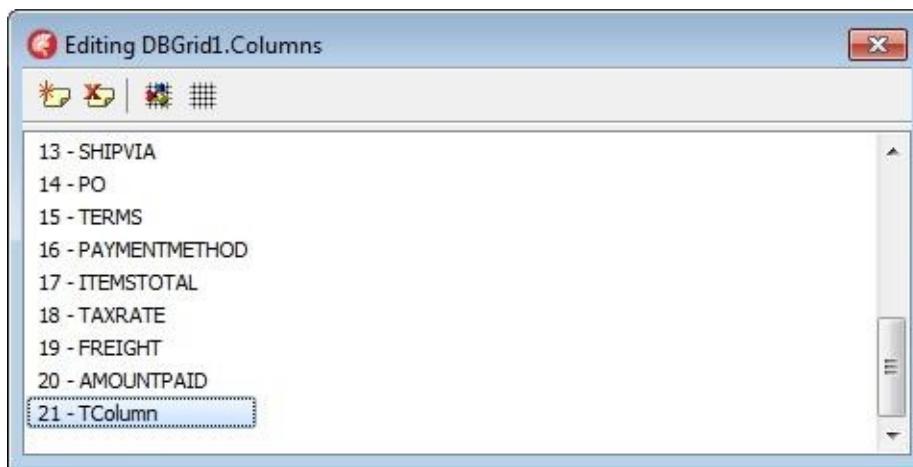
The AggregateField is almost ready. In order for it to work, however, you

must set the AggregateField's Active property to True. In addition, you must set the FireDAC dataset's AggregatesActive property to True. After doing this, the aggregate will automatically be calculated when the dataset is made active.

With AggregateFields, there is one more step than required with Aggregates, which is to associate the AggregateField with a data-aware control or LiveBinding (if you want to display the data in the user interface).

The following steps demonstrate how to activate the AggregateField, as well as make it visible in the DBGrid:

1. With the AggregateField selected in the Object Inspector, set its Active property to True.



## 270 Delphi in Depth: FireDAC

2. Next, select the FDQuery (SalesTable) and set its AggregatesActive property to True, and set its Active property to True. At this point, the aggregate value is being calculated for each customer in the dataset. However, it is not yet visible in the DBGrid. The following steps will make the new AggregateField visible.
3. Right-click the DBGrid and select Columns to display the Columns collection editor.
4. Click the Add All button on the Columns collection editor toolbar to add persistent columns for each dynamic field in the dataset. If you were to scroll to the bottom of the Columns collection editor, you will notice that the CustomerTotal field (our aggregate field) was not added. That is something that you will have to do manually.

5. Click the Add New button on the Columns collection editor toolbar to add one more Column to the collection.

6. With this new Column selected, use the Object Inspector to set its FieldName property to CustomerTotal. Setting this property also has the side effect of changing the name of the new Column in the Columns collection editor. Next, change the position of this Column in the Columns collection editor by dragging it to the fourth position, immediately below the SALES\_REP Column, as shown here:

The screenshot displays two windows. The top window is titled "Editing DBGrid1.Columns" and shows a list of columns with their names and indices: 0 - PO\_NUMBER, 1 - CUST\_NO, 2 - SALES\_REP, 3 - CustomerTotal, 4 - ORDER\_STATUS, 5 - ORDER\_DATE, 6 - SHIP\_DATE, and 7 - DATE\_NEEDED. The "CustomerTotal" column is highlighted. The bottom window is titled "Form1" and shows a DBGrid control displaying data. The grid has columns: PO\_NUMBER, CUST\_NO, SALES\_REP, CustomerTotal, ORDER\_STATUS, ORDER\_DATE, and SHIP\_DATE. The "CustomerTotal" column is visible in the fourth position, as indicated by the column headers.

PO_NUMBER	CUST_NO	SALES_REP	CustomerTotal	ORDER_STATUS	ORDER_DATE	SHIP_DATE
V9324200	1001	72	\$1,045,610.12	shipped	8/9/2012	8/9/
V9324320	1001	127	\$1,045,610.12	shipped	8/16/2012	8/16
V9320630	1001	127	\$1,045,610.12	open	12/12/2012	
V9420099	1001	127	\$1,045,610.12	open	1/17/2013	
V9427029	1001	127	\$1,045,610.12	shipped	2/7/2013	2/10
V9333005	1002	11	\$35,450.50	shipped	2/4/2012	3/3/
V9333006	1002	11	\$35,450.50	shipped	4/27/2012	5/2/
V9336100	1002	11	\$35,450.50	waiting	12/27/2012	1/1/
V9346200	1003	11	\$39,582.12	waiting	12/31/2012	
V9345200	1003	11	\$39,582.12	shipped	11/11/2012	12/2
V9345139	1003	127	\$39,582.12	shipped	9/9/2012	9/20
V91E0210	1004	11	\$75,000.00	shipped	3/4/2010	3/5/
V92E0340	1004	11	\$75,000.00	shipped	10/16/2011	10/1

## Chapter 10: Virtual Fields 271

If you followed all of these steps correctly, your newly added AggregateField should be visible in the fourth column of your DBGrid, as shown in Figure 10-4.

**Figure 10-4: The configured AggregateField appears next to the**

## **SALES\_REP field in the DBGrid**

272 Delphi in Depth: FireDAC

*Code: A saved version of this project can be found in the FDAggregateDemo project in the code download.*

### **TURNING AGGREGATES ON AND OFF**

A couple of additional comments about active aggregates are in order. First, the FireDAC dataset's AggregatesActive property is one that you might find yourself turning on and off at runtime. Setting AggregatesActive to False is extremely useful when you must add, remove, or change a lot of records at runtime. If you make changes to a dataset's data, and these changes affect the aggregate calculations, these changes will be much slower if AggregatesActive is True since the aggregate calculations will be updated with each and every change. As a result, you may want to set AggregatesActive to False before

changing a large number of records. When you later set AggregatesActive to True, any aggregates associated with that dataset will then be recalculated, but just this one time.

Rather than turning all aggregates off or on, the Active property of individual aggregates can be manipulated at runtime. This can be useful if you have many aggregates, but only need one or two to be updated during changes to the

dataset. Subsequently turning other aggregates back on will immediately trigger their recalculation.

### **Creating Aggregate Collection Items**

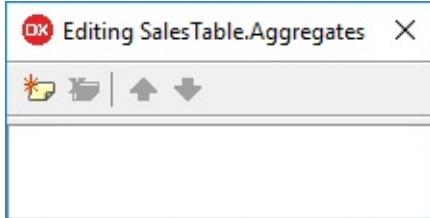
Aggregate collection items, like aggregate fields, perform the automatic calculation of simple descriptive statistics. However, unlike AggregateFields, Aggregates must be read at runtime in order to use their values, as opposed to being bindable to data-aware controls.

With the one exception that Aggregate collection items cannot be hooked up to data-aware controls, nearly all other aspects of the configuration of aggregate collection items are the same as for AggregateFields.

The following steps demonstrate how to add and use an Aggregate collection item in a project. These steps assume that you have been following along with the steps provided earlier to define the AggregateField:

1. Select the FDQuery in the Object Inspector and set its Active property to

False. Next, select its Aggregates property, and click the displayed ellipsis button to display the Aggregates collection editor:



Chapter 10: Virtual Fields 273

2. Click the Add New button twice on the Aggregates collection editor's toolbar to add two aggregates to your dataset.
3. Select the first Aggregate in the Aggregates collection editor. Using the Object Inspector, set the aggregate's Expression property to `AVG(TOTAL_VALUE)`, its AggregateName property to `CustAvg`, its IndexName property to `CustIdx`, its GroupingLevel property to 1, and its Active property to True.
4. Select the second Aggregate in the Aggregates collection editor. Using the Object Inspector, set its Expression property to `MIN(ORDER_DATE)`, its AggregateName property to `FirstOrder`, its IndexName property to `CustIdx`, its GroupingLevel property to 1, and its Active property to True.
5. You can now set the FDQuery's Active property to True.
6. Add a PopupMenu from the Standard page of the Tool Palette to your project. Using the Menu Designer (double-click the PopupMenu to display this editor), add a single MenuItem, setting its caption to "**About this customer.**"
7. Set the PopupMenu property of the DBGrid to `PopUpMenu1`.
8. Finally, add the following event handler to the Add this customer MenuItem:

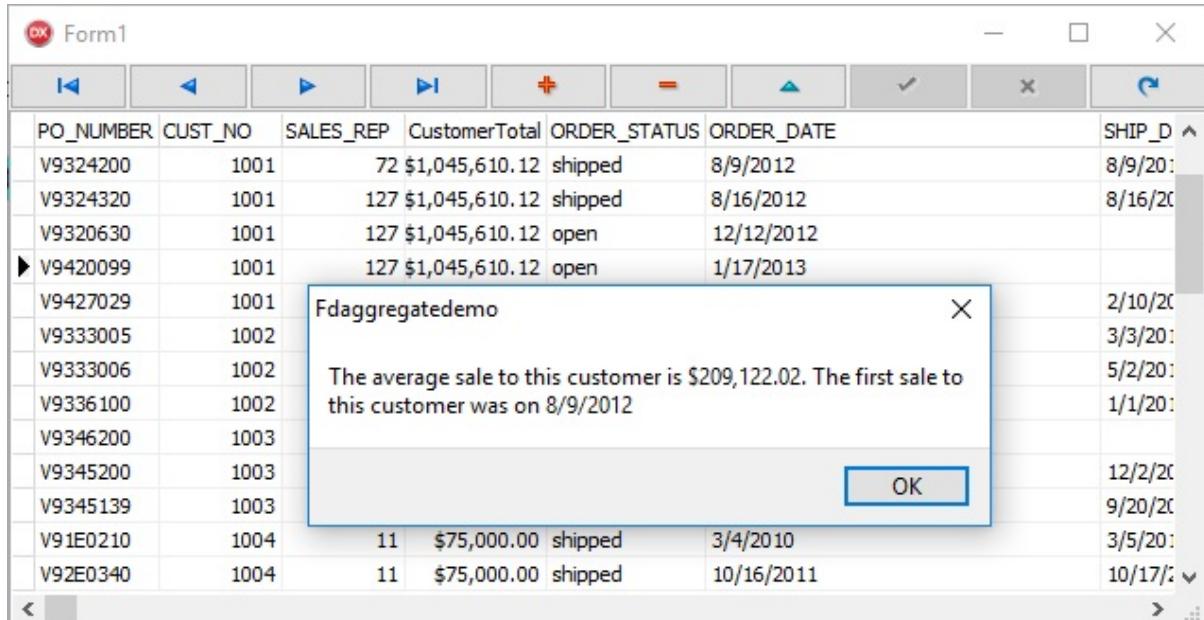
```
procedure TForm1.Aboutthiscustomer1Click(Sender: TObject);  
begin
```

```
ShowMessage('The average sale to this customer is ' +
```

```

Format( '%m',
[StrToFloat(SalesTable.Aggregates[0].Value)]) +
‘. The first sale to this customer was on ‘ +
VarToStr(SalesTable.Aggregates[1].Value));
end;

```



## 274 Delphi in Depth: FireDAC

If you now run this project, you can see the values calculated by the Aggregates collection items by right-clicking a record and selecting About this customer.

The displayed dialog box should look something like that shown in Figure 10-5.

**Figure 10-5: A project that is displaying data from an AggregateField and two Aggregates**

### Understanding Group State

Group state refers to the relative position of a given record within its group.

Using group state, you can discover whether the current record is the first record in its group (given the current index), the last record in its group, neither the last nor the first record in the group, or the only record in the group. You determine group state for the current record by calling the FireDAC dataset's

GetGroupState method. This method has the following syntax:

```
function GetGroupState(Level: Integer): TGroupPosInds;
```

When you call GetGroupState, you pass an integer indicating grouping level. Passing a value of 0 (zero) to GetGroupState will return information about the current record's relative position within the entire dataset. Passing a value of 1 will return the current record's group state with respect to the first field of the

Chapter 10: Virtual Fields 275

current index, passing a value of 2 will return the current record's group state with respect to the first two fields of the current index, and so on.

GetGroupState returns a set of TGroupPosInd flags. TGroupPosInd is declared as follows:

```
TGroupPosInd = (gbFirst, gbMiddle, gbLast);
```

As should be obvious, if the current record is the first record in the group, GetGroupState will return a set containing the gbFirst flag. If the record is the last record in the group, this set will contain gbLast. When GetGroupState is called for a record somewhere in the middle of a group, the gbMiddle flag is returned. Finally, if the current record is the only record in the group,

GetGroupState returns a set containing both the gbFirst and gbLast flags.

GetGroupState can be particularly useful for suppressing redundant information when displaying a dataset's data in a multi-record view, like that provided by the DBGrid component. For example, consider the preceding figure of the main

form, Figure 10-5. Notice that the CustomerTotal AggregateField value is displayed for each and every record, even though it is being calculated on a customer-by-customer basis. Not only is the redundant aggregate data unnecessary, it makes reading the data more difficult. This is a classic signal-to-noise ratio problem.

Using GetGroupState, you can test whether or not a particular record is the last record for the group, and if so, display the value for the CustomerTotal field.

For records that are not the last record in their group (based on the CustIdx index), you can suppress the display of the data. Displaying total amounts for a customer in the last record for that customer serves to emphasize the fact that the calculation is based on all of the customer's records.

Determining group state and suppressing or displaying the data can be achieved by adding an OnGetText event handler to the CustomerTotal AggregateField.

The following is the OnGetText event handler for the FDQuery1CustomerTotal AggregateField in the FDAggregatesAndGroupState project:

```
procedure TForm1.FDQuery1CustomerTotalGetText(Sender: TField;
var Text: String; DisplayText: Boolean);
begin
if gbLast in SalesTable.GetGroupState(1) then
  Text := Format('%.2m',[StrToFloat(Sender.Value)])
  276 Delphi in Depth: FireDAC
else
  Text := '';
end;
```

If you think about it, you might also want to suppress the CustNo field for all but the first records in the group. This event handler can be associated with the CustNo field at design time, but only if you have added persistent data fields for all of the fields that you want displayed in the DBGrid. If you want to use dynamic fields with your DBGrid, you will need to hook the OnGetText event handler to the dynamic CustNo field at runtime.

The following is the OnGetText event handler associated with the persistent CustomerTableCustNo data field in the FDAggregatesAndGroupState project:

```
procedure TForm1.CustNoGetText(Sender: TField;
var Text: String; DisplayText: Boolean);
begin
if gbFirst in SalesTable.GetGroupState(1) then
  Text := Sender.Value
else
  Text := '';
end;
```

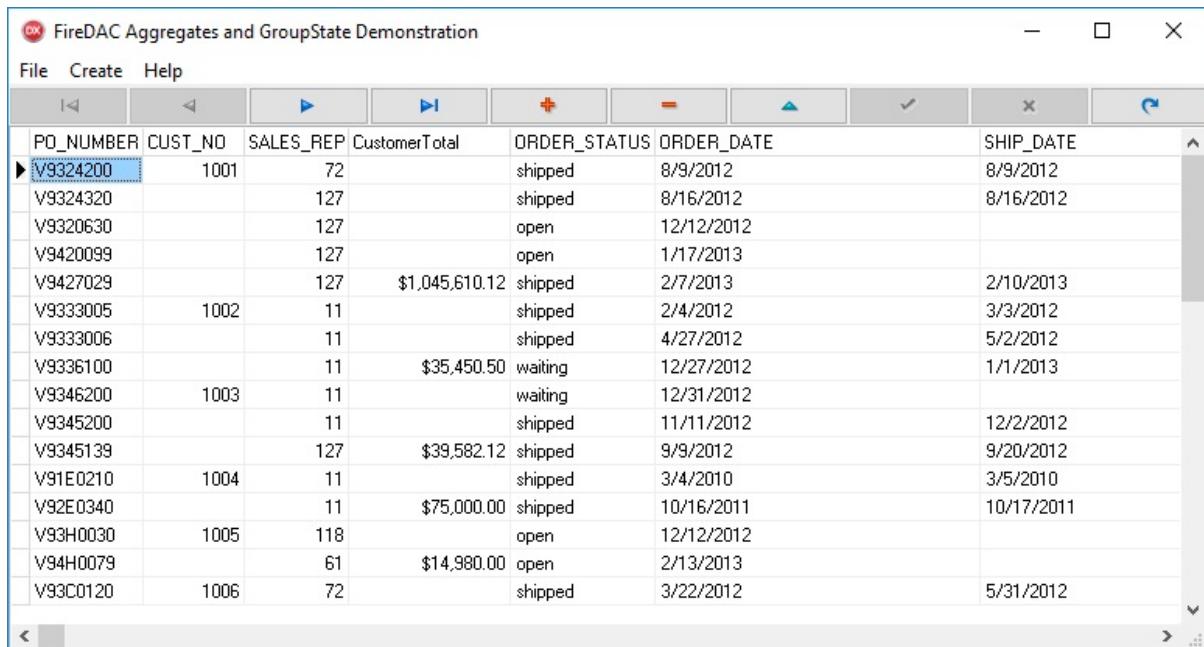
If you want to hook this event handler to a dynamic CustNo field at runtime, you can do this using something similar to the following from the OnCreate event handler of your form:

```
SalesTable.FieldByName('CUST_NO').OnGetText :=
```

CustNoGetText;

Figure 10-6 shows the main form of the running FDAGgregatesAndGroupState

project, which demonstrates the techniques described in this chapter. Notice that the CustNo and CustomerTotal fields are displayed only for the first and last records in each group, respectively.



The screenshot shows a Windows application window titled "FireDAC Aggregates and GroupState Demonstration". The menu bar includes "File", "Create", and "Help". Below the menu is a toolbar with various icons for navigation and editing. The main area is a grid-based data viewer with the following columns: PO\_NUMBER, CUST\_NO, SALES\_REP, CustomerTotal, ORDER\_STATUS, ORDER\_DATE, and SHIP\_DATE. The data consists of 18 rows, grouped by CUST\_NO. The first row (CUST\_NO 1001) has CustomerTotal displayed. The last row (CUST\_NO 1006) also has CustomerTotal displayed. Other rows show standard order details like shipped status and dates.

PO_NUMBER	CUST_NO	SALES_REP	CustomerTotal	ORDER_STATUS	ORDER_DATE	SHIP_DATE
V9324200	1001	72		shipped	8/9/2012	8/9/2012
V9324320		127		shipped	8/16/2012	8/16/2012
V9320630		127		open	12/12/2012	
V9420099		127		open	1/17/2013	
V9427029		127	\$1,045,610.12	shipped	2/7/2013	2/10/2013
V9333005	1002	11		shipped	2/4/2012	3/3/2012
V9333006		11		shipped	4/27/2012	5/2/2012
V9336100		11	\$35,450.50	waiting	12/27/2012	1/1/2013
V9346200	1003	11		waiting	12/31/2012	
V9345200		11		shipped	11/11/2012	12/2/2012
V9345139		127	\$39,582.12	shipped	9/9/2012	9/20/2012
V91E0210	1004	11		shipped	3/4/2010	3/5/2010
V92E0340		11	\$75,000.00	shipped	10/16/2011	10/17/2011
V93H0030	1005	118		open	12/12/2012	
V94H0079		61	\$14,980.00	open	2/13/2013	
V93C0120	1006	72		shipped	3/22/2012	5/31/2012

Chapter 10: Virtual Fields 277

**Figure 10-6: GroupState is used to display CustNo and CustomerTotal values for selected records within a group**

### Creating AggregateFields at Runtime

AggregateFields, because of their potential uses in analytics, are more likely than many other field types to be unknown in advance, and therefore, need to be created at runtime. For example, you may provide users with an interface where they can specify the types of statistics that they want to see for groups of their data. You can then create those AggregateFields on-the-fly.

In short, creating AggregateFields, as well as Aggregate collection items, at runtime requires the same type of configuration as does creating those objects at design time. The obvious difference is that you cannot use the design tools, such as the Fields Editor or the Object Inspector, to set the various properties at runtime.

The creation of a new AggregateField at runtime is demonstrated in the code associated with the FDAGgregatesAndGroupState project. This code can be

found on the OnClick event handler for the menu item labeled Create Aggregate. This event handler is shown in the following code segment:

```
procedure TForm1.NewAggregateField1Click(Sender: TObject);
```

```
begin
```

```
 278 Delphi in Depth: FireDAC
```

```
  SalesTable.DisableControls;
```

```
  SalesTable.Close;
```

```
  try
```

```
    with TAggregateField.Create(Self) do
```

```
      begin
```

```
        FieldName := 'NumberOfSales';
```

```
        Expression := 'COUNT(CUST_NO)';
```

```
        IndexName := 'CustIdx';
```

```
        GroupingLevel := 1;
```

```
        Active := True;
```

```
        Visible := True;
```

```
        Name := 'SalesTableNumberOfSales';
```

```
        DataSet := SalesTable;
```

```
      end;
```

```
      with DBGrid1.Columns.Add do
```

```
        begin
```

```
          FieldName := 'NumberOfSales';
```

```
          Title.Caption := 'Number of Sales';
```

```
          Index := 5;
```

```
        end;
```

```
    //Hook up the OnGetText event handler
```

```
    SalesTable.FieldByName('NumberOfSales').OnGetText :=
```

```
      FDQueryt1GetNumberSales;
```

```
    //Disable the menu to that creates the new AggregateField
```

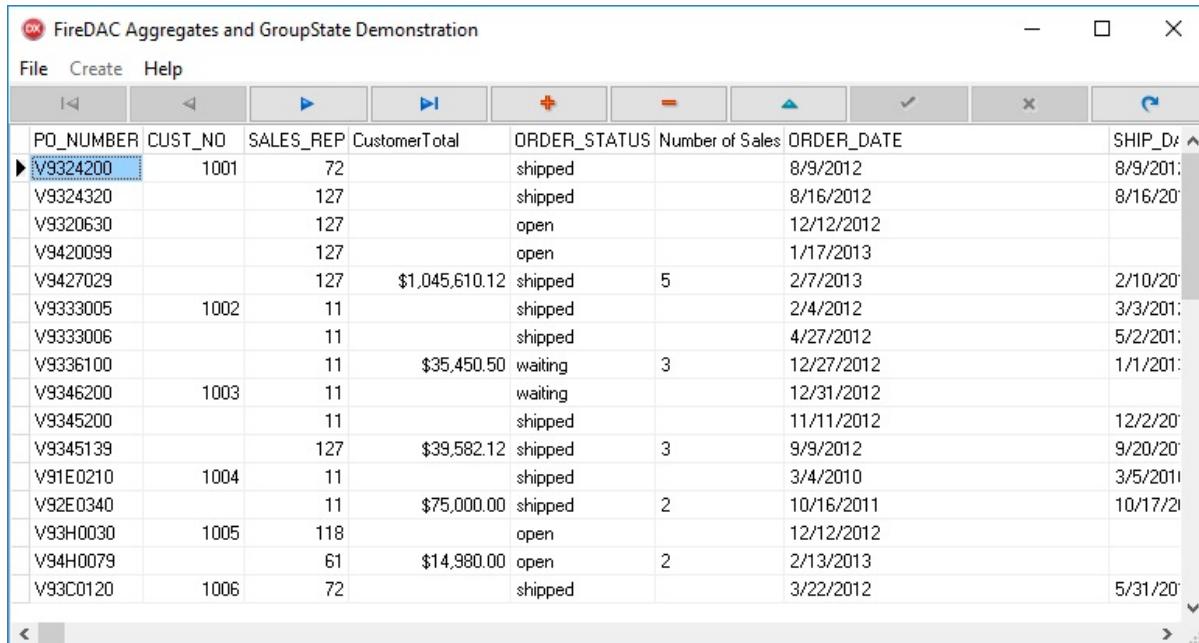
```

Create1.Enabled := False;
finally
  //Enable controls and re-open the FDQuery
  SalesTable.EnableControls;
  SalesTable.Open;
  //Hook up the CUSTNO OnGetText event handler
  SalesTable.FieldByName('CUST_NO').OnGetText :=
    SalesTableCustNoGetText;
end;
end;

```

As you can see from this code, the event handler begins by calling DisableControls on the FDQuery, which will prevent its subsequent closing from causing a flicker on the DBGrid. Next, the AggregateField is created and configured. Then, a new Column is added to the DBGrid for the display of this new virtual field. The new Column is put in position six of the DBGrid, which will cause it to be displayed to the right of the CustomerTotal field.

Next, the OnGetText event handler is set to the SalesTableGetNumberOfSales method. Like the SalesTableCustomerTotalGetText method used by the



PO_NUMBER	CUST_NO	SALES_REP	CustomerTotal	ORDER_STATUS	Number of Sales	ORDER_DATE	SHIP_DATE
V9324200	1001	72		shipped		8/9/2012	8/9/2012
V9324320		127		shipped		8/16/2012	8/16/2012
V9320630		127		open		12/12/2012	
V9420099		127		open		1/17/2013	
V9427029		127	\$1,045,610.12	shipped	5	2/7/2013	2/10/2013
V9333005	1002	11		shipped		2/4/2012	3/3/2012
V9333006		11		shipped		4/27/2012	5/2/2012
V9336100		11	\$35,450.50	waiting	3	12/27/2012	1/1/2013
V9346200	1003	11		waiting		12/31/2012	
V9345200		11		shipped		11/11/2012	12/2/2012
V9345139		127	\$39,582.12	shipped	3	9/9/2012	9/20/2012
V91E0210	1004	11		shipped		3/4/2010	3/5/2011
V92E0340		11	\$75,000.00	shipped	2	10/16/2011	10/17/2011
V93H0030	1005	118		open		12/12/2012	
V94H0079		61	\$14,980.00	open	2	2/13/2013	
V93C0120	1006	72		shipped		3/22/2012	5/31/2012

CustomerTotal field, this event handler displays the count of orders in the last record of the group. Unlike the SalesTableCustomerTotalGetText event handler, however, it does not format the number as a currency value.

The following is the code associated with the SalesTableGetNumberSales method:

```
procedure TForm1.FDQuery1GetNumberSales(Sender: TField;
var Text: String; DisplayText: Boolean);
begin
if gbLast in SalesTable.GetGroupState(1) then
Text := Sender.Value
else
Text := "";
end;
```

The menu item used to create this AggregateField is then disabled (since trying to create another persistent field using the same name would raise an exception).

Finally, the FDQuery is re-associated to its DataSource by calling EnableControls, and the FDQuery is re-opened.

If you run this project, and select Create | New AggregateField, the main form will look something like that shown in Figure 10-7.

### **Figure 10-7: An AggregateField has been added to the FDQuery and the DBGrid at runtime**

280 Delphi in Depth: FireDAC

### **Calculated Fields**

Calculated fields are virtual fields that display data produced by a calculation. That calculation is performed by code you attach to a dataset's OnCalcFields event handler.

Closely related is another virtual field, called InternalCalc. InternalCalc fields are particularly interesting because they actually store the values of the calculations, which a regular calculated field does not. Since the values of the InternalCalc calculation are stored, it is possible to create indexes that use those values. Indexes were discussed in *Chapter 7, Creating Indexes*.

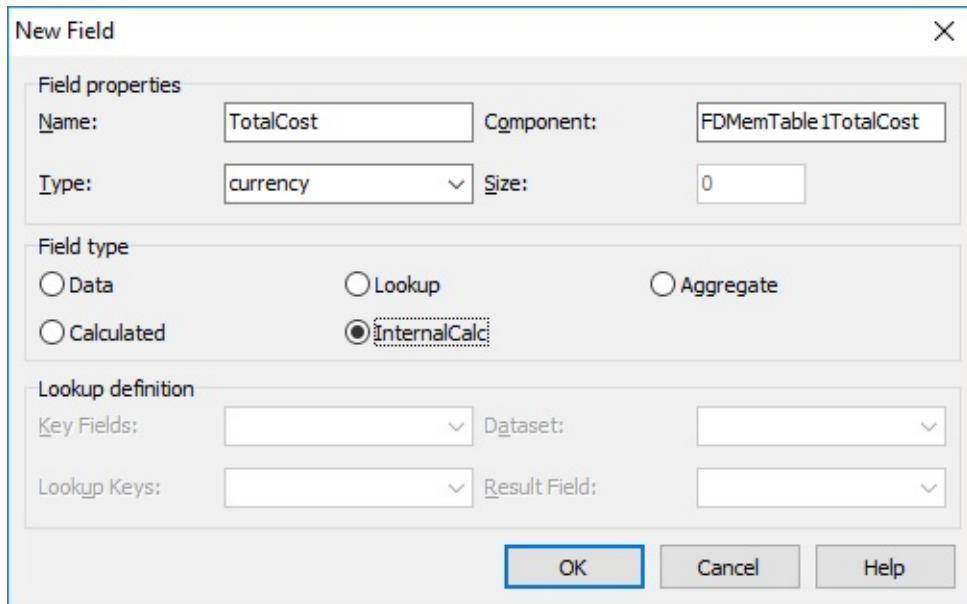
In almost every other way, InternalCalc and Calculated fields are identical.

InternalCalc fields just happen to be index-able, though the internal persistence does impose a slight performance overhead. If you need to create an index on a calculated field, use an InternalCalc field. Otherwise, use a Calculated field.

*Note: Not only is the data associated with InternalCalc fields stored in memory, if you persist the contents of a FireDAC dataset that includes one or more InternalCalc fields, that data is also persisted. Persisting and restoring data associated with FireDAC datasets is discussed in Chapter 11, Persisting Data .*

Both Calculated fields and InternalCalc fields are special in that you cannot assign data to them directly, either through data-aware controls or programmatically, with one exception discussed later. You can assign data to a Calculated or InternalCalc field only when the dataset to which the calculated field is attached is in the dsCalc state. And, datasets are in the dsCalc state only when they are executing their OnCalcFields event handlers.

Here is how it works. When a dataset needs data from a Calculated or an InternalCalc field, it executes its OnCalcFields event handler if one has been assigned. From that event handler, references to the dataset are to the current record only, and it is from this event handler that you can assign data to the Calculated and InternalCalc fields of the current record. If a dataset is being displayed in a grid like a DBGrid, the OnCalcFields event handler will trigger many times over a very short period of time, once for each record that needs to be displayed. As a result, it is important that your calculations be as efficient as possible. Otherwise, the overhead incurred by your calculations can have a negative effect on performance.



## Chapter 10: Virtual Fields 281

*Code: This FDCalcFields project is included the code download.*

An example of an InternalCalc field can be found in the FDCalcFields project.

Figure 10-8 shows how the New Field dialog box looked when this field was being defined.

**Figure 10-8: Adding the InternalCalc field named TotalCost to an FDMemTable**

An FDQuery is populated from a query at runtime. Here is the associated SQL

statement:

```
SELECT i.OrderNo, i.ItemNo, p.PartNo, i.Qty, p.ListPrice
```

```
FROM Items i
```

```
INNER JOIN Parts p ON i.PartNo = p.PartNo
```

The following is the OnCalcFields event handler found in the FDCalcFields project. In it, the value of the FDMemTable1Total field is assigned the product

The screenshot shows a Windows application window titled "FireDAC InternalCalc Field Demo". At the top, there is a toolbar with various icons for navigation and editing. Below the toolbar, a message "Click on column header to sort" is displayed. The main area contains a DBGrid showing a list of items. The columns are labeled: ORDERNO, ITEMNO, PARTNO, QTY, DISCOUNT, LISTPRICE, and Total. The "Total" column displays the calculated value for each row, which is the product of LISTPRICE and QTY minus the discount (calculated as LISTPRICE \* QTY \* (1 - DISCOUNT / 100)).

ORDERNO	ITEMNO	PARTNO	QTY	DISCOUNT	LISTPRICE	Total
1006	1	900	10	0	3999.95	\$39,999.50
1020	1	900	4	0	3999.95	\$15,999.80
1024	1	900	3	0	3999.95	\$11,999.85
1027	1	900	8	0	3999.95	\$31,999.60
1034	1	900	8	0	3999.95	\$31,999.60
1043	1	900	4	0	3999.95	\$15,999.80
1047	1	900	7	0	3999.95	\$27,999.65
1116	1	900	2	0	3999.95	\$7,999.90
1124	1	900	8	0	3999.95	\$31,999.60
1125	1	900	1	0	3999.95	\$3,999.95
1131	1	900	4	0	3999.95	\$15,999.80
1146	1	900	4	0	3999.95	\$15,999.80

## 282 Delphi in Depth: FireDAC

of the FDMemTable1ListPrice and FDMemTable1Qty field values, minus the item discount:

```
procedure TForm1.FDQuery1CalcFields(DataSet: TDataSet);
```

```
begin
```

```
FDQuery1Total.AsCurrency :=
```

```
FDQuery1LISTPRICE.AsCurrency * FDQuery1QTY.AsInteger -
(FDQuery1LISTPRICE.AsCurrency * FDQuery1QTY.AsInteger *
(FDQuery1DISCOUNT.AsInteger / 100));
```

```
//If there were additional InternalCalc or Calculated fields
```

```
//in this dataset, you would perform those calculations here.
```

```
end;
```

Figure 10-9 shows the running FDCalcFields main form. The Total column in the displayed DBGrid contains the results of the calculation performed in the preceding OnCalcFields event handler.

**Figure 10-9: An InternalCalc field calculates the total line item**

Chapter 10: Virtual Fields 283

*Note: Due to a bug in FireDAC at the time of this writing, the Total column will not be entirely sorted if you click its header, even though doing so will attempt to sort on an InternalCalc field. The bug is that FireDAC will not*

*execute OnCalcFields for every record unless we explicitly navigate to every record, which will cause the OnCalcFields event handler to execute for every record.*

*Only then will the sort operation properly sort based on the InternalCalc field whose value is defined from an OnCalcFields event handler execution.*

Ok, I know what you are thinking. I could have performed that calculation in the SQL executed by the query. Well, that's absolutely true, and in many cases, that would be a far more efficient way of doing things, but this is a demonstration of Calculated fields, so that's the route I took.

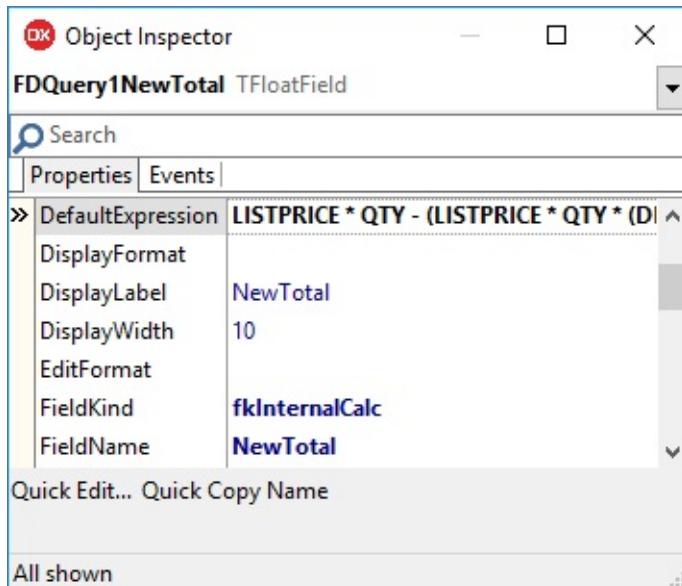
*Note: Do not attempt to navigate the dataset to which your OnCalcFields event handler is attached from within this event handler. You should use this event handler only to assign values to Calculated and InternalCalc fields of the current record.*

## **InternalCalc Fields and FireDAC Scalar Functions**

While Calculated and InternalCalc fields are available to all Delphi datasets, FireDAC adds an additional option which, unlike the traditional calculations, requires no runtime code. With FireDAC you can create an InternalCalc field and assign an expression to the DefaultExpression property. This expression can include literal values, field references, and FireDAC scalar functions. In other words, the expression can contain content similar to a Filter expression, except that it is not limited to Boolean expressions.

An updated version of the FDCalcFields project includes an additional InternalCalc field that calculates the total of a sale using the DefaultExpression property. This field is named NewTotal, and its DefaultExpression property is set to the following expression, which is the same calculation performed from the OnCalcFields event handler:

`LISTPRICE * QTY - (LISTPRICE * QTY * (DISCOUNT / 100))`



FireDAC InternalCalc Field Demo

Click on column header to sort

ORDERNO	ITEMNO	PARTNO	QTY	DISCOUNT	LISTPRICE	Total	NewTotal
1104	286	2630	1	0	18	\$18.00	\$18.00
1275	9	2657	1	0	18	\$18.00	\$18.00
1294	3	2657	1	0	18	\$18.00	\$18.00
1099	1	2619	1	0	19.95	\$19.95	\$19.95
1104	285	2619	1	0	19.95	\$19.95	\$19.95
1119	2	2619	1	0	19.95	\$19.95	\$19.95
1155	5	2619	1	0	19.95	\$19.95	\$19.95
1161	12	2619	1	0	19.95	\$19.95	\$19.95
1205	4	2619	1	0	19.95	\$19.95	\$19.95
1104	11	2350	1	0	29	\$29.00	\$29.00
1152	13	2350	1	0	29	\$29.00	\$29.00
1250	13	2350	1	0	29	\$29.00	\$29.00

## 284 Delphi in Depth: FireDAC

Figure 10-10 shows the now completed main form of the FDCalcFields project, where the right-most field is an InternalCalc field produced by an expression defined at design time. In this case, the NewTotal column has been clicked once, and the FDQuery has been properly sorted by the NewTotal values.

**Figure 10-10: The NewTotal field is an InternalCalc field whose calculation is defined by the DefaultExpression property**

Chapter 10: Virtual Fields 285

For a detailed discussion of the FireDAC scalar functions see *Chapter 14, The SQL Command Preprocessor*.

## **Lookup Fields**

A Lookup field is one that displays data from another, related table. For example, the Orders table in dbdemos.gdb database contains an employee number field, but not an employee name field. If you want to display the employee name in the Orders table, you can do so automatically using a Lookup field, without resorting to a join in your SQL query or additional code. Lookup fields permit this to be done entirely through properties, so long as your data has the correct relationships (in a relational database sense).

Several conditions must be met if you want to use a Lookup field. First, the table to which you want to add a Lookup field must include a *foreign key*. A foreign key is a field (or set of fields) that map to the primary key of another dataset. In the case of the Orders and Employee tables of the dbdemos.gdb database, this condition is met. Specifically, the EmpNo field in the Orders table is a foreign key, corresponding to the EmpNo field (the primary key) in the Employee table. In addition, you must have a dataset for the lookup table, a table that the Lookup field is configured to use for lookup, and in this example, that is the Employee table.

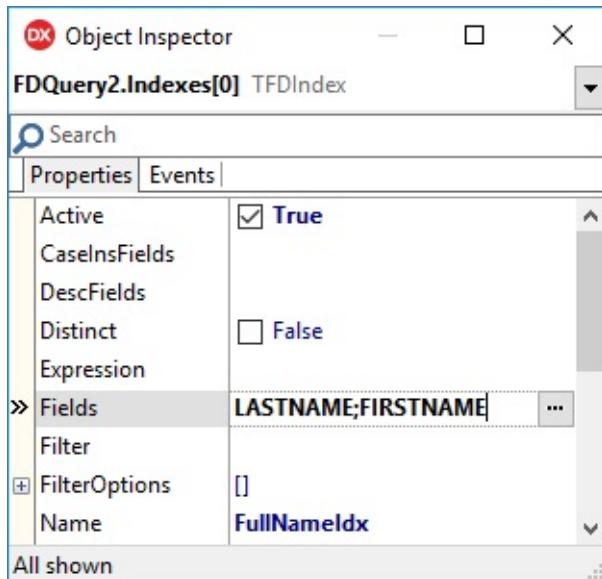
*Code: The FDLookupField project can be found in the code download.*

To be honest, Lookup fields are not used as extensively as they were in the early days of Delphi. Maybe this is because a number of third-party vendors ship components that do much of what lookup fields do but with more bells and

whistles than Delphi. Or, maybe it's just because Delphi developers have forgotten about them. Maybe once discovered again, Lookup fields will regain their popularity. Whatever the reason, they are worth another look.

In this section, I am going to refer to the FDLookupField project, and point out how the Lookup field was configured. I'll leave it to you to decide if it's something you might be interested in using.

To begin with, this project contains two queries, FDQuery1 and FDQuery2. The principle query selects records from the Orders table of the dbdemos.gdb



## 286 Delphi in Depth: FireDAC

database. The second table, which I refer to as the *lookup table*, selects data from the Employee table of this same database.

To begin with, FDQuery2, the lookup table, includes the following select query, which selects EMPNO, FIRSTNAME, and LASTNAME fields, as well as a

field that concatenates the FIRSTNAME and LASTNAME fields, separated by

a single space, from the Employee table:

```
SELECT EMPNO, FIRSTNAME, LASTNAME,  
{ CONCAT( { CONCAT( FIRSTNAME, ' ') } , LASTNAME) } AS  
FULLNAME  
FROM Employee
```

Next, FDQuery2 has an index, which orders the records by LASTNAME and then FIRSTNAME. This index is shown here in the Object Inspector. Finally, FDQuery2 is set to use this index.

Next, FDQuery1 includes the following SQL statement, which selects all records from the Orders table, including the employee number (EMPNO):

```
SELECT * FROM Orders;
```

Let's now turn our attention to defining the Lookup field. First, display the Fields Editor for FDQuery1 and add all fields by right-clicking and selecting Add All Fields from the displayed context menu, or by pressing Ctrl-F. We are now ready to add the Lookup field.

## Chapter 10: Virtual Fields 287

Add a new field by right-clicking in the Fields Editor and selecting New Field, or by pressing Ctrl-N. Delphi responds by displaying the New Field dialog box.

With the Lookup radio button selected in the Field Type panel, set Name to EmployeeName, Type to string and Size to 30.

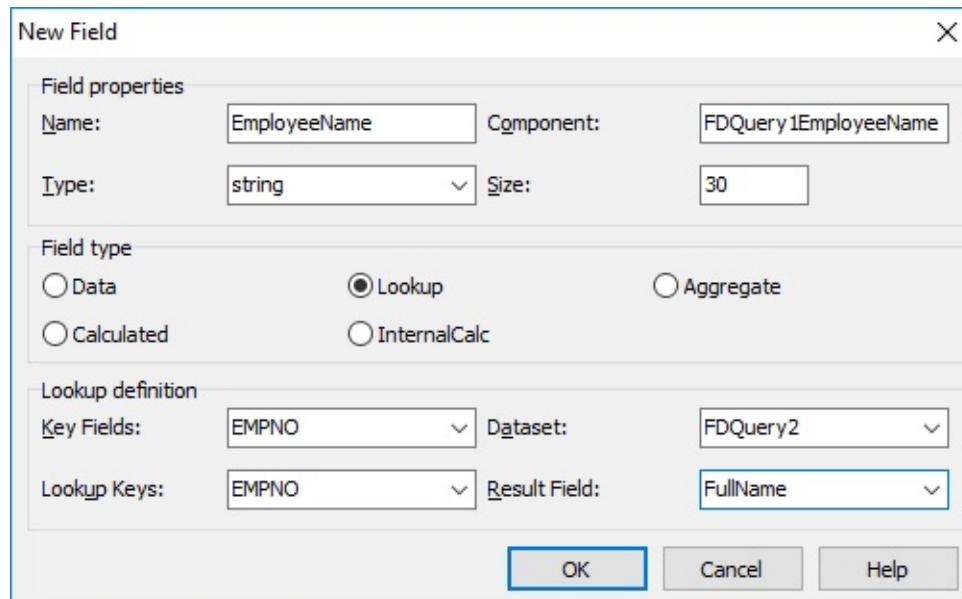
Because we selected the Lookup radio button, the Lookup definition fields become enabled. Set Key Fields to the field or fields that constitute the foreign key of the Orders table, which is the EMPNO field in this case.

Next, set DataSet to the lookup table, which is FDQuery2. In Lookup Keys, enter the name or names of the primary key fields from the lookup table. These primary key don't have to have the same names as the fields you enter in the Key Fields field, but you must have the same number of fields, and they must correspond in type to the fields in the Key Fields field. Since the EMPNO field in the Employee table is the primary key, set the Lookup Keys field to EMPNO.

In short, we are telling Delphi that the EMPNO field of FDQuery1 is associated with the EMPNO field of FDQuery2.

Finally, we set Result Field to the data we want to see when we are performing our lookup. In short, we will be displaying the employee full name to select, but it is the EMPNO field value that we are setting. Sound confusing? Well, once you see it in action you'll get it.

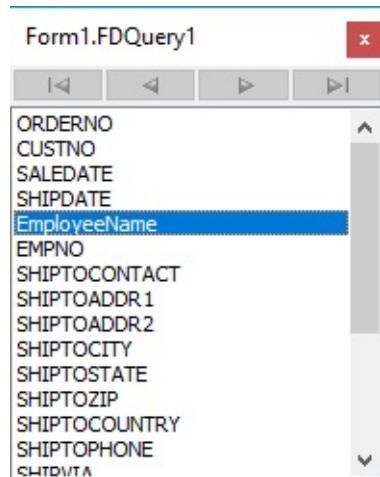
When the Lookup field has been configured, the New Field dialog box will look like that shown in Figure 10-11.



**Figure 10-11: The New Field dialog box with a Lookup field**

**configuration** Once we click OK, we can change the position of this new field in FDQuery1's Field Editor, in order to define where in the DBGrid this field should be shown.

In the following figure, EmployeeName has been moved to a position before (to the left of) the EMPNO field. Traditionally, we would actually remove the EMPNO field, since it's just a number, and we would let the user select the EMPNO by selecting the employee's full name. However we're keeping it here so you can see the Lookup field in action.



## Chapter 10: Virtual Fields 289

That's all we need to do — we don't even need to write code.

Figure 10-12 shows the main form with the Lookup field (EmployeeName) displayed next to the EMPNO field value. Importantly, the names that appear in the EmployeeName field corresponds to the employee number value that appears in the EMPNO field.

The screenshot shows a FireDAC Lookup Field interface. At the top, there's a toolbar with various navigation icons: back, forward, search, and other database-related functions. Below the toolbar is a grid of data with the following columns: ORDERNO, CUSTNO, SALEDATE, SHIPDATE, EmployeeName, EMPNO, and SHIPTOCD. The data rows represent sales transactions from 1988, with the EmployeeName column showing the name associated with each transaction. Row 1011 is currently selected, and a dropdown menu is open over the EmployeeName field, displaying the name "Kim Lambert". The EMPNO column shows the employee ID for each transaction.

ORDERNO	CUSTNO	SALEDATE	SHIPDATE	EmployeeName	EMPNO	SHIPTOCD
1005	1356	4/20/1988	1/21/1988 12:00:00 PM	Yuki Ichida	110	
1006	1380	11/6/1994	11/7/1988 12:00:00 PM	Walter Steadman	46	
1007	1384	5/1/1988	5/2/1988	Ashok Ramanathan	45	
1008	1510	5/3/1988	5/4/1988	Terri Lee	12	
1009	1513	5/11/1988	5/12/1988	Jennifer M. Burbank	71	
1010	1551	5/11/1988	5/12/1988	Roger Reeves	36	
1011	1560	5/18/1988	5/19/1988	Kim Lambert	5	
1012	1563	5/19/1988	5/20/1988	Takashi Yamamoto	118	
1013	1624	5/25/1988	5/26/1988	Jacques Glon	134	
1014	1645	5/25/1988	5/26/1988	John Montgomery	144	
1015	1651	5/25/1988	5/26/1988	Jennifer M. Burbank	71	
1016	1680	6/2/1988	6/3/1988	Sue Anne O'Brien	65	
1017	1984	6/12/1988	6/13/1988	Ann Bennet	28	
1018	2118	6/18/1988	6/19/1988	Takashi Yamamoto	118	
1019	2135	6/24/1988	6/25/1988	Bill Parker	114	
1020	2156	6/24/1988	6/25/1988	Luke Leung	61	
1021	2163	6/24/1988	6/25/1988	Carol Nordstrom	52	

## 290 Delphi in Depth: FireDAC

**Figure 10-12: A Lookup field is displaying the human readable value associated with the EMPNO field**

But it gets better. If we attempt to edit the EmployeeName field, a dropdown menu becomes available, and it displays the names of the employees, using the values found in the FDQuery2 dataset, in the sort order defined by the index on the FDQuery2 dataset. This is shown in Figure 10-13.

FireDAC Lookup Field

The screenshot shows a database grid titled "FireDAC Lookup Field". The grid has columns: ORDERNO, CUSTNO, SALEDATE, SHIPDATE, EmployeeName, EMPNO, and SHIPTOCD. Row 1011 is selected, and its EmployeeName cell displays "Kim Lambert". A dropdown menu is open over this cell, listing employee names: Yuki Ichida, Leslie Johnson, Scott Johnson, Kim Lambert, Terri Lee, Luke Leung, and Mary S. MacDonald. The name "Luke Leung" is highlighted with a blue selection bar. The EMPNO column for row 1011 shows the value 5.

ORDERNO	CUSTNO	SALEDATE	SHIPDATE	EmployeeName	EMPNO	SHIPTOCD
1005	1356	4/20/1988	1/21/1988 12:00:00 PM	Yuki Ichida	110	
1006	1380	11/6/1994	11/7/1988 12:00:00 PM	Walter Steadman	46	
1007	1384	5/1/1988	5/2/1988	Ashok Ramanathan	45	
1008	1510	5/3/1988	5/4/1988	Terri Lee	12	
1009	1513	5/11/1988	5/12/1988	Jennifer M. Burbank	71	
1010	1551	5/11/1988	5/12/1988	Roger Reeves	36	
1011	1560	5/18/1988	5/19/1988	Kim Lambert	5	
1012	1563	5/19/1988	5/20/1988	Yuki Ichida	118	
1013	1624	5/25/1988	5/26/1988	Leslie Johnson	134	
1014	1645	5/25/1988	5/26/1988	Scott Johnson	144	
1015	1651	5/25/1988	5/26/1988	Kim Lambert	71	
1016	1680	6/2/1988	6/3/1988	Terri Lee	65	
1017	1984	6/12/1988	6/13/1988	Luke Leung	52	
1018	2118	6/18/1988	6/19/1988	Mary S. MacDonald	28	
1019	2135	6/24/1988	6/25/1988	Takashi Yamamoto	118	
1020	2156	6/24/1988	6/25/1988	Bill Parker	114	
1021	2163	6/24/1988	6/25/1988	Luke Leung	61	
				Carol Nordstrom		

## Chapter 10: Virtual Fields 291

**Figure 10-13: The Lookup field enables a dropdown menu in the DBGrid,**

**displaying the full names associated with EMPNO field values**

Furthermore, if we select a name, say Luke Leung, the EMPNO field is automatically updated with his employee number, which is 61, as shown in Figure 10-14.

ORDERNO	CUSTNO	SALEDATE	SHIPDATE	EmployeeName	EMPNO	SHIPTOCD
1005	1356	4/20/1988	1/21/1988 12:00:00 PM	Yuki Ichida	110	
1006	1380	11/6/1994	11/7/1988 12:00:00 PM	Walter Steadman	46	
1007	1384	5/1/1988	5/2/1988	Ashok Ramanathan	45	
1008	1510	5/3/1988	5/4/1988	Terri Lee	12	
1009	1513	5/11/1988	5/12/1988	Jennifer M. Burbank	71	
1010	1551	5/11/1988	5/12/1988	Roger Reeves	36	
I	1011	1560	5/18/1988	5/19/1988	Luke Leung	61
	1012	1563	5/19/1988	5/20/1988	Takashi Yamamoto	118
	1013	1624	5/25/1988	5/26/1988	Jacques Glon	134
	1014	1645	5/25/1988	5/26/1988	John Montgomery	144
	1015	1651	5/25/1988	5/26/1988	Jennifer M. Burbank	71
	1016	1680	6/2/1988	6/3/1988	Sue Anne O'Brien	65
	1017	1984	6/12/1988	6/13/1988	Ann Bennet	28
	1018	2118	6/18/1988	6/19/1988	Takashi Yamamoto	118
	1019	2135	6/24/1988	6/25/1988	Bill Parker	114
	1020	2156	6/24/1988	6/25/1988	Luke Leung	61
	1021	2163	6/24/1988	6/25/1988	Carol Nordstrom	52

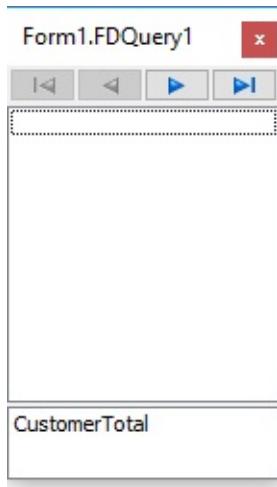
## 292 Delphi in Depth: FireDAC

**Figure 10-14: Selecting the lookup field value updates the lookup key field(s) in FDQuery1**

*Note: The AutoCalcFields property of FireDAC datasets determines when Calculated fields, InternalCalc fields, and Lookup fields are updated. When True, the default, these fields are updated when the dataset is opened, when it enters the edit mode, when additional records are read from the database, and when focus changes in data-aware controls on an edited dataset. When False, these fields are not updated when focus changes in data-aware controls on an edited dataset, but otherwise are updated the same as when AutoCalcFields is True.*

## Understanding FieldOptions

As I described at the outset of this chapter, until XE6, the creation of at least one persistent field prevented the creation of dynamic fields, with the one exception that I've noted. Specifically, you can create one or more Aggregate fields, and so long as no other type of persistent field exists for the dataset, dynamic fields are automatically created at runtime for each field in the dataset's result set.



## Chapter 10: Virtual Fields 293

This special nature of Aggregate fields can be seen in the Fields Editor. Here you can see the Fields Editor for an FDQuery that includes one Aggregate field.

Notice that the Aggregate persistent field appears in a special pane at the bottom of the Fields Editor.

That non-Aggregate persistent fields prevented the creation of dynamic fields posed a formidable obstacle for some developers. Specifically, those developers needing to create Calculated fields (or InternalCalc fields or Lookup fields) were also forced to create persistent data fields for all of the columns in the result set that they wanted to access. This limitation meant that developers would have to resort to complicated runtime code in order to surface these virtual fields for results sets whose structures are not predictable at design time.

*Note: I get a little technical in this section when discussing persistent fields defined using Fields or FieldDefs. I talk about these techniques in more detail in Chapter 12, Understanding FDMemTables. If you want more information on Fields and FieldDefs, you might want to quickly browse through that chapter to familiarize yourself with the associated issues.*

### The FieldOptions Property

Beginning with Delphi XE6, a new property, `FieldOptions`, was introduced in the `TDataSet` class. This property is of the type `TFieldOptions`, a class defined in the `Data.DB` unit. The `TFieldOptions` declaration is shown in the following code listing:

294 Delphi in Depth: FireDAC

`TFIELDOPTIONS = class(TPersistent)`

```
private
FDataSet: TDataSet;
FAutoCreateMode: TFieldsAutoCreationMode;
FPositionMode: TFieldsPositionMode;
FUpdatePersistent: Boolean;
procedure SetAutoCreateMode(const Value:
FieldsAutoCreationMode);
protected
function GetOwner: TPersistent; override;
public
constructor Create(DataSet: TDataSet);
procedure Assign(Source: TPersistent); override;
published
property AutoCreateMode: TFieldsAutoCreationMode
read AutoCreateMode
write SetAutoCreateMode default acExclusive;
property PositionMode: TFieldsPositionMode
read FPositionMode
write FPositionMode default poLast;
[Default(False)]
property UpdatePersistent: Boolean
read FUpdatePersistent
write UpdatePersistent default False;
end;
```

With the introduction of FieldOptions, it is the AutoCreateMode property of the FieldOptions property that controls how dynamic fields are created. As you can see in the preceding type declaration, this property is of type

TFieldsAutoCreationMode, an enumerated type. The TFieldsAutoCreationMode

type declaration is shown here:

```
TFIELDSAutoCreationMode =  
(acExclusive, acCombineComputed, acCombineAlways);
```

The default value for AutoCreateMode is acExclusive, which produces the dynamic field creation behavior that has been around since Delphi's release. In other words, the existence of any persistent field, other than Aggregate fields, prevents the automatic creation of dynamic fields. Consequently, the introduction of this new feature is backwards compatible with your existing applications.

## Chapter 10: Virtual Fields 295

If you set AutoCreateMode to acCombineComputed, dynamic fields are created, but only when the existing persistent fields consist only of Calculated fields and/or Aggregate fields (as well as InternalCalc). With this setting, the presence of persistent fields other than Calculated or Aggregate fields, such as a Data field or Lookup field, will prevent the creation of dynamic fields.

Importantly, this setting serves an invaluable role. In short, it permits you to define Calculated fields at either design time or runtime, and still benefit from dynamic field creation. Since most Calculated fields are defined at design time, the introduction of this feature answers the dreams of many Delphi database developers.

Finally, if you set AutoCreateMode to acCombineAlways, dynamic fields are always created, and these are added to the persistent fields for the underlying dataset. In addition, if a persistent Data field or Lookup field is defined for the dataset, and its name and type maps to a field in the underlying data, the corresponding dynamic field will not be created and the loaded data will be displayed using the persistent field. Importantly, that allows you to define properties for the persistent field, which in turn will affect how the underlying data that maps to that field will be displayed.

### Combine Options and PositionMode

If you set AutoCreateMode to either acCombineComputed or acCombineAlways, you use the PositionMode property of FieldOptions to control the position of the dynamically created fields in the underlying table structure. The default value for PositionMode is poLast, in which case, dynamic fields appear after the persistent fields in the table structure. By comparison, if you set Position to poFirst, the dynamic fields appear before the persistent fields.

The final PositionMode type, poFieldNo, only applies if you have created one

or more persistent Data fields. In these cases, those fields appear in their index order. This order is defined by the Index property of the individual TFields.

Data fields get their default Index property value based on their underlying field's position in the result set after taking into account the position of any non-Data persistent fields (field position is 0-based).

## The UpdatePersistent FieldOptions Property

When you create data fields at design time, you can also configure those fields at design time. In some cases, this permits you to affect the FieldDef property of the resulting field. This happens when the metadata of a dynamic field would

296 Delphi in Depth: FireDAC

have created FieldDef values that are different from the configured persistent field.

Under these conditions, you can control how the persistent field should initialize the resulting FieldDef property of the field at runtime. When you set UpdatePersistent to True, the Size, Precision, and Required properties of the resulting FieldDef will be based on metadata. When set to False, the configured properties of the persistent field take precedence.

## Field and Fields Properties

Now that fields in a single dataset can be a mixture of persistent and dynamic fields, it is natural that you might want to determine at runtime how a particular field was created. At the individual TField level, you can use the

TField.LifeCycle property to make this determination. LifeCycle is lcAutomatic if the field is dynamic, and lcPersistent if it is the result of a persistent field definition.

You can ask a similar question about the collection of TFields in a FireDAC dataset. Use the dataset's Fields property to examine the LifeCycles property.

TDataSet.Fields.LifeCycles will return a set of TFieldLifeCycle flags. If this set includes both the lcAutomatic and lcPersistent flags, at least one dynamic and one persistent field can be found in the dataset. If LifeCycles contains a single flag, all fields of that dataset are of the returned type.

In this next chapter, I discuss FireDAC dataset persistence.

# Chapter 11

## Persisting Data

All FireDAC datasets, as well as the FDSchemaAdapter, can save their data to a file or stream. At a later time, that data can be loaded into a compatible component, thereby restoring some or all of the state of the original object. For example, a FireDAC FDQuery can be edited and then saved to a file. At a later time, that data can be loaded into another FDQuery and edited again. If that FDQuery was in a cached updates mode at the time that the data was saved, any changes in the change cache will be present in the FDQuery that subsequently loads that data (assuming that you permit the change cache to be saved, which is the default). This feature permits you to maintain an editing session over an extended period of time after which the cached updates can be applied in a single call to `ApplyUpdates`, even though the application may have been closed and re-opened many times between the first edit and the call to `ApplyUpdates`.

This capability, being able to maintain a change cache across many application sessions, supports a feature often referred to as *the briefcase model* of data access. In the briefcase model, data is obtained from a database server and stored locally. From that point forward, the data is loaded from a file, and saved back to that file. At some later time, the application reconnects to the database after which `ApplyUpdates` is used to write those changes to the original tables.

Prior to the introduction of FireDAC into RAD Studio, the ClientDataSet was the only component in Delphi that provided you with this capability. (Cached updates is discussed in detail in *Chapter 16, Using Cached Updates*.)

Being able to persist and restore data permits you to implement features that go well beyond simply supporting the briefcase model. For example, the contents of an FDQuery can be streamed across the Internet, from one endpoint to

another, providing you with a convenient means of sharing data between applications.

Here's another example. Imagine that you have a complex accounting report that takes significant time to generate because it must first perform a large number of calculations on a closed accounting period. Since those calculations will not change, given that the accounting period is closed, the

printing of additional copies of that report in the future can be streamlined by saving the

## 298 Delphi in Depth: FireDAC

results of the calculations in an FDMemTable or an FDQuery, and then saving those results to a BLOB field of a database table. The first time the report is printed, the database is checked for the calculations. Since they are not found, the process of performing the calculations is initiated, after which, those calculations are stored in the database. Each subsequent attempt to print the report also checks the database, and finding that the calculations are already available, skips the calculation step and goes straight to printing.

This chapter begins with a look at saving data to, and loading data from, files. In this section, I explore the many properties and methods that you use, and the flexibility that they provide. I also describe how you can automate some of the operations associated with the saving and loading.

Towards the end of this chapter, I show you how to use streams to persist your datasets.

For the remainder of this chapter, I am going to speak about persisting data using FireDAC datasets. However, as I have already pointed out, this operation is also available for FDSchemaAdapters, which you need to use if you are

persisting FireDAC datasets that are participating in a centralized cached updates session. That having been said, in order to simplify the discussion of persistence-related operations, I am going to refer only to FireDAC datasets, with the exception of the very end of this chapter, where I discuss persistence as it relates to FDSchemaAdapter components.

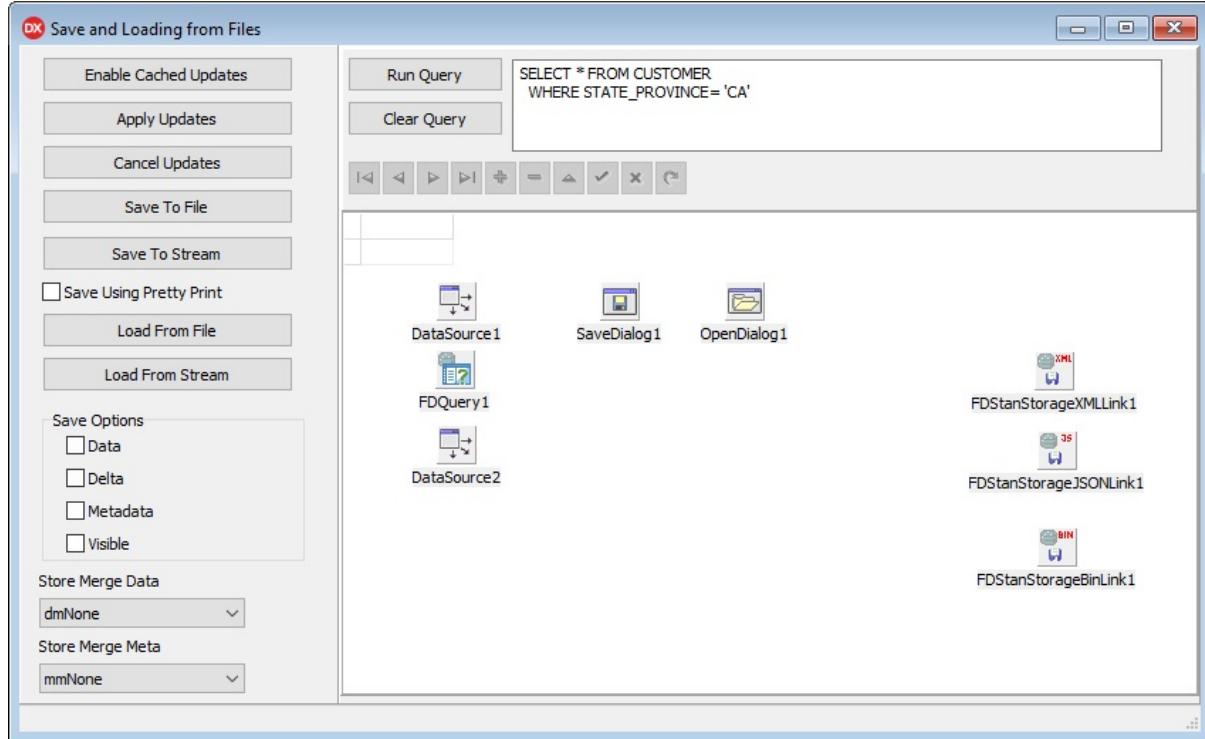
*Note: In my discussion of persistence, I am going to focus on the properties and methods of FireDAC datasets. However, I want to acknowledge that there are additional means for persisting data using FireDAC. These mechanisms make use of additional components, for example, FDBatchMoveTextWriter and*

*FDBatchMoveTextReader. These components provide you with a custom means of persisting data, and you are advised to take a look at them, but their coverage is outside the scope of this chapter.*

### Persisting Data to Files

There are two primary methods used to save data from a FireDAC dataset to a file. The most common, and the one that I will cover first, is the SaveToFile method. The second involves the PersistentFileName property of the dataset's

ResourceOptions property. I am going to cover PersistentFileName and its associated properties later in this section. However, SaveToFile also relies on



## Chapter 11: Persisting Data 299

some of the same properties as does PersistentFileName, so I will include a discussion of those properties where appropriate.

### Saving to Files

You call SaveToFile to save the data from your FireDAC dataset to a file. The SaveToFile method has the following signature, and is demonstrated in the FDSaveAndLoad project shown in Figure 11-1.

```
procedure SaveToFile(const AFileName: String = "";
AFormat: TFDStorageFormat = sfAuto);
```

### Figure 11-1: The main form of the FDSaveAndLoad project

*Code: The FDSaveAndLoad project can be found on the code download. See Appendix A for more information.*

SaveToFile has two optional parameters. The first parameter is the name of the file that you are saving. If you pass a simple file name, with no path information, the file will be saved in the directory listed in the ResourceOptions.DefaultStoreFolder property. (FireDAC datasets inherit this property from their FDConnection, and it cannot be overridden.) If the

## 300 Delphi in Depth: FireDAC

DefaultStoreFolder property is empty, and no path is provided in the AFileName parameter, the file will be saved in the same directory as the application's EXE file.

In addition to using either a relative path or an absolute path to identify your file location, you can include substitution variables with the format `$(name)` in either the filename or the path. There are several predefined variable names, and they are shown in Table 11-1.

### **Variable Name Description**

**DOC**

The documents folder path.

**TEMP**

The temporary files folder path.

**RUN**

The application executable path.

**RAND**

The random integer number.

**NEXT**

The next integer number in the specified path.

### **Table 11-1: Predefined variable names and their descriptions**

Environment variables can also be used. So long as the value of the environment variable defines a directory, or a portion of a directory path, you can use the following in a path definition:

`$(EnVar)`

where *EnVar* is the name of an environment variable.

As must be obvious from this discussion, you can use one or more substitution variable segments in your filename. For example, the following statement will write the data from FDQuery1 into a file named DATA *n*, where *n* is the next unused integer, with the extension XML, into the directory where the application's executable resides:

```
FDQuery1.SaveToFile('$(RUN)\DATA$(NEXT).XML');
```

You must supply the slashes in your filename string, but you can use either backslashes or forward slashes, and FireDAC will convert them, if necessary, based on the operating system.

## Chapter 11: Persisting Data 301

If you omit the extension of the filename, the extension defined in the `ResourceOptions.DefaultStoreExt` property is used (again, inherited from the `FDConnection`). If `DefaultStoreExt` is empty, a file with no extension is saved.

You can omit the filename parameter of `SaveToFile` altogether, so long as you have assigned a value to `ResourceOptions.PersistentFileName`. The rules that apply to the filename parameter are applied to `PersistentFileName`. It can include a path and an extension, and if either of these are omitted, the rules described above are applied.

An example of `SaveToFile` is found in the `OnClick` event handler of the button labeled `Save To File` in the `FDSaveAndLoad` project. This event handler is

shown here:

```
procedure TForm1.btnSaveToFileClick(Sender: TObject);  
var  
  FileExt: string;  
begin  
  if SaveDialog1.Execute then  
    begin  
      FileExt := ExtractFileExt(SaveDialog1.FileName).ToUpper;  
      if FileExt = ‘XML’ then  
        FDQuery1.SaveToFile( SaveDialog1.FileName,  
          TFDStorageFormat.sfXML )  
      else  
        if FileExt = ‘JSON’ then  
          FDQuery1.SaveToFile( SaveDialog1.FileName,  
            TFDStorageFormat.sfJSON )  
        else  
          if FileExt = ‘BIN’ then
```

```
FDQuery1.SaveToFile( SaveDialog1.FileName,  
TFDStorageFormat.sfBinary )  
else //everything else  
FDQuery1.SaveToFile( SaveDialog1.FileName,  
TFDStorageFormat.sfAuto )  
end;  
end;
```

## FILE FORMATS

The second optional parameter is the format of the saved file. FireDAC supports three file formats: XML (eXtensible Markup Language), JSON (JavaScript

Object Notation), and a binary format. You define the format explicitly by 302 Delphi in Depth: FireDAC

passing a TFDStorageFormat enumeration value in the second parameter. The declaration of TFDStorageFormat is shown here:

```
TFDStorageFormat = (sfAuto, sfXML, sfBinary, sfJSON);
```

If you omit the second parameter, FireDAC will use sfAuto, which will base the format on the file extension of the filename being written. If the extension is XML, the XML format will be used. If the extension is JSON, the JSON format will be used. If you use FDB, BIN, or DAT, the binary format will be used

(technically, with sfAuto, any format other than XML or JSON will use the binary format). If you use sfXML, sfJSON, or sfBinary, the corresponding format will be used, regardless of the file extension.

If the filename has no extension, the rules will be applied to the extension defined in ResourceOptions.DefaultStoreExt. If DefaultStoreExt is empty, the format specified in ResourceOptions.DefaultStoreFormat is used. When the second parameter is sfAuto, the filename has no extension, DefaultStoreExt is empty, and DefaultStoreFormat is sfAuto, the binary format is used.

In addition to specifying the file format to use, you must also ensure that you supply FireDAC with the resources it needs to support that format. This can be done by adding an instance of a corresponding FDStanStorage *format* Link component, where *format* is one of the following values: XML, JSON, or

BIN.

The use of the FDStanStorage *format* Link components is similar to the use of the FDGUIxWaitCursor component, in that the component itself is not important.

Instead, it is the act of adding this component to the form and then saving or compiling that results in the addition of an essential unit that provides FireDAC

with the resources it needs to support the corresponding format. With respect to format support, these corresponding units are named

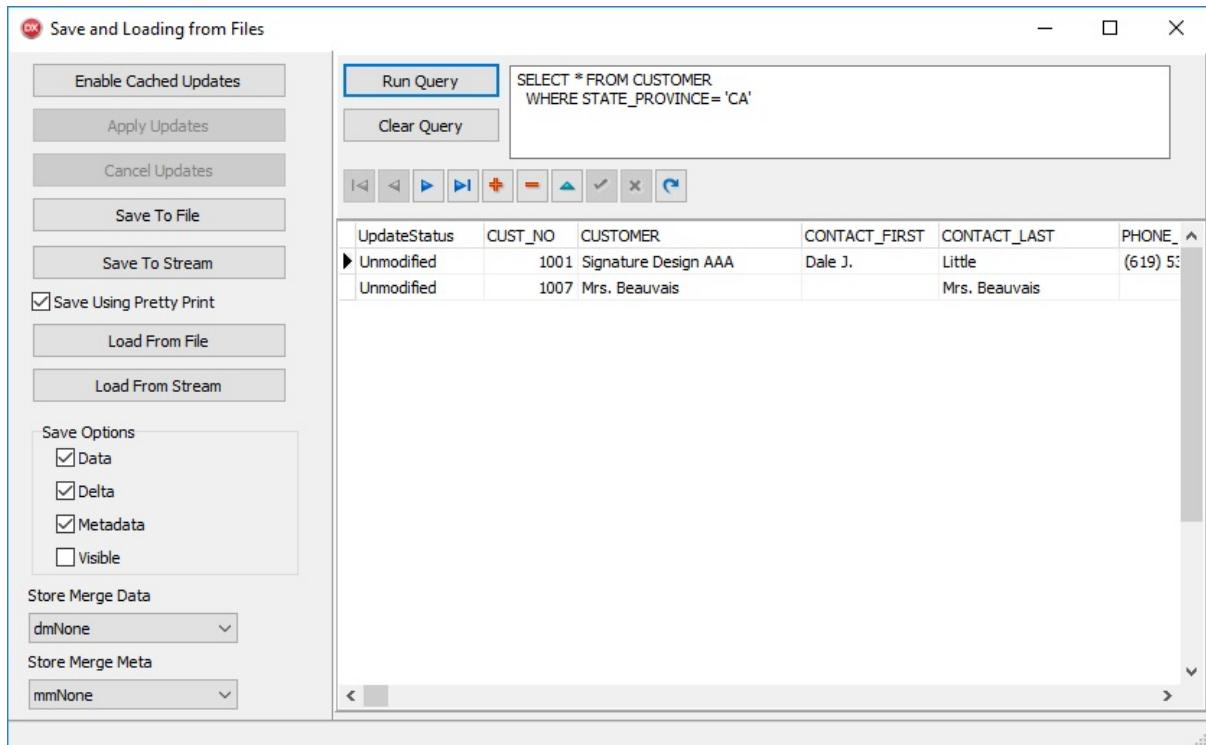
FireDAC.Stan.StorageXML, FireDAC.Stan.StorageJSON, and

FireDAC.Stan.StorageBin. Once these units appear in your uses clause, you may remove the corresponding component, though I like to leave them in place as a reminder to me or a future developer, that a requirement needed to be met.

While there are similarities between FDGUIxWaitCursor and the FDStanStorage *format* Link components, there is one important difference.

Beginning with Delphi XE8, you no longer need to add the FDGUIxWaitCursor

component to your project. By comparison, even with the most recent version of FireDAC, you must either add the FireDAC.Stan.Storage *format* unit manually or through the placement of the corresponding FDStanStorage *format* Link component.



## Chapter 11: Persisting Data 303

### STORING HUMAN-READABLE DATA

FireDAC can format both XML and JSON in a hierarchical structure that is easy to read, or it can store it as a string without line feeds, which will save a small amount of space in the resulting file.

You control whether or not formatting is applied using the

`ResourceOptions.StorePrettyPrint` Boolean property. Set it to True to format XML and JSON. When False, the XML or JSON is stored as one long string.

Figure 11-2 shows the FDSaveAndLoad project running, and shows that a query that returns just two records has been executed. When the Save Using Pretty Print checkbox is checked, the `StorePrettyPrint` property is set to True. If you now click the button labeled Save To File, and select a file name that uses the XML file extension, FireDAC will create the file shown in Figure 11-3.

**Figure 11-2: The FDSaveAndLoad project is ready to save or load data**

The screenshot shows a Windows Notepad window titled "SampleData.xml - Notepad". The window contains XML code representing a FireDAC Manager configuration. The XML is well-formatted with indentation, showing nested elements like <TableList>, <ColumnList>, and <RowList>. The code includes details about tables (FDQuery1), columns (CUST\_NO, CUSTOMER, CONTACT\_FIRST, etc.), rows, and constraints. The XML ends with a closing </FDBS> tag.

```
<?xml version="1.0" encoding="utf-8"?>
<FDBS Version="15">
  <Manager UpdatesRegistry="True">
    <TableList>
      <Table Name="FDQuery1" SourceName="CUSTOMER" SourceID="1">
        <ColumnList>
          <Column Name="CUST_NO" SourceName="CUST_NO" SourceID='1' />
          <Column Name="CUSTOMER" SourceName="CUSTOMER" SourceID='2' />
          <Column Name="CONTACT_FIRST" SourceName="CONTACT_FIRST" SourceID='3' />
          <Column Name="CONTACT_LAST" SourceName="CONTACT_LAST" SourceID='4' />
          <Column Name="PHONE_NO" SourceName="PHONE_NO" SourceID='5' />
          <Column Name="ADDRESS_LINE1" SourceName="ADDRESS_LINE1" SourceID='6' />
          <Column Name="ADDRESS_LINE2" SourceName="ADDRESS_LINE2" SourceID='7' />
          <Column Name="CITY" SourceName="CITY" SourceID="8" DataSize="50" />
          <Column Name="STATE_PROVINCE" SourceName="STATE_PROVINCE" SourceID="9" DataSize="50" />
          <Column Name="COUNTRY" SourceName="COUNTRY" SourceID="10" DataSize="50" />
          <Column Name="POSTAL_CODE" SourceName="POSTAL_CODE" SourceID="11" DataSize="10" />
          <Column Name="ON_HOLD" SourceName="ON_HOLD" SourceID="12" />
        </ColumnList>
        <ConstraintList/>
        <ViewList/>
        <RowList>
          <Row RowID="0">
            <Original CUST_NO="1001" CUSTOMER="Signature Design'" />
          </Row>
          <Row RowID="1">
            <Original CUST_NO="1007" CUSTOMER="Mrs. Beauvais" />
          </Row>
        </RowList>
      </Table>
    </TableList>
    <RelationList/>
    <UpdatesJournal>
      <Changes/>
    </UpdatesJournal>
  </Manager>
</FDBS>
```

304 Delphi in Depth: FireDAC

**Figure 11-3: The StorePrettyPrint property has been used to format this XML. The data has been clipped intentionally**

Chapter 11: Persisting Data 305

## WHAT TO PERSIST

Under normal circumstances, you will want to persist all of the information associated with the FireDAC dataset that you are saving. This includes the data itself, as well as the metadata. The information is necessary in order to load this information back into a corresponding FireDAC dataset.

In addition, if your FireDAC dataset is in cached updates mode at the time you are saving, you will need to save that information in order to continue your editing session when the data is once again loaded.

What information is persisted when you call SaveToFile (as well as SaveToStream) is controlled by the ResourceOptions.StoreItems property. This property consists of a set of one or more TFDStoreItem flags. The declaration of the TFDStoreItem enumeration is shown here:

```
TFDStoreItem = (siMeta, siData, siDelta, siVisible)
```

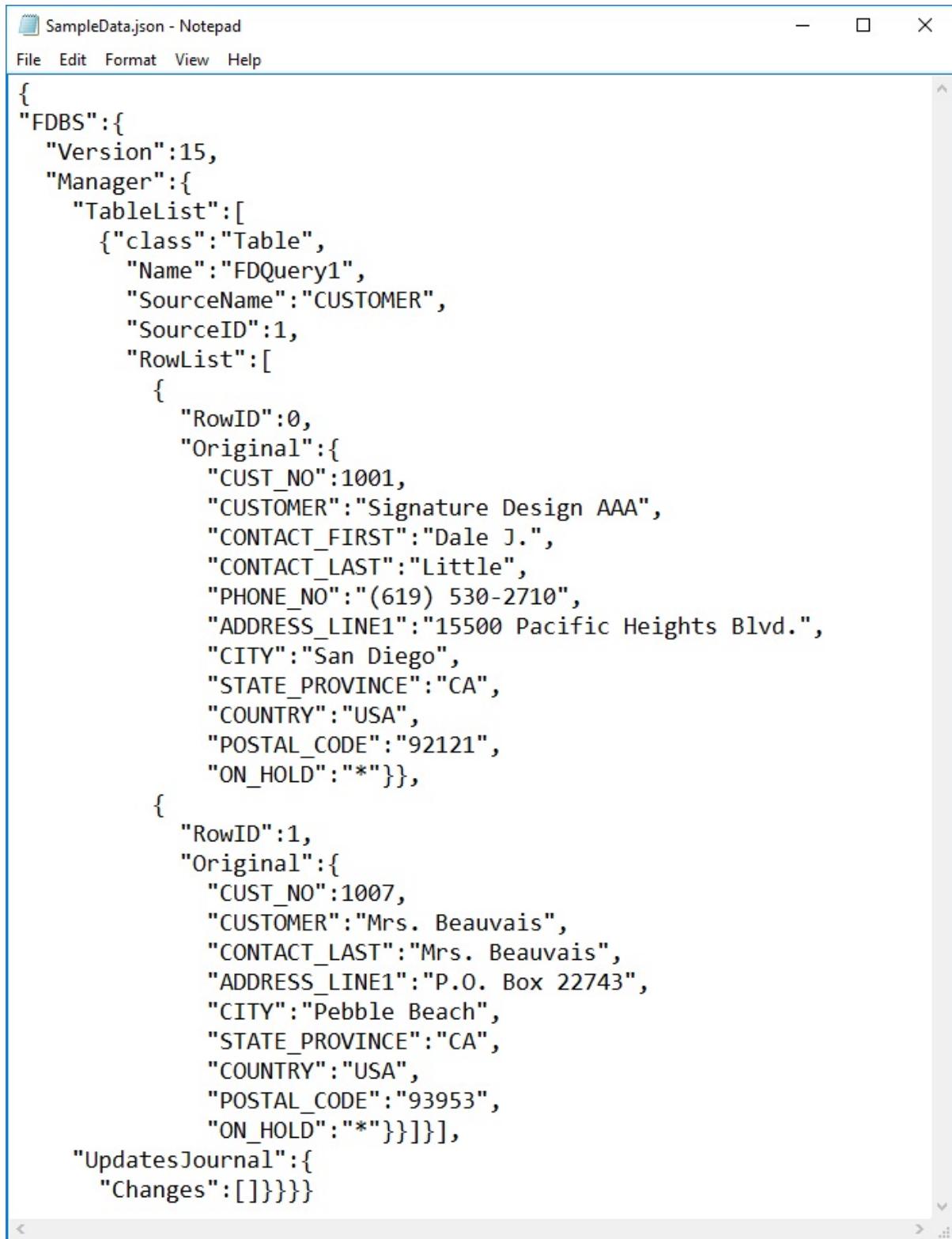
The default value of StoreItems is [siData, siDelta, siMeta]. This instructs FireDAC to store the data, the metadata, and the change cache. If you are not caching updates, you can omit the siDelta flag.

Omitting either siMeta or siData is done only for special purposes. For example, if you are creating JSON or XML that will be consumed by some non-Delphi

endpoint, you might omit the siMeta flag, since FireDAC's metadata is not going to be compatible with any other framework (though it could conceivably be useful information, if the endpoint is designed to process that information).

Omitting siMeta can also be useful when you want to transmit the data to another FireDAC dataset across a network, and that dataset already has metadata that matches the data being transmitted. In that case, omitting the siMeta flag will reduce the total amount of data that needs to be transmitted.

An example of JSON output where the siMeta flag has been omitted is shown in Figure 11-4. This output is created for the same query whose XML output was shown in Figure 11-3.



The screenshot shows a Windows Notepad window titled "SampleData.json - Notepad". The window contains a JSON object representing FireDAC data. The structure includes an "FDBS" key pointing to a "Version" of 15 and a "Manager" object. The "Manager" object contains a "TableList" array with two entries. Each entry has a "RowID" and an "Original" object containing various customer-related fields like CUST\_NO, CUSTOMER, CONTACT\_FIRST, CONTACT\_LAST, PHONE\_NO, ADDRESS\_LINE1, CITY, STATE\_PROVINCE, COUNTRY, POSTAL\_CODE, and ON\_HOLD. The "UpdatesJournal" key points to an array of changes, which is currently empty. The JSON is formatted with indentation and line breaks.

```
{
  "FDBS": {
    "Version": 15,
    "Manager": {
      "TableList": [
        {"class": "Table",
         "Name": "FDQuery1",
         "SourceName": "CUSTOMER",
         "SourceID": 1,
         "RowList": [
           {
             "RowID": 0,
             "Original": {
               "CUST_NO": 1001,
               "CUSTOMER": "Signature Design AAA",
               "CONTACT_FIRST": "Dale J.",
               "CONTACT_LAST": "Little",
               "PHONE_NO": "(619) 530-2710",
               "ADDRESS_LINE1": "15500 Pacific Heights Blvd.",
               "CITY": "San Diego",
               "STATE_PROVINCE": "CA",
               "COUNTRY": "USA",
               "POSTAL_CODE": "92121",
               "ON_HOLD": "*"
             }
           },
           {
             "RowID": 1,
             "Original": {
               "CUST_NO": 1007,
               "CUSTOMER": "Mrs. Beauvais",
               "CONTACT_LAST": "Mrs. Beauvais",
               "ADDRESS_LINE1": "P.O. Box 22743",
               "CITY": "Pebble Beach",
               "STATE_PROVINCE": "CA",
               "COUNTRY": "USA",
               "POSTAL_CODE": "93953",
               "ON_HOLD": "*"
             }
           }
         ],
         "UpdatesJournal": [
           {"Changes": []}
         ]
       }
     }
   }
}
```

## 306 Delphi in Depth: FireDAC

**Figure 11-4: JSON output with the metadata omitted**

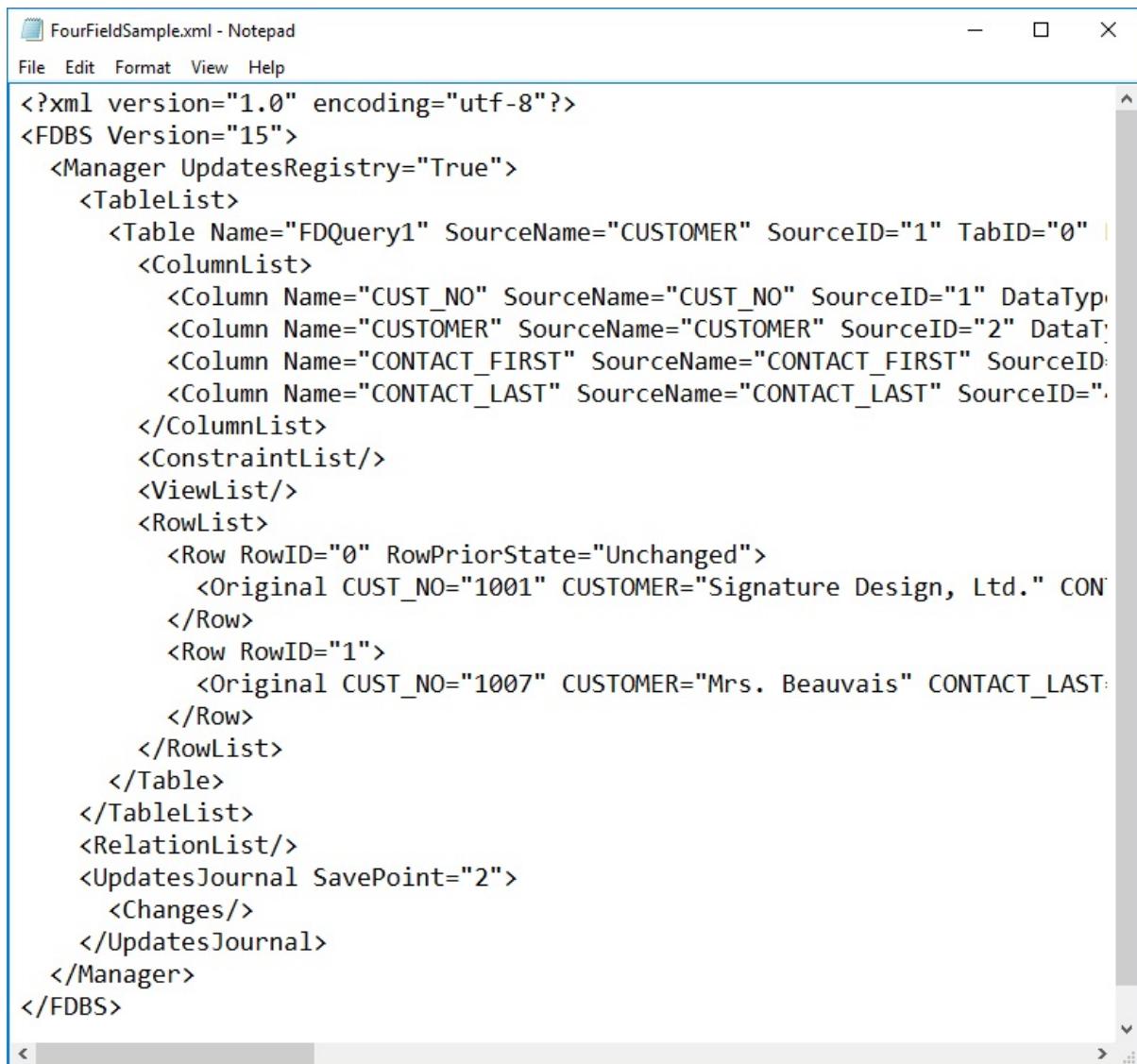
Similarly, you might omit the siData flag if you want to save just the structure to be loaded at a later time. This might be done in cases where the data is being saved in a traditional fashion, for example, to a database, but the

structure was created dynamically at runtime and your code needs to re-create an empty

FireDAC dataset with that same structure at a later time.

As mentioned previously, you can include the change cache by including the siDelta flag in StoreItems. Consider the following query:

```
SELECT CUST_NO, CUSTOMER, CONTACT_FIRST, CONTACT_LAST  
FROM CUSTOMER  
WHERE STATE_PROVINCE= 'CA'
```



The screenshot shows a Windows Notepad window titled "FourFieldSample.xml - Notepad". The window contains XML code representing a FireDAC dataset. The XML structure includes a Manager node with an UpdatesRegistry attribute set to "True". Inside the Manager node, there is a TableList node containing a single Table node named "FDQuery1". This Table node has a SourceName attribute set to "CUSTOMER" and SourceID attributes for CUST\_NO (1), CUSTOMER (2), CONTACT\_FIRST (3), and CONTACT\_LAST (4). The Table node also contains a ColumnList node with four Column nodes corresponding to these fields. Below the Table node is a RowList node containing two Row nodes. The first Row node (RowID=0) has an Original node with CUST\_NO="1001", CUSTOMER="Signature Design, Ltd.", CONTACT\_FIRST="John", and CONTACT\_LAST="Doe". The second Row node (RowID=1) has an Original node with CUST\_NO="1007", CUSTOMER="Mrs. Beauvais", CONTACT\_FIRST="Jane", and CONTACT\_LAST="Doe". The XML code is well-formatted with indentation and line breaks.

```
<?xml version="1.0" encoding="utf-8"?>  
<FDBS Version="15">  
  <Manager UpdatesRegistry="True">  
    <TableList>  
      <Table Name="FDQuery1" SourceName="CUSTOMER" SourceID="1" TabID="0" |  
        <ColumnList>  
          <Column Name="CUST_NO" SourceName="CUST_NO" SourceID="1" DataType="String" |  
          <Column Name="CUSTOMER" SourceName="CUSTOMER" SourceID="2" DataType="String" |  
          <Column Name="CONTACT_FIRST" SourceName="CONTACT_FIRST" SourceID="3" DataType="String" |  
          <Column Name="CONTACT_LAST" SourceName="CONTACT_LAST" SourceID="4" DataType="String" |  
        </ColumnList>  
        <ConstraintList/>  
        <ViewList/>  
        <RowList>  
          <Row RowID="0" RowPriorState="Unchanged">  
            <Original CUST_NO="1001" CUSTOMER="Signature Design, Ltd." CONTACT_FIRST="John" CONTACT_LAST="Doe" />  
          </Row>  
          <Row RowID="1">  
            <Original CUST_NO="1007" CUSTOMER="Mrs. Beauvais" CONTACT_FIRST="Jane" CONTACT_LAST="Doe" />  
          </Row>  
        </RowList>  
      </Table>  
    </TableList>  
    <RelationList/>  
    <UpdatesJournal SavePoint="2">  
      <Changes/>  
    </UpdatesJournal>  
  </Manager>  
</FDBS>
```

## Chapter 11: Persisting Data 307

Figure 11-4 show a portion of the pretty printed XML that was generated when the record for customer 1001 has had the Customer field changed from 'Signature Design' to 'Signature Design, Ltd.'. In this figure, the change is

seen in the Original and Current elements of RowID="0", and can be used to restore this revision if loaded into a FireDAC dataset.

**Figure 11-5: FireDAC saves the change cache information when siMeta is included in the StoreItems property. This image has been clipped intentionally**

You use the siVisible flag to control the persistence of visible records of the FireDAC dataset. When the siVisible flag is omitted, all records contained in the dataset are persisted.

## 308 Delphi in Depth: FireDAC

In most situations, this flag doesn't make any difference, since you want to persist all records contained in the dataset. On the other hand, if your dataset is in the cached updates mode and you have changed the default value of FilterChanges, siVisible can have a significant impact on what gets persisted. Here is an example of a situation where siVisible effects what gets saved.

Imagine an FDQuery that is in the cached updates mode, and you have deleted from it some records, and you have modified the FilterChanges property to

include only the rtDeleted flag. The result is that only deleted records will be visible (accessible programmatically or shown in a grid) in the dataset, even though other unmodified, inserted, and modified records may be present. In this situation, including the siVisible flag in StoreItems will result in only the deleted records being persisted. By comparison, if omitted, all records, including deleted, will be persisted.

*Note: The FilterChanges property, and the displaying of deleted records, is discussed in detail in Chapter 16, Using Cached Updates.*

## PERSISTED VERSION INFORMATION

Over time, the format used to persist data by FireDAC has been modified and improved. As a result, when FireDAC is loading previously persisted data, it needs to know which version of the persistence format was used when the content was created. As a result, when SaveToFile or SaveToStream is called, FireDAC writes version information into the persisted data. FireDAC can then use this information when it subsequently loads that data.

For an application that saves data and then later loads it, version information can be useful when you have upgraded the application to a newer version of Delphi containing a newer version of FireDAC, and it attempts to load a file

persisted with an earlier version of your application.

Another situation where version information is valuable is when you have a multi-tier application, where an application server shares persisted datasets with client applications over the internet. You might upgrade the application server, but you may have little control over when, or even if, the client applications get upgraded. Version information can help these applications manage

inconsistencies in the content of the data being passed back and forth.

Version information is controlled by the `ResourceOptions.StoreVersion` property, which contains an integer value. The default is -1, which instructs FireDAC to use the value of the `C_FD_StorageVer` constant in the

## Chapter 11: Persisting Data 309

`FireDAC.Stan.Consts` unit. Here is the declaration of `C_FD_StorageVer` from Delphi 10.1 Berlin and Delphi 10.2 Tokyo:

`C_FD_StorageVer = 15;`

It is recommended that you leave `StoreVersion` set to -1.

## Loading from Files

So long as a FireDAC dataset has been used to create a file, and the `siMeta` flag was present in the `ResourceOptions.StoreItems` property at the time that the file was saved, you can use a FireDAC dataset's `LoadFromFile` method to make that dataset active based on the structure found in the file. If the `siData` flag was also present at the time of saving, the saved data is also loaded. Finally, if the `StoreItems` contained the `siDelta` flag when the file was saved, the change cache is restored.

*Note: If the saved file included cached updates, and `StoreItems` includes the `siData` flag, the loaded FireDAC dataset's `UpdatesPending` property will be `True`. However, loading cached updates using `LoadFromFile` (or `LoadFromStream`) will not change the `CachedUpdates` property of that dataset from `False` to `True`. If you load a FireDAC dataset from a file that may include cached updates, and `siDelta` is in `StoreItems`, you should test the `UpdatesPending` property following the call to `LoadFromFile`, and set `CachedUpdates` to `True` if `UpdatesPending` is `True`. Otherwise, you will not be able to edit the change cache or apply those updates.*

You reload a properly saved FireDAC dataset by calling the `LoadFromFile`

method, and its signature is shown here:

```
procedure LoadFromFile(const AFileName: String = ";  
AFormat: TFDStorageFormat = sfAuto);
```

The use of LoadFromFile has many parallels to the SaveToFile method. It supports two optional parameters, one identifying the file to load and the other specifying the expected file format.

Like with SaveToFile, the first parameter, when present, specifies the file to load. This parameter is a string that can contain a file name, and can include a

### 310 Delphi in Depth: FireDAC

relative path, an absolute path, as well as use variable substitution using the format `$( name)`, where name can either be one of the predefined variables listed in Table 11-1 or an environment variable.

Other similarities to SaveToFile include the use of

ResourceOptions.DefaultStoreFolder and ResourceOptions.DefaultStoreExt when the path or file extension parts of the filename are omitted, as well as the use of the ResourceOptions.PersistentFileName property when the filename is omitted entirely.

LoadFromFile is demonstrated in the OnClick event handler of the button labeled Load From File, shown here. Also shown is the UpdateButtons custom method, which enables or disables buttons on the main form based on the active state of the form's FDQuery, as well as the value of the FDQuery's UpdatesPending property. (The UpdateButtons method is also called from the DataSource's OnUpdateData property, in order to keep the buttons properly synchronized with the state of the FDQuery on the form.)

```
procedure TForm1.btnExitClick(Sender: TObject);
```

```
var
```

```
  FileExt: string;
```

```
  i: Integer;
```

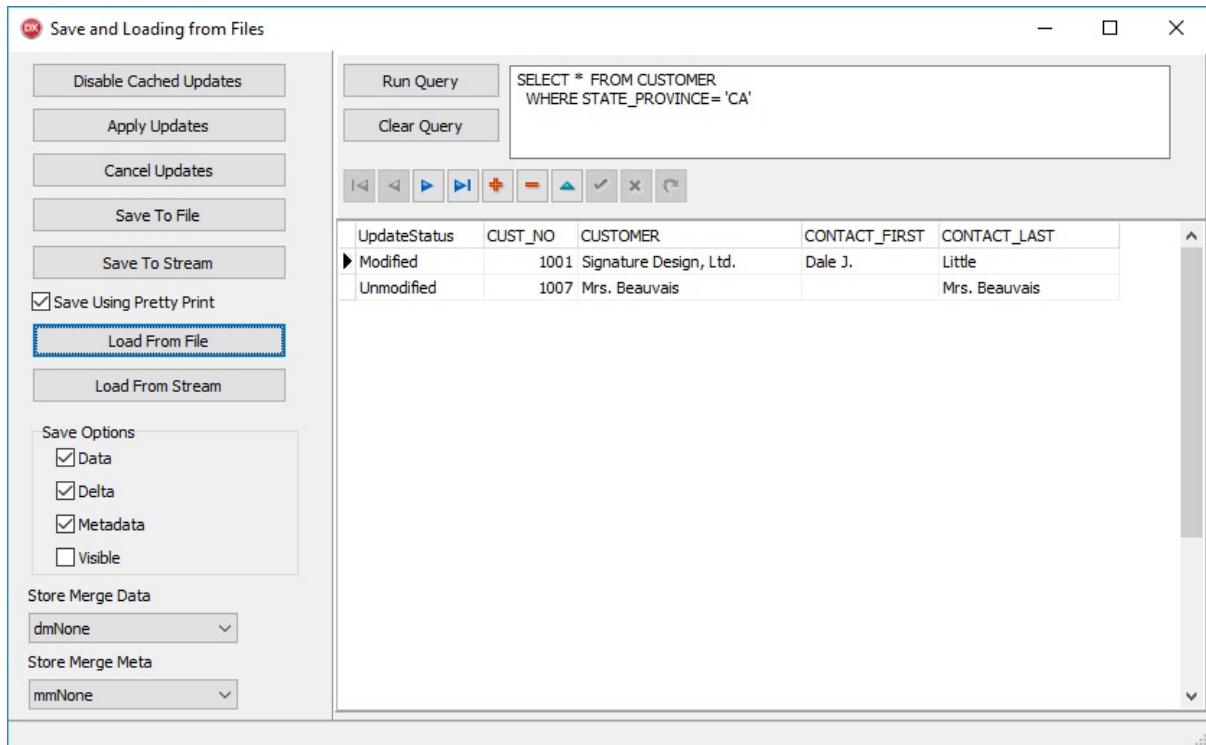
```
begin
```

```
  if OpenDialog1.Execute then
```

```
    begin
```

```
      FileExt := ExtractFileExt(OpenDialog1.FileName).ToUpper;
```

```
if FileExt = ‘XML’ then
  FDQuery1.LoadFromFile( OpenDialog1.FileName,
  TFDStorageFormat.sfXML )
else
  if FileExt = ‘JSON’ then
    FDQuery1.LoadFromFile( OpenDialog1.FileName,
    TFDStorageFormat.sfJSON )
  else
    if FileExt = ‘BIN’ then
      FDQuery1.LoadFromFile( OpenDialog1.FileName,
      TFDStorageFormat.sfBinary )
    else //everything else
      FDQuery1.LoadFromFile( OpenDialog1.FileName,
      TFDStorageFormat.sfAuto )
  end;
if not FDQuery1.CachedUpdates and FDQuery1.UpdatesPending then
  FDQuery1.CachedUpdates := True;
  UpdateButtons( FDQuery1.Active );
```



## Chapter 11: Persisting Data 311

**end;**

**procedure** TForm1.UpdateButtons(**const** Value: Boolean);

**begin**

btnSaveToFile.Enabled := Value;

btnSaveToStream.Enabled := Value;

btnLoadFromFile.Enabled := True;

btnLoadFromStream.Enabled := True;

btnApplyUpdates.Enabled := FDQuery1.UpdatesPending;

btnCancelUpdates.Enabled := FDQuery1.UpdatesPending;

**if** FDQuery1.UpdatesPending **then**

btnEnableCachedUpdates.Caption := ‘Disable Cached Updates’

**else**

btnEnableCachedUpdates.Caption := ‘Enable Cached Updates’;

**end;**

Figure 11-6 shows how the main form looks when the FourFieldsSample.xml file is loaded. A clipped view of this file was shown in Figure 14-5, and it contained one update in the change cache. The UpdateButtons method,

having detected updates pending, has configured the form to enable working with the cache.

### **Figure 11-6: A previously saved file that contained change cache information has been loaded**

312 Delphi in Depth: FireDAC

## **FILE FORMATS**

There are also similarities between LoadFromFile and SaveToFile when it comes to the file format. If you have supplied the filename parameter, you can also specify the format of the file being loaded by providing an FDStorageFormat value in the second parameter. Importantly, if you provide a value other than sfAuto (or if ResourceOptions.DefaultStoreFormat in the FDConnection is set to a value other than sfAuto), the file format that is specified must match that associated with the format written when SaveToFile was called. If there is a mismatch, the call to LoadToFile will raise an exception.

Likewise, if you omit the parameter (in which case sfAuto is used), or you specify sfAuto explicitly, or the FDConnection's

ResourceOptions.DefaultStoreFormat is set to sfAuto, the extension of the file being loaded must match the default extension for the associated format, and that too must correspond to the format used when SaveToFile was called. In other words, LoadFromFile will only work when the format of the file matches the format that LoadFromFile is expecting.

One final word concerning LoadFromFile. FireDAC must have access to the resources associated with reading the format you are loading. As described earlier in this chapter, you can achieve this by placing the corresponding FDStanStorage *format* Link components in your project or by manually adding the necessary FireDAC.Stan.Storage *format* units to your uses clause.

## **MERGING DATA WHEN LOADING**

In most situations, you call LoadFromFile (or LoadFromStream) to activate a FireDAC dataset using the metadata that was previously saved, and load it with the saved data. You might even be loading a change log. As mentioned at the outset of this chapter, this is a capability that previously was only available in Delphi through the ClientDataSet.

But FireDAC goes a step further. FireDAC can call LoadFromFile (or LoadFromStream) on an active FireDAC dataset, which, so long as the

structure of the active dataset is consistent with that of the data being loaded, will merge the data from the file being loaded with that already in the dataset.

ClientDataSets cannot do this.

Being able to merge previously saved data into an active FireDAC dataset is a significant feature that gives you added flexibility. FireDAC provides for this merging in two ways. You can merge the data, and you can merge the metadata.

This is controlled through the `ResourceOptions.StoreMergeData` and

Chapter 11: Persisting Data 313

`ResourceOptions.StoreMergeMeta` properties, respectively. These properties were introduced in Delphi 10 Seattle.

The `StoreMergeData` property is of the type `TFDMergeDataMode`, which is an

enumeration. The declaration of `TFDMergeDataMode` is shown here:

```
TFDMergeDataMode = (dmNone,  
dmDataSet, dmDataAppend, dmDataMerge,  
dmDeltaSet, dmDeltaAppend, dmDeltaMerge);
```

When `StoreMergeData` is set to `dmNone`, the default, no merging, takes place, and the FireDAC dataset is loaded with the contents of the file or stream. If the FireDAC dataset was in the cached updates mode at the time `LoadFromFile` (or `LoadFromStream`) was called, and there were changes in cache, those changes are lost. Similarly, if a record is being edited, but has not yet been posted, when the previously saved data is loaded, those changes are discarded.

The remaining six possible values of `StoreMergeData` are organized into two sets of three values, with one set applying to the data, and the second set applying to the change cache. Let's consider the set that applies to data first, and these values are `dmDataSet`, `dmDataAppend`, and `dmDataMerge`. For these

values, the data that is being loaded will either replace the data in the active dataset (`dmDataSet`), be added to the data in the active dataset (`dmDataAppend`), or be merged with data in the active dataset (`dmDataMerge`).

Replacing the data in the active dataset and adding to (appending) the data in the active dataset does just that. The data is either replaced or appended.

Merging is a little more involved. When merging, if fields in the records being loaded correspond to key field values in the active FireDAC dataset, the active records are updated if values being loaded differ from those in the active dataset.

Otherwise the records are appended.

For the set of values associated with the change cache (Delta), essentially the same rules apply. When StoreMergeData is set to dmDeltaSet, any data and change cache information in the active dataset is replaced by any data and change cache information being loaded. A value of dmDeltaAppend causes any data and change cache information being loaded that is not already in the active dataset to be added to the active dataset's data and change cache.

Finally, merge will either update or append the data and change cache of the active dataset based on the data and change cache that is being loaded.

The StoreMergeMeta property is designed to affect how the metadata found in the file being loaded effects the dataset that is doing the loading.

### 314 Delphi in Depth: FireDAC

StoreMergeMeta is a TFDMergeMetaMode type, which is also an enumerated type. The declaration of TFDMergeMetaMode is shown here:

```
TFDMergeMetaMode = (mmNone, mmMerge, mmAdd,  
mmUpdate, mmAddOrError, mmError);
```

When StoreMergeMeta is set to mmNone, the dataset retains its original metadata, regardless of the contents of the data being loaded. If the structures of the dataset and that of the file being loaded match, everything is fine. If there are discrepancies, data will be loaded into the dataset, using the rules defined by StoreMergeData, only for those fields in the structures that match. For example, if the active dataset includes string fields named "First" and "Second", and the data being loaded defines fields named "First" and "Last", data will only be loaded into the dataset's "First" field.

When StoreMergeMeta is set to mmNone, it is necessary that the FireDAC dataset into which data is being loaded has metadata, or that its potential metadata can be discovered. If this dataset is active, or it is inactive but its structure is defined by persistent fields, that requirement is met. That requirement is also met if the dataset is inactive, but has discoverable metadata.

For example, if the inactive FDQuery has an associated SQL statement,

FireDAC can use that query definition to determine what the metadata is for that dataset. Similarly, an FDTable that points to a table permits FireDAC to assemble the metadata. However, if an FDQuery, for example, has no SQL, an exception is raised if you call one of the Load methods and StoreMergeMeta is set to mmNone, since the FDQuery has no discoverable metadata.

When StoreMergeMeta is set to mmMerge, columns from the dataset being loaded will update the dataset doing the loading. When matching column names are found, any differences in those column's type and size will be updated in the loading dataset to match those values in the dataset being loaded. If columns are found in the dataset being loaded that do not match those in the dataset doing the loading, those columns are added.

If StoreMergeMeta is set to mmAdd, any column found in the dataset being loaded that is not already present in the dataset doing the loading will be added.

Columns with the same names are unaffected in the dataset doing the loading. Assigning mmUpdate to the StoreMergeMeta property will result in matching column names being updated in the dataset doing the loading, based on the metadata found in the dataset being loaded. Column names found in the dataset being loaded that are not present in the dataset doing the loading are ignored.

## Chapter 11: Persisting Data 315

When StoreMergeMeta is set to mmAddOrError, one of two things will happen.

If the dataset doing the loading has no metadata, the metadata found in the dataset being loaded will be used. If the dataset doing the loading has metadata, it must match that of the dataset being loaded by name, type, and size.

Otherwise, an exception is raised.

The final StoreMergeMeta value, mmError, requires that the dataset doing the loading has metadata, and that it matches the metadata of the dataset being loaded. If there is a mismatch on column names, types, or sizes, an exception is raised.

*Note: FireDAC creator, Dmitry Arefiev, notes that StoreMergeMeta is particularly useful when loading a file persisted by saving an*

*FDSchemaAdapter participating in a centralized cached updates session.*

The FDSaveAndLoad project contains code that permits you to test the effects of various StoreMergeData and StoreMergeMeta settings. Two comboboxes on

the main form are initialized when the form is opened to the StoreMergeData and StoreMergeMeta settings of the FDQuery, as shown in this code segment:

```
cbxStoreMergeData.ItemIndex :=
```

```
Ord( FDQuery1.ResourceOptions.StoreMergeData );
```

```
cbxStoreMergeMeta.ItemIndex :=
```

```
Ord( FDQuery1.ResourceOptions.StoreMergeMeta );
```

Likewise, changing the value of these comboboxes updates the associated properties, as shown in the following event handlers:

```
procedure TForm1.cbxStoreMergeDataChange(Sender: TObject);  
begin
```

```
FDQuery1.ResourceOptions.StoreMergeData :=
```

```
TFDMergeDataMode( cbxStoreMergeData.ItemIndex );
```

```
end;
```

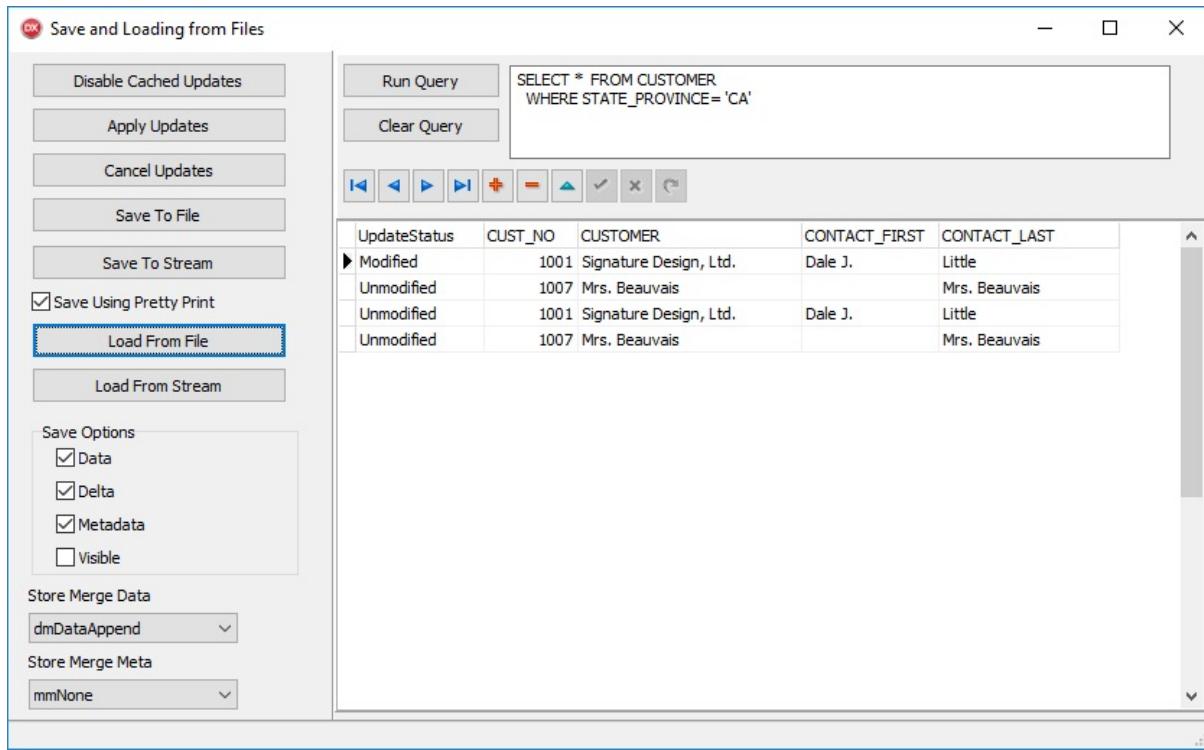
```
procedure TForm1.cbxStoreMergeMetaChange(Sender: TObject);
```

```
begin
```

```
FDQuery1.ResourceOptions.StoreMergeMeta :=
```

```
TFDMergeMetaMode( cbxStoreMergeMeta.ItemIndex );
```

```
end;
```



## 316 Delphi in Depth: FireDAC

Figure 11-7 depicts the FDSaveAndLoad main form after two previously saved files were loaded. The first file, FourFieldsSample.XML, which is shown in Figure 11-5, was loaded with StoreMergeData set to dmNone and StoreMergeMeta set to mmNone. The second file, SampleData.xml, was loaded with StoreMergeData set to dmDataAppend and StoreMergeMeta set to mmNone. A clipped version of this second file is shown in Figure 11-3.

### **Figure 11-7: Use the Store Merge Data and Store Merge Meta comboboxes**

**to test the effects of merging one or more previously saved files with the contents of a FireDAC dataset**

### **Automating Persistence**

As you learned earlier, you can omit the name of the file to which you want to save data or from which you want to load data by providing a file name in the ResourceOptions.PersistentFileName property. Furthermore, you can control the directory in which this file is stored using the

ResourceOptions.DefaultStoreFolder property, and the file extension using the ResourceOptions.DefaultStoreExt property. These two properties are associated with the FDConnection and FDManager to which the FireDAC dataset is

associated.

## Chapter 11: Persisting Data 317

PersistentFileName has another use. If you also set ResourceOptions.Persistent to True, you can simply call Open and Close on the FireDAC dataset to load and save the contents of the file, completely avoiding the need to call LoadFromFile or SaveToFile. The default value of Persistent is False.

When Persistent is True, the filename, format, the stored items, pretty print settings, version number, and merge settings are all determined by the ResourceOptions of the FireDAC dataset being opened and closed.

In addition to the Persistent property, FireDAC provides you with two functions that permit you to discover programmatically if a given FireDAC dataset is going to automatically load when opened and automatically saved when closed.

These Boolean functions are called LoadFromFileAtOpen and SaveToFileAtClose, and their signatures are shown here:

```
function LoadFromFileAtOpen: Boolean;  
function SaveToFileAtClose: Boolean;
```

### Maintaining Backups

When you are saving a FireDAC dataset to a file, FireDAC can maintain a backup file of the previous version of your data. This is especially useful if you want to treat one or more persisted FireDAC datasets as a local database, which can be very nice if you need to implement a briefcase model of data access.

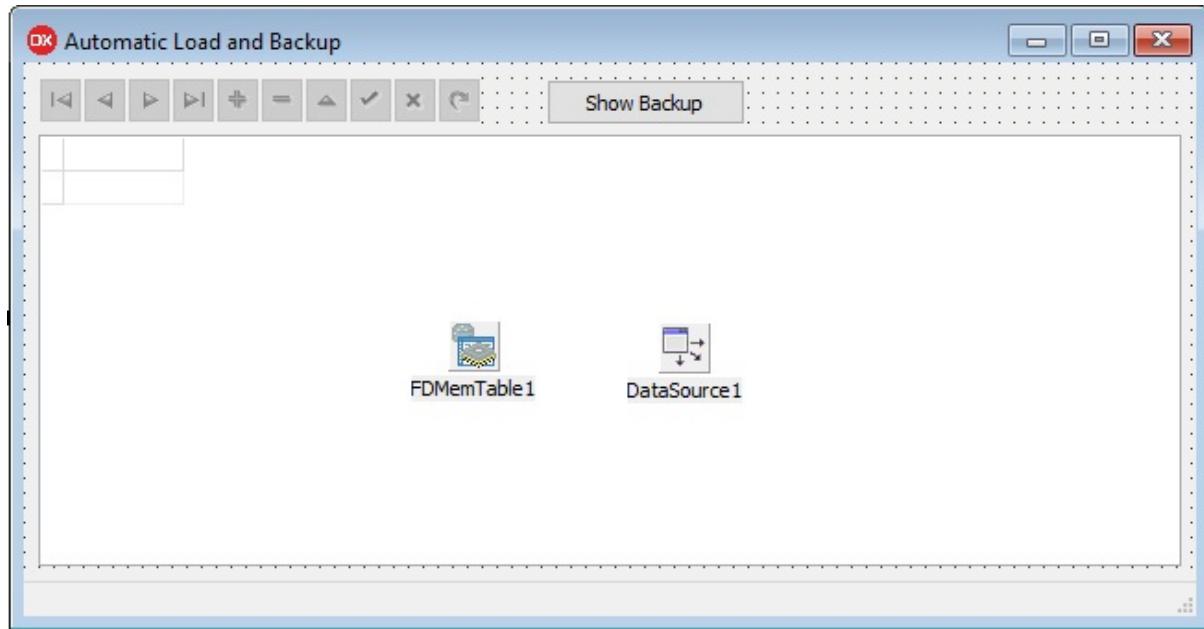
When you set the ResourceOptions.Backup property of a FireDAC dataset to True, FireDAC will create a backup file of your dataset when you save changes, whether you call SaveToFile or have FireDAC automatically save your changes.

Furthermore, you use the ResourceOptions.BackupFolder and ResourceOptions.BackupExt properties of your FireDAC's connection to define a folder in which the backup file should be placed, and the extension to use for the file. The default backup folder is the folder in which you save your data, and the default extension is .bak.

FireDAC saves the backup file using the name of the data file plus the backup

extension. FireDAC creates the backup file using the same format as the data file. As a result, if you save a data file using the binary format, the backup file will be in the binary format.

The project FDAutoLoadBackup demonstrates both automatic persistence as well as FireDAC's backup capability. Figure 11-8 shows the main form of this project in Delphi's form designer.



## 318 Delphi in Depth: FireDAC

**Figure 11-8: The main form of the FDAutoLoadBackup project**

*Code: The FDAutoLoadBackup project can be found in the sample code.*

The OnCreate event handler of this project is used to ensure that the backup folder exists, and that the FDMemTable on the form is configured to auto load and create backups. In addition, if the file to be loaded does not already exist, it is created. This OnCreate event handler is shown here:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  BackupFolder: string;
begin
  FDManager.ResourceOptions.BackupFolder := '$(RUN)\Backup';
  FDManager.ResourceOptions.BackupExt := '.BAK';
  BackupFolder :=
    FDExpandStr(FDManager.ResourceOptions.BackupFolder);
```

```

if not DirectoryExists( BackupFolder ) then
  ForceDirectories( BackupFolder );
  FDManager.ResourceOptions.DefaultStoreFolder := ‘$(RUN)’;
  FDMemTable1.ResourceOptions.Backup := True;
  FDMemTable1.ResourceOptions.PersistentFileName :=
    ‘Clients.xml’;
  FDMemTable1.ResourceOptions.DefaultStoreFormat := sfXML;
  FDMemTable1.ResourceOptions.Persistent := True;
if not FileExists(
  Chapter 11: Persisting Data 319
  FDMemTable1.ResourceOptions.PersistentFileName ) then
begin
  with FDMemTable1.FieldDefs do
    begin
      Clear;
      Add(‘ClientID’, ftInteger);
      Add(‘ClientName’, ftString, 45 );
      Add(‘Address1’, ftString, 50 );
      Add(‘Address2’, ftString, 50 );
      Add(‘LastContact’, ftDateTime );
      Add(‘Notes’, ftMemo );
      Add(‘Active’, ftBoolean);
    end; //with FDMemTable.FieldDefs
  FDMemTable1.CreateDataSet;
end
else
  FDMemTable1.Open;
end;

```

*Note: FDMemTables, like the one used in this example, do not use*

*FDConnections. As a result, the singleton FDManager was used to configure the TFDTopResourceOptions settings, such as DefaultStoreFolder, BackupFolder, and BackupExt. Also, since I have used a substitution variable with the format \$(name) in the BackupFolder property, I used the FDExpandStr function from the FireDAC.Stan.Util unit to expand the substitution variable to a fully qualified directory path.*

*As you may recall from Chapter 3, Configuring FireDAC, the FDManager is the base class for all configuration inheritance*

The automatic saving of the data occurs when the FDMemTable is closed. This closing is performed from the OnClose event handler of the form. This event handler, shown here, also ensures that the current record has been posted before the dataset is closed.

```
procedure TForm1.FormClose(Sender: TObject; var Action:  
TCloseAction);  
begin  
if FDMemTable1.State in dsEditMode then  
  FDMemTable1.Post;  
  FDMemTable1.Close;  
end;
```

## 320 Delphi in Depth: FireDAC

This project also includes a button on the main form labeled Show Backup, which you can use to view the contents of the backup that was created the last time the FDMemTable's contents were saved. This event handler is shown here.

Figure 11-9 shows how the backup looks after some changes were posted.

```
procedure TForm1.btnShowBackupClick(Sender: TObject);  
var  
  BackupForm: TBackupForm;  
  BackupFile: String;  
begin  
  BackupFile := FDManager.ResourceOptions.BackupFolder +  
    ChangeFileExt(
```

```

FDMemTable1.ResourceOptions.PersistentFileName,
FDManager.ResourceOptions.BackupExt );

if not FileExists( FDExpandStr( BackupFile ) then
raise Exception.Create('Backup file has not been created');

BackupForm := TBackupForm.Create( nil );

try

BackupForm.FDMemTable1.LoadFromFile( BackupFile,
TFDStorageFormat.sfXML );

BackupForm.ShowModal;

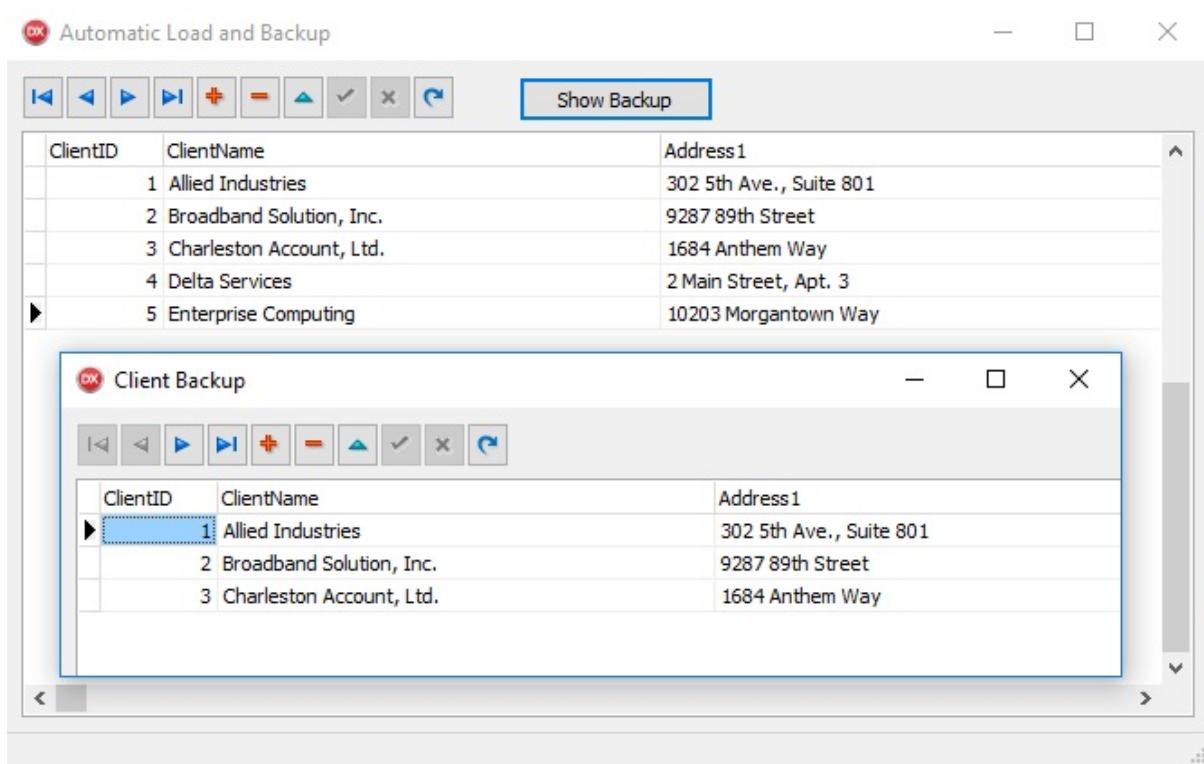
finally

BackupForm.Release;

end;

end;

```



Chapter 11: Persisting Data 321

**Figure 11-9: The Clients table has been created and edited. The contents of the previous backup are shown in the top-level form**

*Note: Recall that you can use the \$(NEXT) substitution variable in path definitions, permitting you to create a different backup each time a file is*

*persisted.*

## Persisting to Streams

Persisting a FireDAC dataset to a stream is very similar to persisting to a file, with just a few differences. These include no default folder option, no automatic load on open and save on close, and no backup option. Otherwise, the options are the same, such as having a choice of persistence format, storage options (data, metadata, and delta), and version.

Persisting a FireDAC dataset to a stream, and restoring that dataset from a stream, is demonstrated in the FDSaveAndLoad project. In this project, the stream is used to write the contents of the FireDAC dataset to a BLOB field of an InterBase table. Loading from a stream is demonstrated by retrieving the previously saved stream into a FireDAC dataset.

### 322 Delphi in Depth: FireDAC

The table that this code writes to and reads from is created in the OnCreate event handler of the main form of the FDSaveAndLoad project if it does not already exist. The relevant code is shown here:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  FDQuery: TFDQuery;
const
  TabName = 'STOREDDATASETS';
begin
  SharedDMVcl.FDConnection.Open();
  FDQuery := TFDQuery.Create( nil );
  try
    FDQuery.Connection := SharedDMVcl.FDConnection;
    FDQuery.Open( 'SELECT RDB$RELATION_NAME ' +
      ' FROM RDB$RELATIONS ' +
      ' WHERE RDB$RELATION_NAME = "' +
      DataStoreTab + '"');
    FDQuery.Open();
    //Create the table if necessary
```

```

if FDQuery.IsEmpty then
begin
  SharedDMVcl.FDConnection.StartTransaction;
  try
    FDQuery1.Connection.ExecSQL(
      ‘CREATE TABLE ”” + DataStoreTab + ”” ( ‘ +
      ‘ DataSetName VarChar( 30 ) ‘ +
      ‘NOT NULL Primary Key, ‘ +
      ‘ DataSet BLOB NOT NULL );
  finally
    SharedDMVcl.FDConnection.Commit;
  end;
  end;
  finally
    FDQuery.Close;
  end;
...

```

### Saving to a Stream

You save to a Stream by calling the FireDAC dataset's SaveToStream method.

This method has the following signature:

Chapter 11: Persisting Data 323

```

procedure SaveToStream(AStream: TStream;
  AFormat: TFDStorageFormat = sfAuto);

```

The first parameter is a required parameter, and it is the stream to which you are writing the data. The second parameter is an optional parameter. If you omit it, FireDAC will use its binary format for the stream. If you want to persist your data to a text-based stream, set AFormat to either sfXML or sfJSON.

Writing a FireDAC dataset to a stream is demonstrated in the OnClick event handler for the button labeled Save To Stream. This event handler is shown

here:

```
procedure TForm1.btnSaveToStreamClick(Sender: TObject);
var
  DataSetName: string;
  ms: TMemoryStream;
  FDQuery: TFDQuery;
begin
  if InputQuery( 'Save to Stream', 'DataSet Name', DataSetName )
  then
    begin
      ms := TMemoryStream.Create;
      try
        FDQuery1.SaveToStream( ms );
        ms.Position := 0;
        FDQuery := TFDQuery.Create( nil );
        try
          FDQuery.Connection := SharedDMVcl.FDConnection;
          FDQuery.Open(
            'SELECT DataSetName, DataSet FROM ' + DataStoreTab +
            ' WHERE DataSetName = "' +
            DataSetName.ToUpper + "'");
          if FDQuery.IsEmpty then
            FDQuery.Append
          else
            FDQuery.Edit;
            FDQuery.Fields[0].AsString := DataSetName;
            TBlobField(FDQuery.Fields[1]).LoadFromStream( ms );
            FDQuery.Post;
        finally
```

```
FDQuery.Free;  
end;  
finally  
 324 Delphi in Depth: FireDAC  
  ms.Free;  
end;  
end;  
end;
```

The essential part of this code is found immediately following the call to the TMemoryStream constructor. After creating the memory stream, the data of the FDQuery is written to it, after which, the Position property of the memory stream is reset to 0. What happens next depends on what you need to do with the stream. In this case, I am writing it to a BLOB field in the StoredDataSets table.

However, I could just as easily have opened a socket connect to some service and sent that stream over the Internet to a receiving endpoint.

When the SaveToStream call is made, the StoreItems, Version, and StorePrettyPrint properties are used to determine what and how to write to the stream.

### **Loading from a Stream**

You load the previously persisted contents of a FireDAC dataset from a stream by calling the LoadFromStream method. Here is the signature of this method:

**procedure** LoadFromStream(AStream: TStream;

AFormat: TFDStorageFormat = sfAuto);

LoadFromStream has one required parameter and one optional parameter. The required parameter is the stream containing the previously persisted data, and the optional parameter permits you to declare the format used to persist the stream. If you omit the second parameter, FireDAC will attempt to read the stream using the binary format. If you know that the stream was created using either XML or JSON formats, you must specify which format was used during

creation. FireDAC raises an exception if there is a mismatch between the format used during creation and the one FireDAC expects when loading the stream.

The use of LoadFromStream is demonstrated in the OnClick event handler of the button labeled Load From Stream. This event handler is shown here:

```
procedure TForm1.btnLoadFromStreamClick(Sender: TObject);
```

```
var
```

```
  FDQuery: TFDQuery;
```

```
  ChooseDataSetForm: TChooseDataSetForm;
```

```
  ms: TMemoryStream;
```

```
begin
```

Chapter 11: Persisting Data 325

```
  FDQuery := TFDQuery.Create( nil );
```

```
try
```

```
  FDQuery.Connection := SharedDMVcl.FDConnection;
```

```
  FDQuery.Open('SELECT * FROM ' + DataStoreTab );
```

```
  if FDQuery.IsEmpty then
```

```
    raise Exception.Create( 'No stored datasets to load' );
```

```
  ChooseDataSetForm := TChooseDataSetForm.Create( nil );
```

```
try
```

```
  ChooseDataSetForm.Caption := DataStoreTab;
```

```
  ChooseDataSetForm.DataSource1.DataSet := FDQuery;
```

```
  if ChooseDataSetForm.ShowModal = mrOK then
```

```
begin
```

```
  ms := TMemoryStream.Create;
```

```
try
```

```
  TBlobField(FDQuery.Fields[1]).SaveToStream( ms );
```

```
  ms.Position := 0;
```

```
  FDQuery1.LoadFromStream( ms );
```

```
finally
```

```
  ms.Free;
```

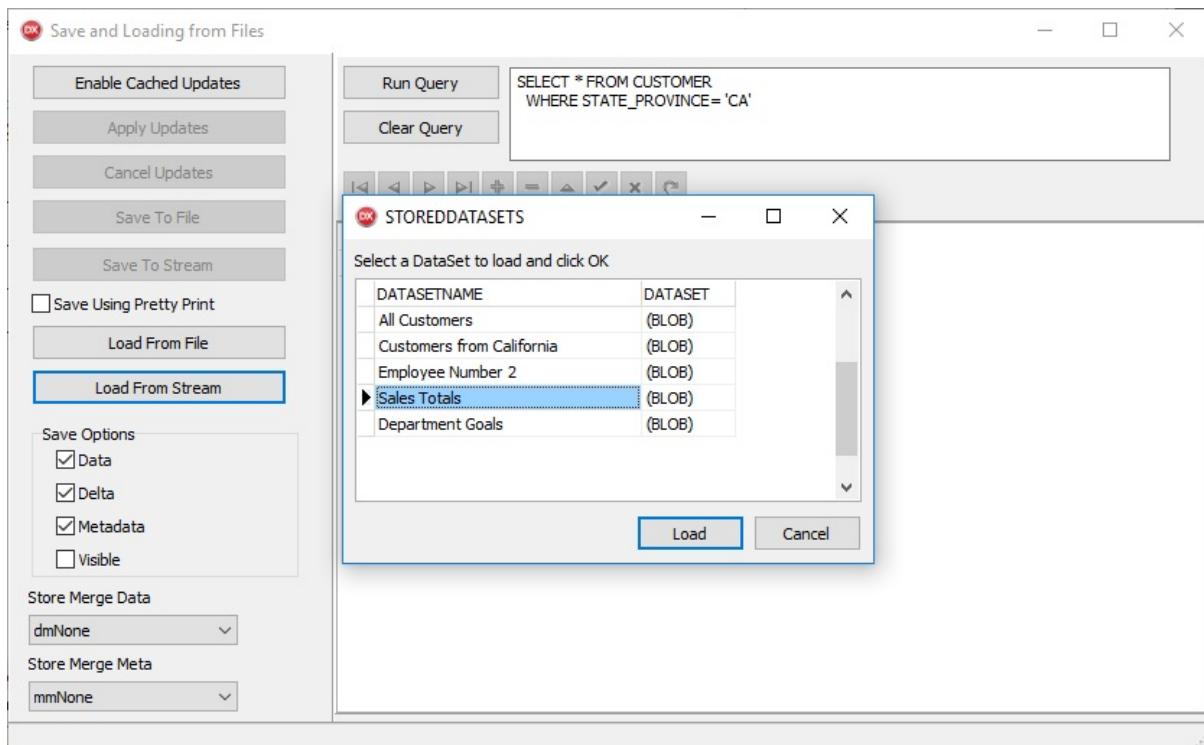
```
end;
```

```

end;
finally
  ChooseDataSetForm.Free;
end;
finally
  FDQuery.Free;
end;
if not FDQuery1.CachedUpdates and FDQuery1.UpdatesPending then
  FDQuery1.CachedUpdates := True;
  UpdateButtons( FDQuery1.Active );
end;

```

This event handler is a bit more involved, in that it must first ask the user to select which of the previously saved datasets they want to load. After querying the `StoredDataSets` table for its records, those records are displayed in the `ChooseDataSetForm` dialog box shown in Figure 11-10. If the user clicks the OK button on the dialog box, the BLOB field from the current record of the query is used to write the previously saved data to a stream. The `Position` property of this stream is then reset to 0 after which, the stream is loaded into `FDQuery1`.



## **Figure 11-10: A saved dataset is about to be loaded into FDQuery1**

As with the LoadFromFile method, how the contents of the stream get loaded depends on the values you provide in the ResourceOptions.StoreMergeData and ResourceOptions.StoreMergeMeta properties. For example, if StoreMergeData

is set to mdDataAppend, the contents of the stream will be added to the records already present in the FireDAC dataset.

### **Persistence and FDSchemaAdapters**

A number of the properties discussed in this chapter have included options for persisting Delta. Delta is associated with the cached updates mode, and it stores the changes that have been cached for datasets, but which have not yet been either applied or canceled.

As you will learn in *Chapter 16, Using Cached Updates*, the centralized model of cached updates permits you to manage cached updates for one or more

FireDAC datasets using a common change log. This common change log is provided for by an FD Schema Adapter, to which each of the participating FireDAC datasets must point. Importantly, not only does this common change

Chapter 11: Persisting Data 327

log identify the changes that have been made to the participating FireDAC datasets, it also maintains the order in which those changes will be applied.

Like FireDAC datasets, schema adapters can also be persisted. In fact, if you are using the centralized model of cached updates, and need to persist the state of your datasets, you do so using the schema adapter.

For example, if you have two FireDAC datasets pointing to the same schema adapter through their SchemaAdapter properties, and the datasets are in cached updates mode, you persist this information using the SaveToFile or SaveToStream methods of the schema adapter. Specifically, it is not necessary to call the SaveToFile or SaveToStream methods of the individual FireDAC datasets.

Furthermore, you restore the persisted state by calling the LoadFromFile or LoadFromStream methods of the schema adapter. Doing so will restore the persisted state of the associated datasets. For example, if the schema adapter and the two FireDAC datasets are closed, and you call the LoadFromFile

method of the schema adapter, the schema adapter as well as the two FireDAC datasets will become active. If the persisted data includes the change cache, that

information will be restored as well.

All of this assumes that the properties of the schema adapter were configured to persist sufficient delta and metadata information using the StoreItems property, and that the StoreMergeData and StoreMergeMeta properties were also properly configured. In addition, it is also assumed that FireDAC datasets with the appropriate component names and configurations are present at the time the persisted data is loaded.

For example, if two FDQueries named CustomerQueryRB and SalesQueryRB are connected to the schema adapter at the time when the schema adapter's SaveToFile method is called, there must two FDQueries with these names connected to the schema adapter at the time that LoadFromFile is invoked, and these two FDQueries must be configured with SQL statements consistent with those used by the FDQueries that were present when the schema adapter was

persisted.

In the next chapter, I will introduce you to the interesting world of FDMemTables.

Chapter 12: Understanding FDMemTables 329

# Chapter 12

## Understanding

### FDMemTables

FireDAC's FDMemTable is an in-memory dataset that implements the TDataSet interface. FDMemTable is Delphi's second in-memory dataset, a relative newcomer compared with ClientDataSet, which made its originally debut in Delphi 3 Client/Server.

In-memory datasets are similar to other datasets with one important distinction

— an in-memory dataset maintains all of its content in memory. There are two consequences to this feature.

First, operations, such as locating records, sorting, setting ranges, filtering, and the like, are very fast, which make them ideal for handling largely read-only data in a cached environment.

Second, there is a limit to the amount of data that an in-memory dataset can hold, though the upper limit is typically so high that this is rarely a concern. For example, I've had in-memory datasets that hold many thousands of records, and in some applications, many millions of records. It really depends on the amount of space consumed by individual records.

That Delphi supports two different in-memory datasets begs the question, “Which one should you use?” The answer is, “The one that best suits your needs.” FDMemTable and ClientDataSet were designed for different purposes, and as a result, each have their own strengths and weaknesses. I find that I use both FDMemTables and ClientDataSets in my applications, where appropriate.

330 Delphi in Depth: FireDAC

*Note: You can find a YouTube video from a presentation I made at CodeRage 9*

*that explains some of these differences. I've learned a lot more about FireDAC*

*datasets since I made that recording, so it includes some inaccuracies.*

*Nonetheless, it is mostly correct. You can find this video by searching YouTube for FDMemTables and ClientDataSets Compared.*

This chapter takes an in-depth look at FDMemTables. It begins by considering the special role that FDMemTables play in applications that use other FireDAC

datasets, such as FDQueries and FDStoredProcs. It continues by demonstrating how to define the structure of FDMemTables. Finally, it concludes with a look at a feature first introduced in Delphi 10.2 Tokyo, which permits you to edit the contents of an FDMemTable at design time.

## The Role of FDMemTable in FireDAC

### Applications

Before the introduction of FDMemTable, Delphi supported a single in-memory dataset, the ClientDataSet. ClientDataSet introduced a collection of features not found in other Delphi data access mechanisms, such as on-the-fly indexing, cloned cursors, cached updated, dataset persistence, aggregate fields, nested datasets, to name some of the most salient features. As a result, if you wanted to use these features in your applications, you needed to employ the ClientDataSet.

The same cannot be said about FireDAC. Specifically, FireDAC datasets, such as FDQueries, FDTables, and FDStoredProcs, support most of these advanced

features directly. For example, you can create temporary indexes on-the-fly for the result set returned by an FDQuery by assigning a value to the FDQuery's IndexFieldNames property. Likewise, you can write the records returned by the execution of a stored procedure to a file by calling an FDStoredProc's

SaveToFile method. And both of these classes, as well as the FDTable, implement cached updates.

Granted, these features are implemented by these datasets internally by means of an encapsulated FDMemTable, but that is not a relevant detail. What is relevant is that you do not need to explicitly introduce FDMemTables to your application in order to use these advance features.

On the other hand, there are capabilities supported by FDMemTables that are very limited in other FireDAC datasets. These include cloned cursors and nested

datasets, and these are two features that are hugely powerful when applied correctly.

In addition, the cached nature of in-memory datasets alone is a game-changing capability. And Delphi 10.2 Tokyo introduced design-time editing of

FDMemTable data, which is interesting, and not currently available in other FireDAC datasets.

What I am trying to say is that FDMemTables can play an important role in your applications, though this role is more focused than that of ClientDataSets. If you need cached updates, temporary indexes, aggregate fields, and dataset

persistence, you will use these capabilities as they are exposed from other FireDAC datasets. On the other hand, if you want the blinding performance of in-memory data, cloned cursors, and nested datasets, FDMemTable is there to help.

*Note: While the `CloneCursor` method is exposed by all FireDAC datasets, it really only applies to FDMemTables. Similarly, nested datasets are fully supported by FDMemTables. By comparison, FDQueries support nested datasets only when they are connected to a database whose driver supports nested datasets, such as Oracle and PostgreSQL.*

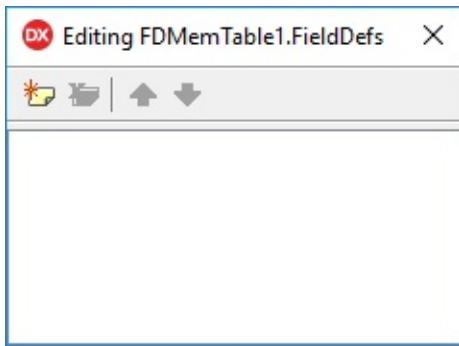
Let's begin this discussion with a look at defining the structure of an FDMemTable.

## Defining an FDMemTable's Structure

There are a number of ways in which you can define structure for an FDMemTable. Some of these techniques create the structure based on the results of a query or stored procedure call. We saw this in *Chapter 5, More Data Access*, in the section *FDCommand, FDTableAdapter, and FDMemTable*. The FDMemTable in that example obtained its structure from the SQL assigned to the FDCommand.CommandText property. Similarly, using the CopyDataSet

method of the FDMemTable will create metadata for the FDMemTable based on the dataset that is being copied.

While these techniques are valuable and important, there are times when you need to create an FDMemTable that is not based on an existing result set or dataset. For example, you might want to create a table to hold a list of file names, file sizes, and creation dates based on the contents of a folder, and



## 332 Delphi in Depth: FireDAC

populate this table when your application loads so that this information can be searched quickly while the application is running without having to repeatedly read the contents of the directory. This can be accomplished by creating an FDMemTable that has fields in which this information can be held. Doing this, however, requires that you know how to define fields for your FDMemTable,

since there is no database structure from which these fields can be derived. There are two mechanisms by which you can define the structure of an FDMemTable — FieldDefs and Fields.

### **Defining Structure Using FieldDefs**

The FieldDefs property of an FDMemTable is a collection of TFieldDef instances. Each FieldDef represents a column, or field of the FDMemTable if the FDMemTable is activated.

You can configure FieldDefs either at design time or at runtime. To define the structure of an FDMemTable at design time, you use the FieldDefs collection editor to create individual FieldDefs. You then use the Object Inspector to configure each FieldDef, defining the field name, data type, size, or precision, among other options.

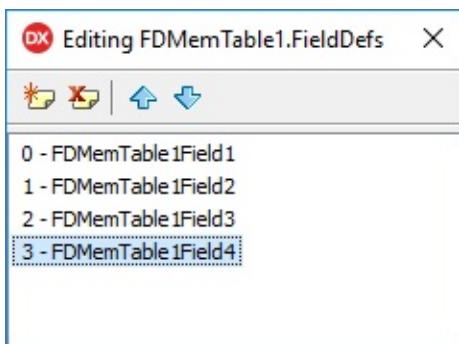
At runtime, you define your FieldDef objects by calling the FieldDef's AddFieldDef or Add methods.

This section begins by showing you how to create your FDMemTables's structure at design time. Defining the structure at runtime is shown later in this chapter.

### **CREATING FIELDDEFS AT DESIGN TIME**

You create FieldDefs at design time using the FieldDefs collection editor. To display this collection editor, select the FieldDefs property of an FDMemTable in the Object Inspector and click the displayed ellipsis button.

The FieldDefs collection editor is shown in the following illustration:

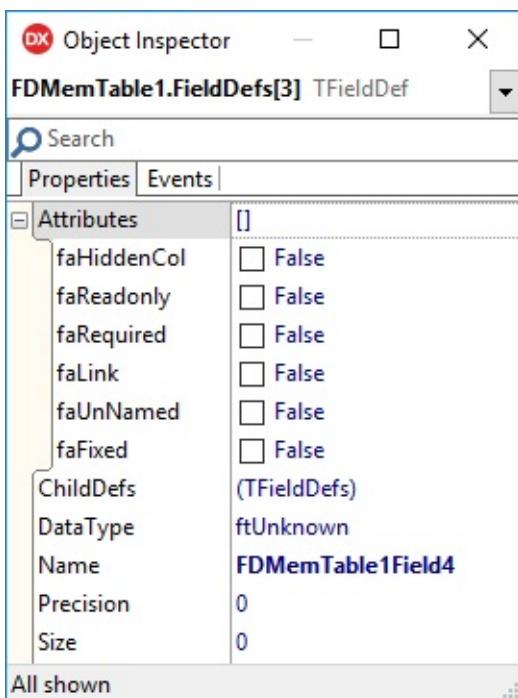


## Chapter 12: Understanding FDMemTables 333

Using the FieldDefs collection editor, click the Add New button (or press Ins) once for each field that you want to include in your FDMemTable. Each click of the Add New button (or press of Ins) will create a new FieldDef instance, which will be displayed in the collection editor. For example, if you add four new FieldDef instances to the FieldDefs collection editor, it will look something like that shown here:

You must configure each FieldDef that is added to the FieldDefs collection editor before you activate the FDMemTable. To configure a FieldDef, select the FieldDef you want to configure in the collection editor or the Structure Pane, and then use the Object Inspector to set its properties. Figure 12-1 shows how the Object Inspector looks when a FieldDef is selected. (Notice that the

Attributes property has been expanded to display its subproperties.)



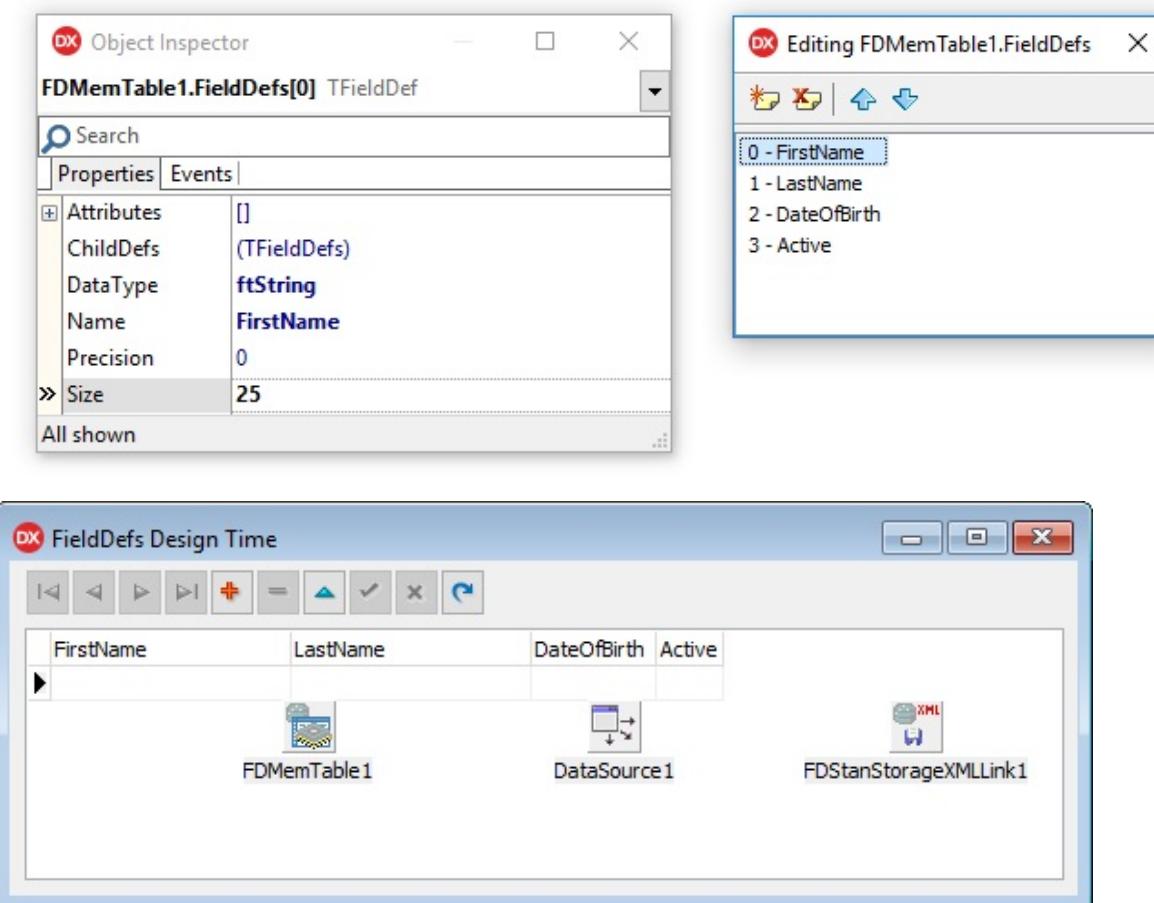
### Figure 12-1: A FieldDef is selected in the Object Inspector

At a minimum, you must set the `DataType` property of each `FieldDef`. You will also want to set the `Name` property. The `Name` property defines the name of the corresponding `Field` that will be created.

Other properties you will often set include the `Size` property, which you define for String, BCD (binary coded decimal), byte, and VarByte fields, and the

`precision` property for BCD fields. Similarly, if a particular field requires a value before the record to which it is associated can be posted, set the `faRequired` flag in the `Attributes` set property.

Figure 12-2 shows both the `FieldDefs` collection editor (with the first field selected), as well as the Object Inspector. Names, data types, and sizes have now been defined for each of the four fields defined for this `FDMemTable`.



**Figure 12-2: Four FieldDef instances have been configured for an FDMemTable**

After setting the necessary properties of each FieldDef, you must create the FDMemTables's data store before you can use it. You can do this at either design time or runtime.

To create the FDMemTable's data store at design time, select the FDMemTable and use the Object Inspector to set its Active property to True. Creating the FDMemTable data store at design time makes the FDMemTable active. This

data store is necessarily empty, as shown in Figure 12-3.

There are several advantages to creating the FDMemTable data store at design time. The first is that the active FDMemTable has Fields (dynamic fields, to be precise), and these Fields can easily be hooked up to data-aware controls such as DBEdit, DBLabel, and DBImage, or to a BindSourceDB (if you are using

LiveBindings).

### **Figure 12-3: The FieldDefs define the fields of this active FDMemTable**

336 Delphi in Depth: FireDAC

*Code: You will find the FDFieldDefsDesignTime project in the code download.*

*See Appendix A for more information.*

The second advantage is that you can save the FDMemTable to a file. When you save an FDMemTable to a file, you are saving its metadata as well as its data.

But in this case, there is no data. Nonetheless, the metadata is valuable, in that any FDMemTable that subsequently loads the saved file from disk will become active and will have the structure that you originally defined.

Saving an FDMemTable to file at design time, like any other FireDAC dataset, involves right-clicking the dataset and selecting Save To File. Saving and loading FireDAC datasets was discussed in greater detail in *Chapter 11, Persisting Data*.

## **CREATING FIELDDEFS AT RUNTIME**

Being able to create FieldDefs at design time is an important capability, in that the Object Inspector provides you with assistance in defining the various properties of each FieldDef you add. However, there may be times when you cannot know the structure of the FDMemTable that you need until runtime.

There are two methods that you can use to define the FieldDefs property at runtime. The easiest technique is to use the Add method of the TFieldDefs class.

The following is the syntax of Add:

```
procedure Add(const Name: String; DataType: TFieldType;  
Size: Integer = 0; Required: Boolean = False);
```

This method has two required parameters and two optional parameters. The first parameter is the name of the field and the second is its type. If you need to set the Size property, as is the case with fields of type ftString and ftBCD, set the Size property to the size of the field. For required fields, set the fourth property to a Boolean True.

The following code sample creates an in-memory table with five fields:

```
const
```

```
DataFile = ‘mydata.xml’;
```

```
procedure Tmainform.FormCreate(Sender: TObject);
```

```
begin
```

```
if FileExists( DataFile ) then
```

```
Chapter 12: Understanding FDMemTables 337
```

```
FDMemTable1.LoadFromFile( DataFile )
```

```
else
```

```
begin
```

```
with FDMemTable1.FieldDefs do
```

```
begin
```

```
Clear;
```

```
Add(‘FirstName’,ftString, 20);
```

```
Add(‘LastName’,ftString, 25);
```

```
Add(‘DateOfBirth’,ftDate);
```

```
Add(‘Active’,ftBoolean);
```

```
end; //with FDMemTable1.FieldDefs
```

```
FDMemTable1.CreateDataSet;
```

```
end;
```

**end;**

This code begins by defining the name of the data file, and then tests whether or not it already exists. When it does not exist, the Add method of the FieldDefs property is used to define the structure, after which, the in-memory dataset is created using the CreateDataSet method. Setting the Active property of the FDMemTable to True would produce the same result. CreateDataSet was

introduced in FDMemTable to provide compatibility with ClientDataSets.

If you consider how the Object Inspector looks when an individual FieldDef is selected in the FieldDefs collection editor, you will notice that the Add method is rather limited. Specifically, you cannot create hidden fields, readonly fields, or BCD fields where you define precision, using the Add method. For these

more complicated types of FieldDef definitions, you can use the AddFieldDef method of the FieldDefs property. The following is the syntax of AddFieldDef:

```
function AddFieldDef: TFieldDef;
```

As you can see from this syntax, this method returns a TFieldDef instance. Set the properties of this instance to configure the FieldDef. The following code sample shows how to do this:

**const**

```
DataFile = 'mydata.xml';
```

```
procedure Tmainform.FormCreate(Sender: TObject);
```

```
begin
```

```
if FileExists( DataFile ) then
```

```
  FDMemTable1.LoadFromFile( DataFile )
```

338 Delphi in Depth: FireDAC

```
else
```

```
begin
```

```
  with FDMemTable1.FieldDefs do
```

```
begin
```

```
  Clear;
```

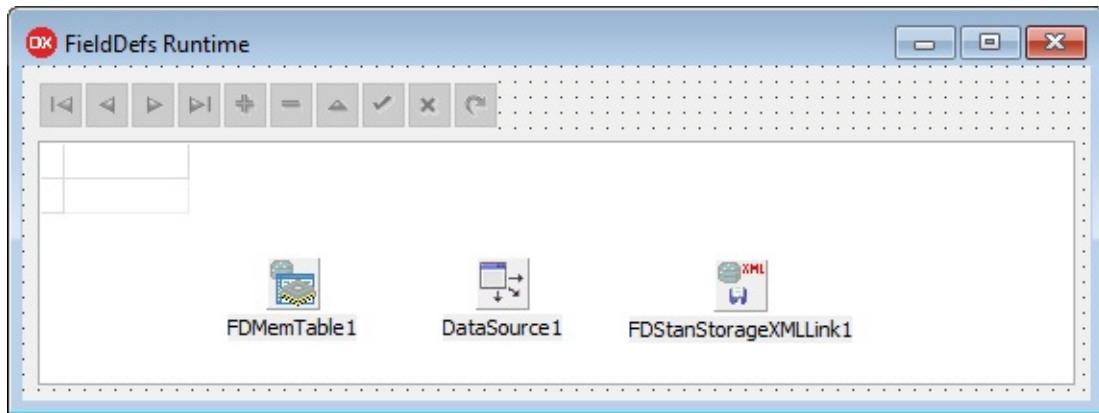
```
  with AddFieldDef do
```

```
begin
```

```
Name := 'First Name';
DataType := ftString;
Size := 20;
end; //with AddFieldDef do
with AddFieldDef do
begin
Name := 'Last Name';
DataType := ftString;
Size := 25;
end; //with AddFieldDef do
with AddFieldDef do
begin
Name := 'Date of Birth';
DataType := ftDate;
end; //with AddFieldDef do
with AddFieldDef do
begin
Name := 'Active';
DataType := ftBoolean;
end; //with AddFieldDef do
end; //with FDMemTable1.FieldDefs
FDMemTable1.CreateDataSet;
end;
end;
```

Both of these techniques are demonstrated in the FDFieldDefsRuntime project.

The main form of this project is shown in Figure 12-4.



Chapter 12: Understanding FDMemTables 339

**Figure 12-4: The FDFieldDefsRuntime project main form**

*Code: You can find the FDFieldDefsRuntime project in the code download.*

### **Defining an FDMemTable's Structure Using Fields**

While the FieldDefs property provides you with a convenient and valuable mechanism for defining an FDMemTable's structure, it has several shortcomings. Specifically, you cannot use FieldDefs to create virtual fields, which include calculated fields, lookup fields, and aggregate fields.

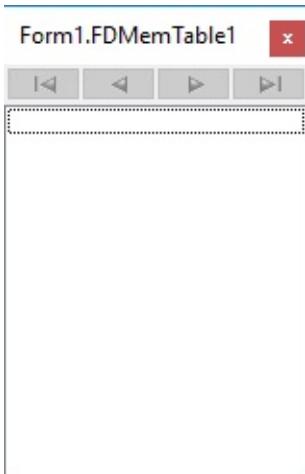
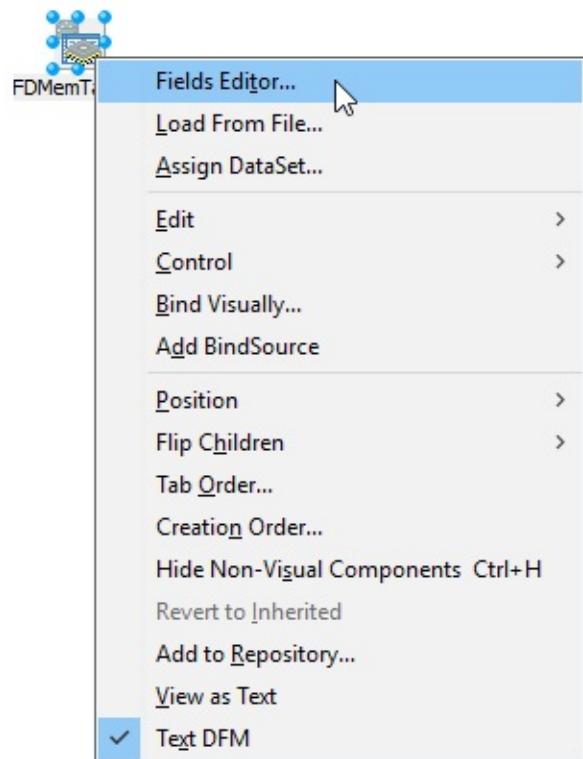
Similar to using FieldDefs to define the structure of an FDMemTable, you can define an FDMemTable's structure using fields either at design time or at runtime. Since the design time technique is the easiest to demonstrate, it is discussed next. Defining an FDMemTable's structure using Fields at runtime is shown later in this section.

### **CREATING FIELDS AT DESIGN TIME**

You define the fields that represent the structure of an FDMemTable at design time using the Fields Editor. Unfortunately, this process is a bit more work than the process of using FieldDefs. Specifically, using the FieldDefs collection editor, you can quickly add one or more FieldDef definitions, each of which defines the characteristic of a corresponding field in the FDMemTable's

structure.

Using the Fields Editor technique to define the structure of your FDMemTable, you must add one field at a time. All this really means is that it takes a little longer to define an FDMemTable's structure using fields than it does using FieldDefs.

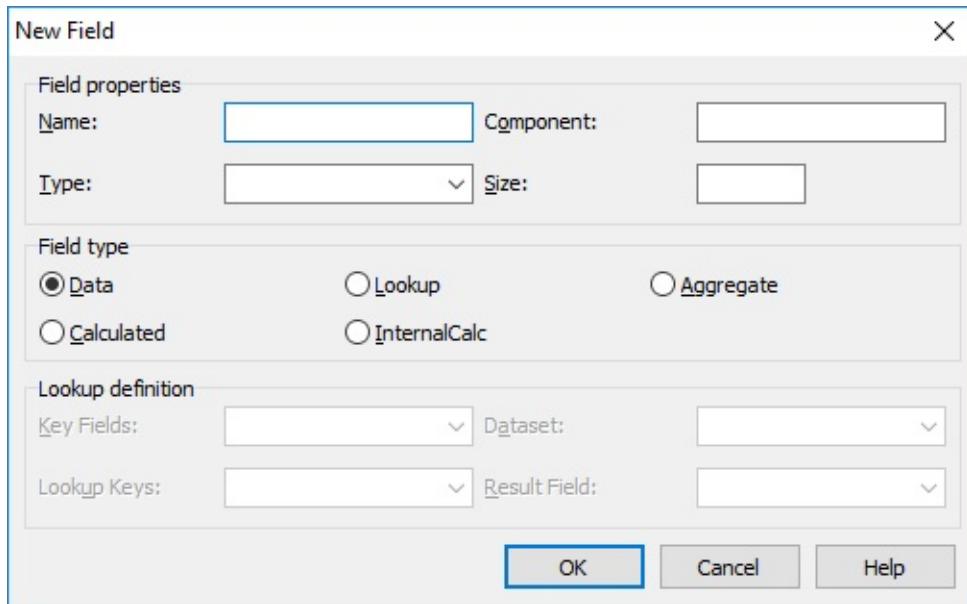


## 340 Delphi in Depth: FireDAC

To create a field, right-click an FDMemTable and select Fields Editor from the displayed context menu. Alternatively, double-click the FDMemTable:

Delphi responds by displaying the Fields Editor.

Right-click the Fields Editor and select New Field (or press the Ins key or Ctrl-N). The New Field dialog box is displayed, as shown in the Figure 12-5.



## Chapter 12: Understanding FDMemTables 341

**Figure 12-5: You use the New Field dialog box to define a persistent Field**

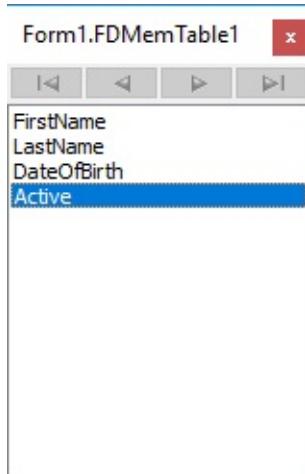
To define the structure of your FDMemTable, you use Data fields. A Data field is one whose data is stored in the FDMemTable and can be saved to a file or to a stream. By default, Field Type is set to Data, and when created this way, it is a persistent field.

*Note: The other field types, Lookup, Aggregate, Calculated, and InternalCalc, are virtual fields, and are described in detail in Chapter 10, Creating and Using Virtual Fields .*

Define your field by providing a field name and field type, at a minimum. The value you enter in the Name field defines the name of the column in the FDMemTable. You set Type to one of the supported Delphi field types, such as String, Integer, DateTime, and so forth.

If your field is one that requires additional information, such as size, provide that information as well. You can also optionally set Component. By default, Component is set to the name of the dataset concatenated with the Name value.

You can set Component manually, but if you do so, ensure that it will be a unique component name, as this is the name that is used for the TField object



## 342 Delphi in Depth: FireDAC

that will be created, and components names must be unique within a given form, frame, or data module.

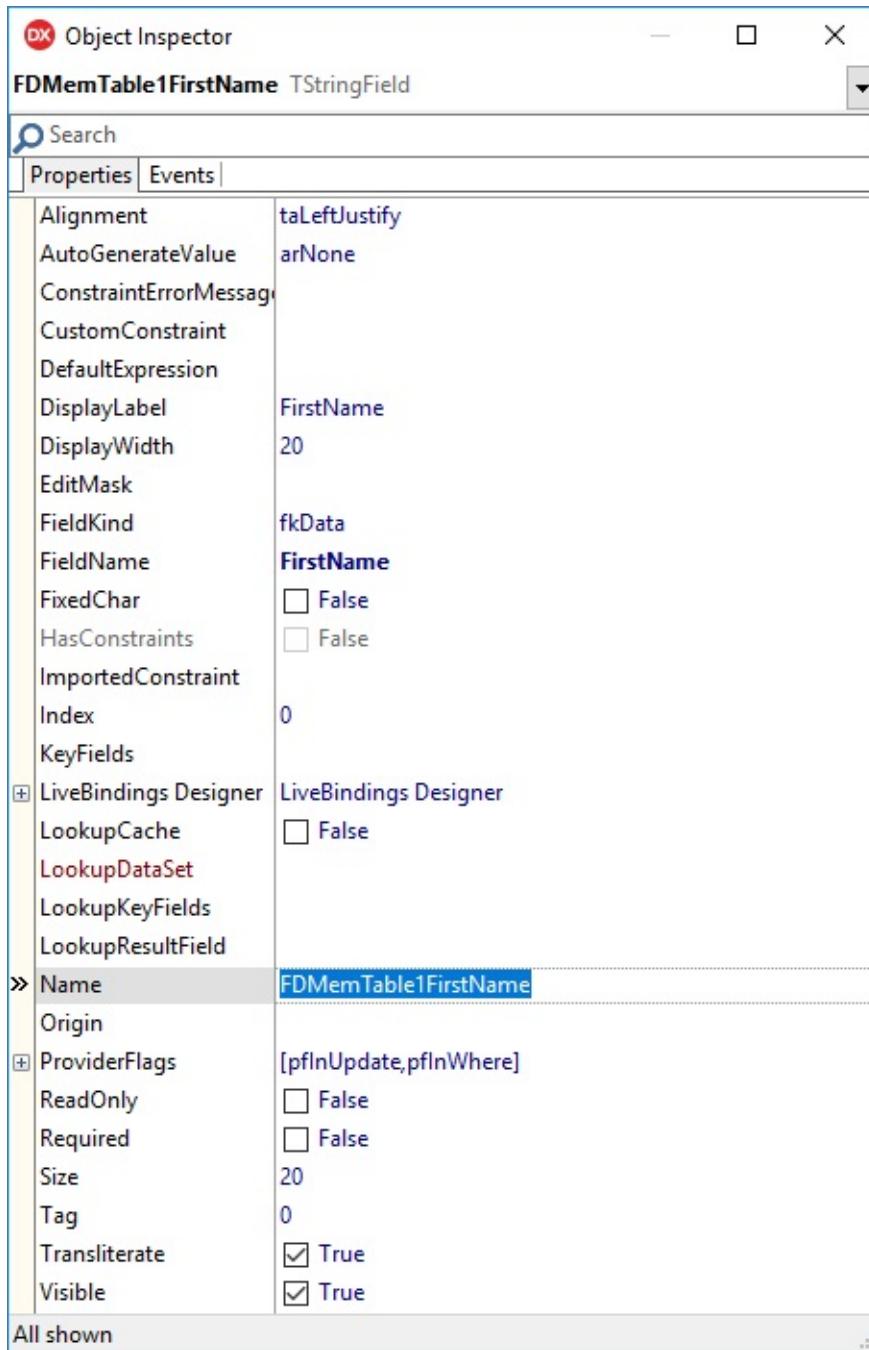
When done, select OK to save your new field. That field will now appear in the Fields Editor. The following illustration depicts the Fields Editor after four fields have been added using the New Field dialog box.

The field that is created when you accept the New Field dialog box is one of the descendants of TField, based on the value you set Type to on the New Field dialog box. For example, if you set Type to string, a TStringField is created. By comparison, if you set Type to integer, a TIntegerField is created.

You cannot change a field type once it has been created. If you have mistakenly selected the wrong value for Type, you need to delete that field and create a new field.

You also cannot use the New Field dialog box to make changes to a field once it has been created. However, there is really no need to do that since the created field is a published member of the form, data module, or frame on which your FDMemTable appears. As a result, you can select the field in the Object

Inspector and make changes there. For example, Figure 12-6 shows the Object Inspector for a string data field named FirstName that was created by the New Field dialog box.



Chapter 12: Understanding FDMemTables 343

**Figure 12-6: Use the Object Inspector to modify a Field that you have created**

344 Delphi in Depth: FireDAC

That fields are published members of the container on which the FDMemTable resides results in another interesting feature. They appear in the container's class type definition, as shown in the following type definition from the

FDFIELDSDesignTime project:

**type**

```
TForm1 = class(TForm)
  FDMemTable1: TFDMemTable;
  DataSource1: TDataSource;
  DBGrid1: TDBGrid;
  DBNavigator1: TDBNavigator;
  FDMemTable1FirstName: TStringField;
  FDMemTable1LastName: TStringField;
  FDMemTable1DateOfBirth: TStringField;
  FDMemTable1Active: TBooleanField;
```

**private**

```
{ Private declarations }
```

**public**

```
{ Public declarations }
```

**end;**

*Code: You can find the FDFIELDSDesignTime project in the code download.*

It is interesting to note that these fields are defined in this fashion even before the FDMemTable's data store has been created.

Fields that are created prior to an FDMemTable being made active are known as *persistent fields*. (When you create an FDMemTable's data store based on FieldDefs, fields are also created, but those are known as *dynamic fields*, since they are created as part of the activation process.)

Persistent fields can be created for any type of dataset, not just FireDAC datasets, or even FDMemTables in particular. However, the role that fields play in FDMemTables is different from that for other datasets (other than the ClientDataSet), in that FDMemTables and ClientDataSets are the only datasets whose physical structure can be defined using Data field type definitions.

If you want to create the data store of an FDMemTable whose structure is defined at design time, you use the same steps as you would with an FDMemTable defined by FieldDefs — once you have defined the fields, you

make the FDMemTable active by setting its Active property to True.

## Chapter 12: Understanding FDMemTables 345

Similarly, even though you have defined the Fields at design time, you can still defer the creation of the FDMemTable's data store until runtime. Again, just as you do with an FDMemTable whose structure is defined by FieldDefs, you

create the FDMemTable data store at runtime by calling the FDMemTable's CreateDataSet method or by setting its Active property to True.

*Note: Saving an FDMemTable, either to a file or to a stream, saves its structure based on non-virtual fields, with one exception. Specifically, any dynamic fields created by FieldDefs, or persistent Data field type fields created through fields, contribute to the metadata that is persisted. Virtual fields, with the exception of InternalCalc fields, do not. Virtual fields, other than InternalCalc fields, are features of the FDMemTable, as well as datasets in general, but not of the internal data store, and therefore, no metadata concerning these fields is persisted when you save to a file or a stream. InternalCalc fields, despite being virtual fields, are managed by the internal store. As a result, metadata about these fields is persisted when you save to a file or stream.*

## CREATING FIELDS AT RUNTIME

Earlier in this chapter, where an FDMemTable structure was defined using FieldDefs, I described how you can define the structure either at design time or at runtime. As explained in that section, the advantage of using design-time configuration is that you can use the features of the Object Inspector to assist in the definition of the FDMemTable's structure. It also lets you conveniently hook up data-aware controls or LiveBindings. This approach, however, is only useful if you know the structure of your FDMemTable in advance. If you do not, your only option is to define your structure at runtime.

You define your fields at runtime using the methods and properties of the appropriate TField descendant class. Specifically, you call the constructor of the appropriate TField, setting the properties of the created object to define its characteristics.

The most important property of the constructed field object is the DataSet property. This property defines to which TDataSet you want the field associated (which will be an FDMemTable in this case, since we are discussing this type of dataset). After creating all of the fields, you call the FDMemTable's

CreateDataSet method or set the FDMemTable's Active property to True. Doing so creates the FDMemTable's structure based on the fields to which it is

associated.

## 346 Delphi in Depth: FireDAC

The project named FDFIELDSRuntime includes examples of creating an FDMemTable's structure at runtime using fields. This is demonstrated in the following code segment:

```
procedure Tmainform.FormCreate(Sender: TObject);  
begin  
  if FileExists( DataFile ) then  
    FDMemTable1.LoadFromFile( DataFile )  
  else  
    begin  
      with TStringField.Create(Self) do  
        begin  
          Name := 'FDMemTable1FirstName';  
          FieldKind := fkData;  
          FieldName := 'FirstName';  
          Size := 20;  
          DataSet := FDMemTable1;  
        end; //First Name  
      with TStringField.Create(Self) do  
        begin  
          Name := 'FDMemTable1LastName';  
          FieldKind := fkData;  
          FieldName := 'LastName';  
          Size := 25;  
          DataSet := FDMemTable1;  
        end; //Last Name
```

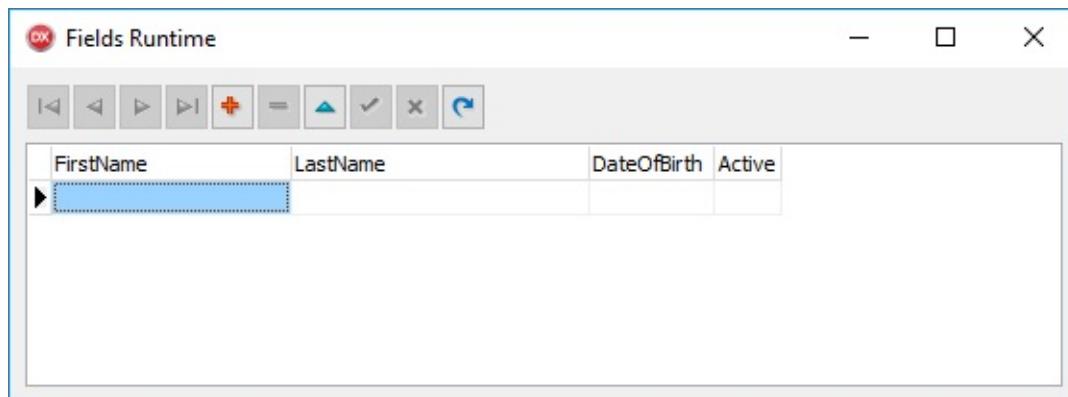
```

with TDateField.Create(Self) do
begin
  Name := 'FDMemTable1DateOfBirth';
  FieldKind := fkData;
  FieldName := 'DateOfBirth';
  DataSet := FDMemTable1;
end; //Date of Birth

with TBooleanField.Create(Self) do
begin
  Name := 'FDMemTable1Active';
  FieldKind := fkData;
  FieldName := 'Active';
  DataSet := FDMemTable1;
end; //Active Name

  FDMemTable1.CreateDataSet;
end;
end;

```



Chapter 12: Understanding FDMemTables 347

*Code: You can find the FDFIELDSRuntime project in the code download.*

Figure 12-7 shows the main form from the FDFIELDSRuntime project that uses Fields to create an FDMemTable structure at runtime.

**Figure 12-7: An FDMemTable structure created at runtime using fields Loading FDMemTables Directly From DataSets**

As I mentioned at the beginning of this chapter, the most common reason for you to use an FDMemTable is to cache records in memory for high-speed operations. In the preceding section, you learned how to define the structure of an FDMemTable, after which you can permit the user to add data, or you can add data programmatically. This approach gives you complete control over the resulting structure of the FDMemTable, relying solely on the FieldDef or field definitions that you provide either at design time or at runtime.

There are other ways in which an FDMemTable's structure can be defined. If you have previously saved an FDMemTable to a file or a stream, loading that file or stream back into an FDMemTable (using LoadFromFile or LoadFromStream) will restore both the structure as well as the data. Saving and loading data to a file or stream is a feature supported by all of FireDAC's datasets, not only FDMemTables, and was discussed in detail in *Chapter 11, Persisting Data*.

You can also define structure by loading an FDMemTable through an FDTableAdapter. In those situations, the FDTableAdapter is associated with an FDCmd instance which defines a SELECT query from which the structure

### 348 Delphi in Depth: FireDAC

of the FDMemTable will be derived. This technique is useful when you want to base the structure of the FDMemTable on the definitions found in an underlying database. Loading an FDMemTable's structure (and data) from an FDTableAdapter is demonstrated in *Chapter 5, More Data Access*.

There is a final mechanism for defining an FDMemTable's structure, and in most cases, loading data as well. You can load data from an existing dataset into an FDMemTable.

There are two ways to load data from an existing dataset into an FDMemTable.

The most flexible is to use the CopyDataSet method, which permits you to copy data from any dataset. The second is to assign the Data property of a FireDAC dataset to the Data property of an FDMemTable. Both of these techniques are discussed in the following sections.

### **Loading an FDMemTable Using CopyDataSet**

The CopyDataSet method is introduced in the FDDataset class, which means that it is available for all FireDAC datasets, including FDTables, FDQueries, and FDMemTables. When you call CopyDataSet, you affect a destination FireDAC dataset, the one from which CopyDataSet is invoked, based on one or more characteristics of a source TDataSet. These characteristics can include structure (the fields and their data types), indexes, calculated fields, aggregate fields, and most importantly, data.

CopyDataSet takes two parameters. The first is the source TDataSet instance. This is the dataset whose characteristics are being copied. Importantly, this dataset does not need to be a FireDAC dataset — it can be any TDataSet descendant, such as SQLDataSet, ClientDataSet, or ADOTable, to name just a few possibilities.

The second parameter is a set of flags that identify the characteristics that will be copied from the source dataset, and how they are copied. Here is the syntax of CopyDataSet:

```
procedure CopyDataSet(ASource: TDataset;  
AOptions: TFDCopyDataSetOptions = [coRestart, coAppend]);
```

And here is the definition of the AOptions parameter:

```
TFDCopyDataSetOptions = set of (coStructure, coCalcFields,  
coIndexesReset, coIndexesCopy,  
coConstraintsReset, coConstraintsCopy,  
coAggregatesReset, coAggregatesCopy,  
coRestart, coAppend,  
coEdit, coDelete, coRefresh);
```

Before I continue, it's worth noting that you can call CopyDataSet from other FireDAC datasets, including FDTables and FDQueries. However, with these classes, the benefits are far more limited. For example, while CopyDataSet is a great way to define an FDMemTable's structure (its fields and their data types), doing this makes little sense with other FireDAC datasets. For example, both an FDTable and an FDQuery get their structure from some underlying database or an SQL statement. If you are creating a new data structure, you are likewise doing so in the underlying database, not in the FDQuery itself.

As for populating an FDTable, FDQuery, or FDStoredProc with data, that is typically done in order to write data into the underlying database. And while that data might come from sources similar to where an FDMemTable's data can

come from (user input or explicit programmatic operations), the end purpose is to update the database. By comparison, data loaded into an FDMemTable might never affect an underlying database, depending on the reason for your use of the FDMemTable.

Granted, you might find some very useful purposes for calling CopyDataSet from an FDQuery, and this is why I will discuss this operation briefly later in this chapter, but for now, I am going to focus on using CopyDataSet to define the structure of, and insert data into, FDMemTables.

When used with an FDMemTable, there are two primary reasons to call CopyDataSet. The first is to define the structure of an FDMemTable, as well as other properties, such as indexes, from an existing Delphi dataset. When used in this manner, you will not load data, but you can then use this structure to add data programmatically, permit the user to add data through the user interface, or do anything else that a structure-only in-memory table can do (such as save this structure to a file or stream for a future use).

The second reason to call CopyDataSet, and probably the most common one, is to load both structure and data from an existing Delphi dataset. You can then work with this data in memory, save the data (and structure) to a file or stream, transfer the FDMemTable to some other process, permit the user to work with the data through the user interface, and more.

It is the second, optional parameter of the CopyDataSet method that controls exactly what is copied. This parameter, named AOptions, consists of a set of zero or more flags that define what is to be copied (though with zero flags,

### 350 Delphi in Depth: FireDAC

nothing will happen, so it might be better to say it consists of a set of one or more flags).

The default flags are coRestart and coAppend, and if these flags are the only ones you need, you can omit the second parameter from a call to CopyDataSet.

These defaults are well suited for loading data from a dataset into an FDDataset whose structure is already defined. With these flags, data from fields of the same names from the source dataset are copied into the corresponding fields in the FDDataset. Of course, this requires that the fields

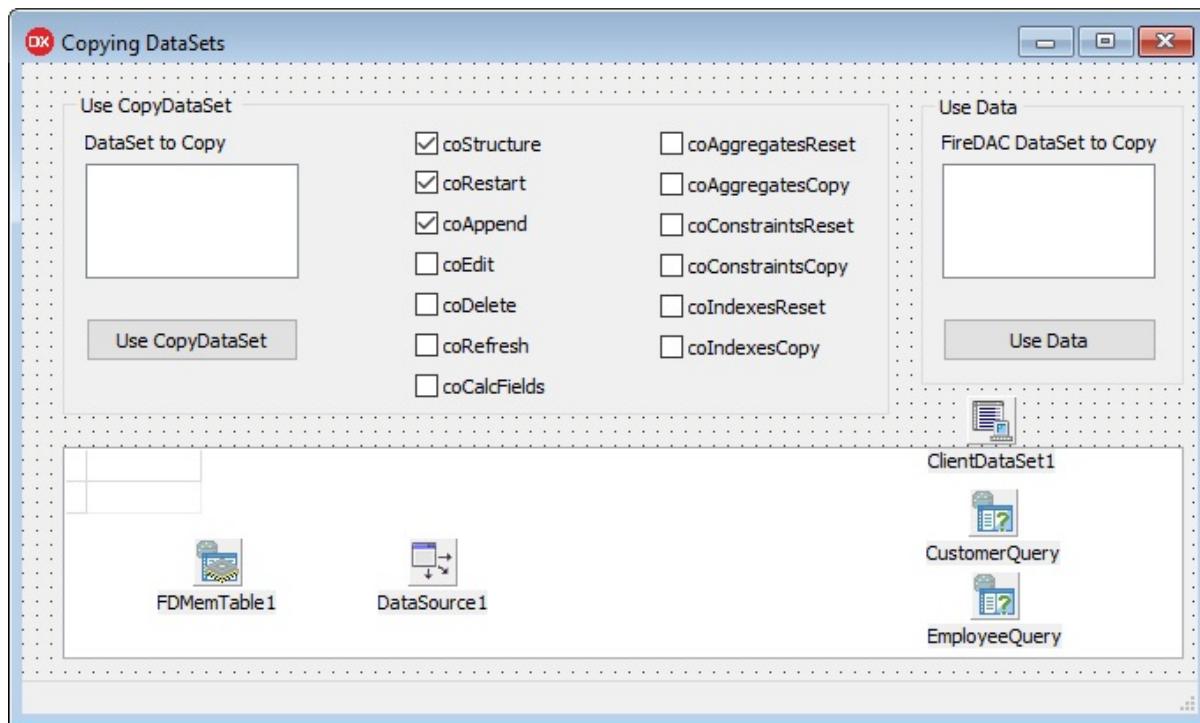
be compatible.

With the coRestart flag, the source dataset cursor will be moved to the first record before the operation begins. If this flag is absent, the records that will be processed from the source dataset will begin with the current position of the source dataset cursor.

When the coAppend flag is present in the set, as it is in the default value of the AOptions parameter, each record from the source table that is processed is added to the destination table, again based on field names with necessary data type compatibility.

If you are calling CopyDataSet from an FDMemTable that does not have structure, there is an additional flag that you will need to include, and that is coStructure. When coStructure is included in AOptions, the destination dataset will get its field definitions from the source dataset.

The basic use of CopyDataSet is demonstrated using the FDCopyingDataSets project, whose main form is shown in Delphi's designer in Figure 12-8.



Chapter 12: Understanding FDMemTables 351

**Figure 12-8: The FDCopyingDataSets project in Delphi's designer**

*Code: You can find the FDCopyingDataSets project in the code download.*

The FDMemTable on the left side of Figure 12-8 is the dataset whose CopyDataSet method will be called, and possible source datasets (a

ClientDataSet and two FDQueries) appear on the right side of this figure. You initiate a call to CopyDataSet by first selecting one of the datasets displayed in the DataSet to Copy list box which appears in the upper left area of the form, after which you select which flags you want to include in the AOptions parameter from the provided checkboxes, and then clicking the button labeled Use CopyDataSet. (At this point, ignore the Use Data group box on the right side of this figure. I will discuss this option later in this chapter.)

Since the checkboxes, and the construction of the AOptions parameter, are instrumental to the use of CopyDataSet, let's take a moment to discuss it. The AOptions parameter is built through a call to the BuildCopyDataSetOptions, which is a custom method that appears in each of the projects referred to in this section. BuildCopyDataSetOptions is shown here:

352 Delphi in Depth: FireDAC

```
function TForm1.BuildCopyDataSetOptions: TFDCopyDataSetOptions;  
begin  
  result := [];  
  if cbxStructure.Checked then Result := Result + [coStructure]; if  
  cbxRestart.Checked then Result := Result + [coRestart];  
  if cbxAppend.Checked then Result := Result + [coAppend];  
  if cbxEdit.Checked then Result := Result + [coEdit];  
{$IF CompilerVersion > 27.0} // Added in Delphi XE7  
  if cbxDelete.Checked then Result := Result + [coDelete];  
{$ENDIF}  
  if cbxRefresh.Checked then Result := Result + [coRefresh];  
  if cbxCalcFields.Checked then Result :=  
    Result + [coCalcFields];  
  if cbxAggregatesReset.Checked then Result :=  
    Result + [coAggregatesReset];  
  if cbxAggregatesCopy.Checked then Result :=  
    Result + [coAggregatesCopy];  
  if cbxConstraintsReset.Checked then  
    Result := Result + [coConstraintsReset];
```

```

if cbxConstraintsCopy.Checked then Result :=  

  Result + [coConstraintsCopy];  

if cbxIndexesReset.Checked then Result :=  

  Result + [coIndexesReset];  

if cbxIndexesCopy.Checked then Result :=  

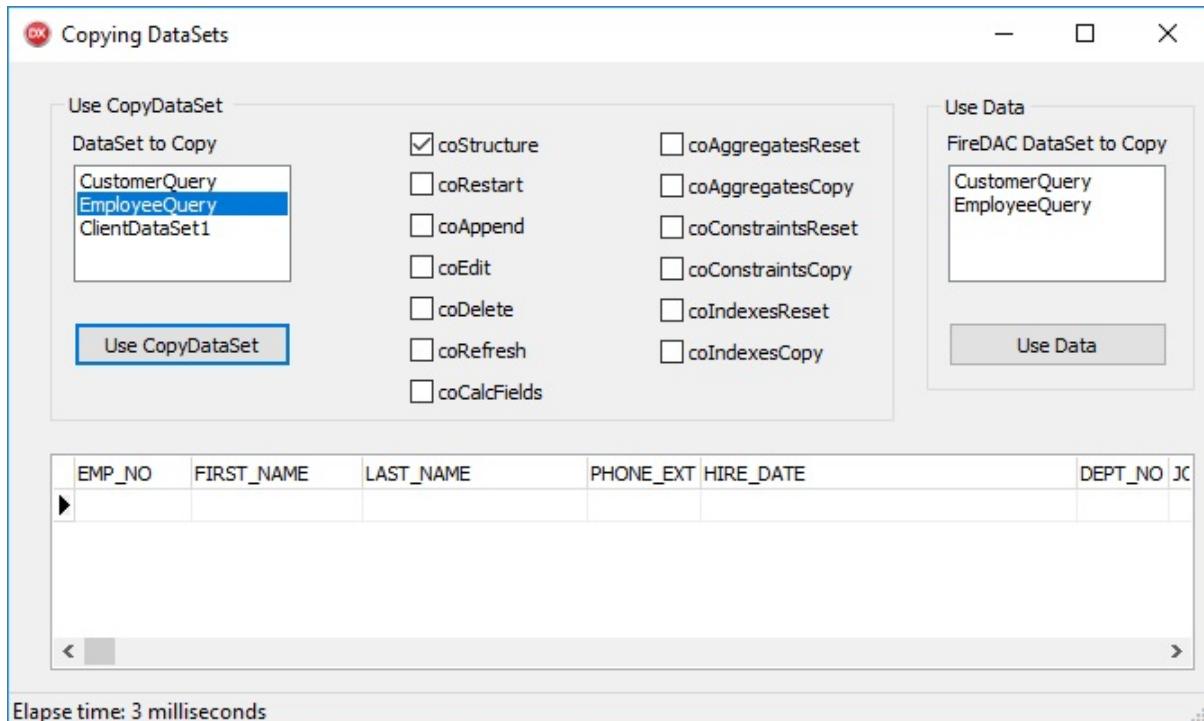
  Result + [coIndexesCopy];  

end;

```

Figure 12-9 shows how this main form looks after calling CopyDataSet with just the coStructure flag in the AOptions parameter when the EmployeeQuery FDQuery is used as the source dataset. For reference, the SQL statement associated with EmployeeQuery is shown here:

```
SELECT * FROM Employee
```



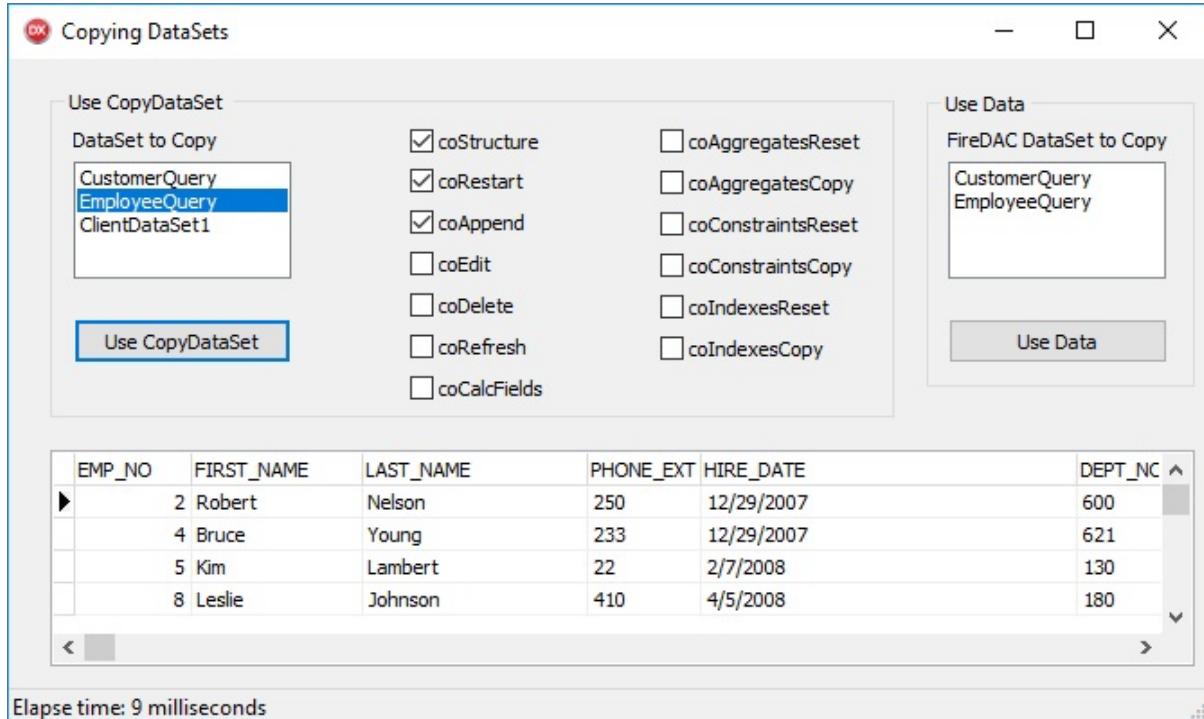
Chapter 12: Understanding FDMemTables 353

**Figure 12-9: CopyDataSet has been called, copying only the structure of a query result**

Though we've constructed the call to CopyDataSet at runtime with the help of the BuildCopyDataSetOptions method, the equivalent call to CopyDataSet that would produce the result shown in Figure 12-9 looks like this:

```
FDMemTable1.CopyDataSet( EmployeeQuery, [coStructure] );
```

By comparison, when coRestart and coAppend are also included in AOptions, both structure and data are included, as shown in Figure 12-10.



### 354 Delphi in Depth: FireDAC

**Figure 12-10: CopyDataSet has been called, copying the structure and data** The equivalent call to CopyDataSet that would produce the results shown in Figure 12-10 is shown here:

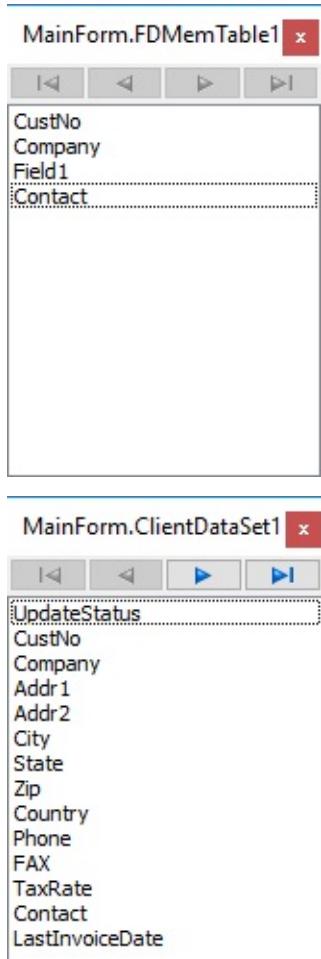
```
FDMemTable1.CopyDataSet( EmployeeQuery, [coStructure,
coRestart, coAppend]);
```

If the FDDataset whose CopyDataSet method you invoke already has structure, you omit the coStructure flag in AOptions. In that case, only the fields from the source dataset whose names match fields in the destination dataset are copied.

This is demonstrated in the FDCopyClientDataSet project.

*Code: You can find the FDCopyClientDataSet project in the code download.*

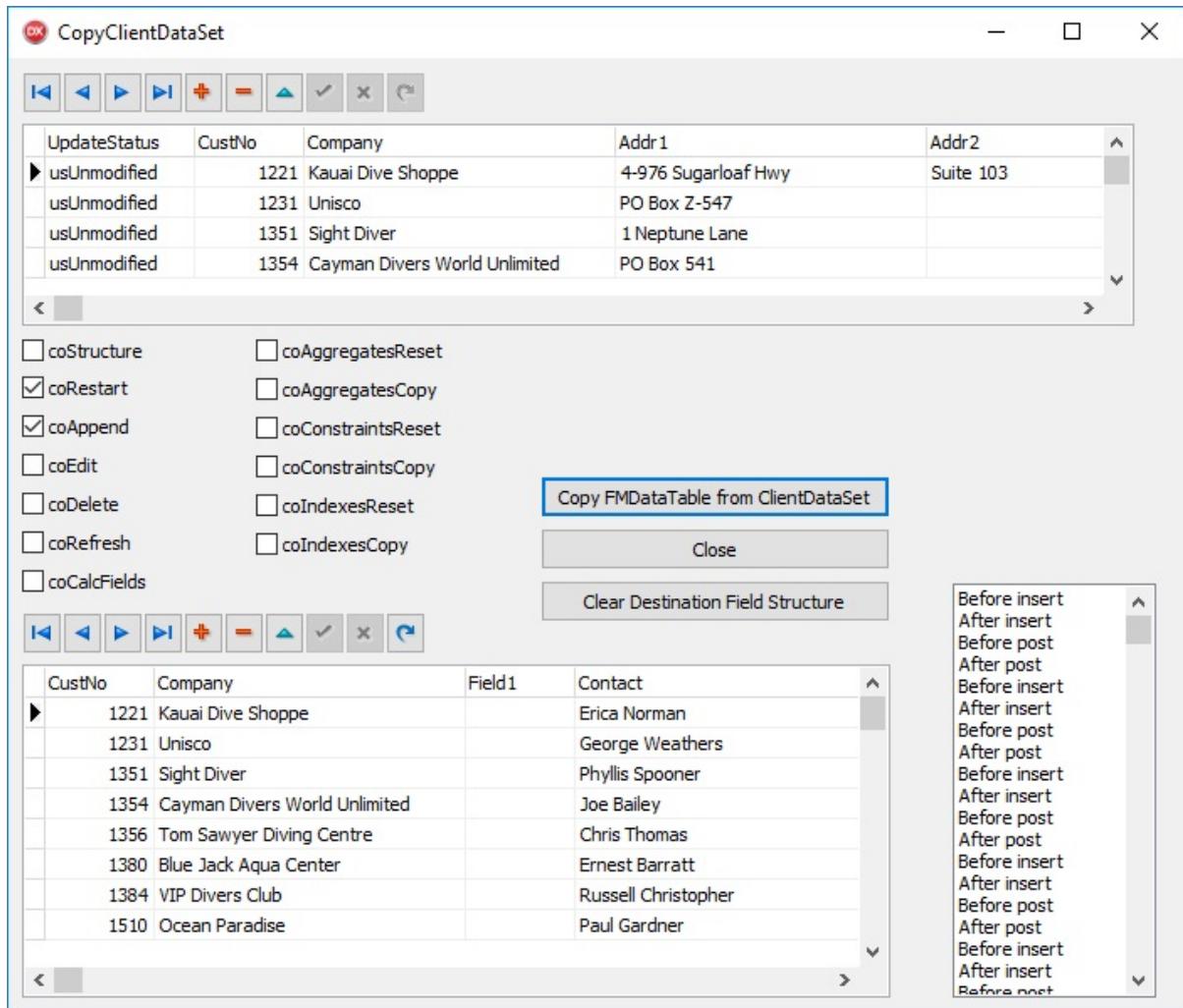
In this project, the destination dataset, an FDMemTable, has a predefined structure that includes four fields, as shown here:



## Chapter 12: Understanding FDMemTables 355

Three of these fields match fields from the source dataset, which is a ClientDataSet that points to the customer.cds table provided for in Delphi's sample files. These fields, including a calculated field named UpdateStatus, are shown here:

Figure 12-11 depicts the result when coStructure is omitted from the destination FDMemTable. In this case, only the three fields whose names and types match those currently defined for the destination dataset appear in the destination dataset. By comparison, if coStructure is included in the AOptions parameter, many more fields will appear in the result dataset (which can be seen in Figure 12-12).



## 356 Delphi in Depth: FireDAC

**Figure 12-11: The structure is not copied, and compatible field data is appended**

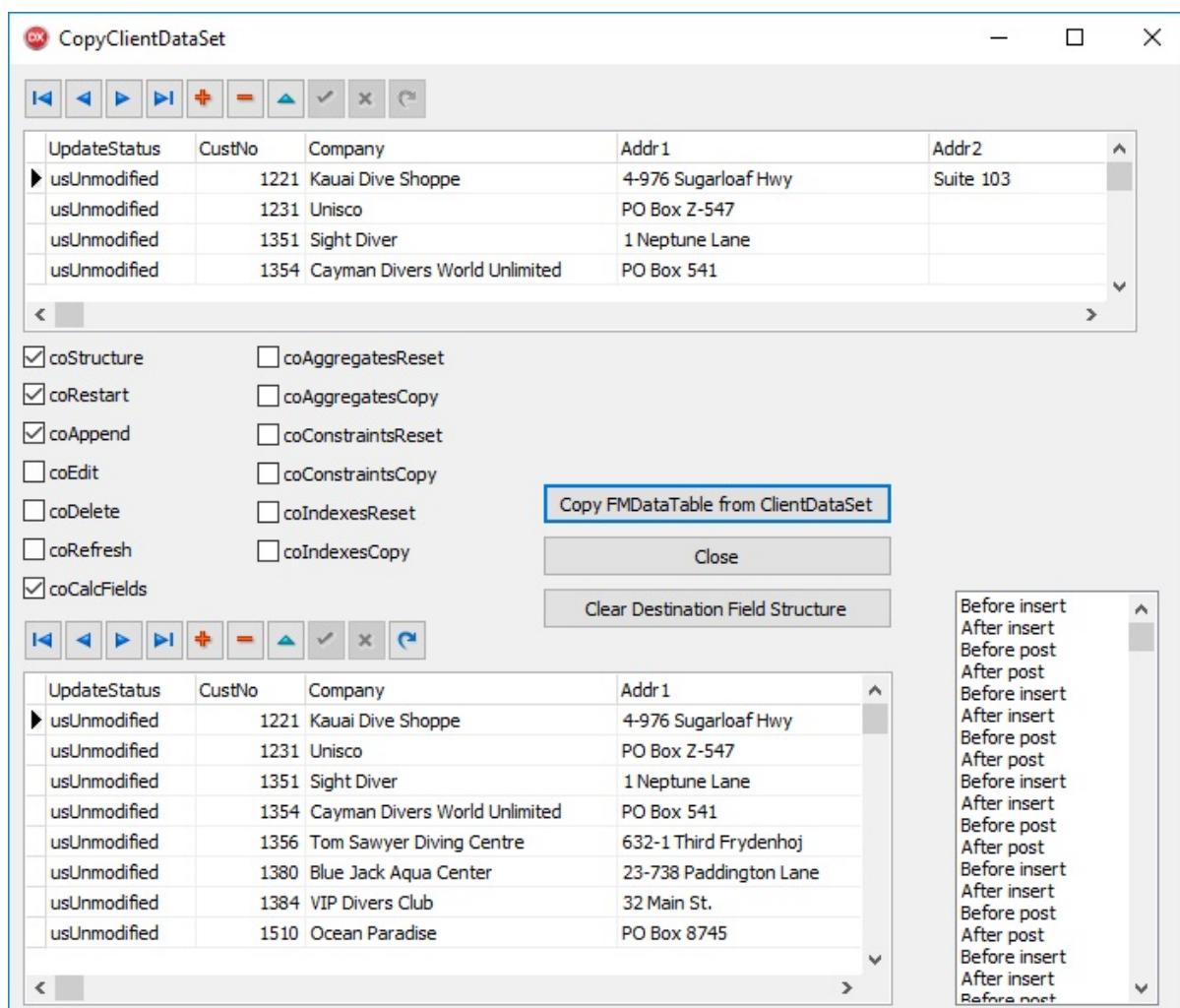
In the FDCopyClientDataSet project, I included event handlers on the destination FDMemTable. The results of these can be seen in the list box that appears on the right side of Figure 12-11. These event handlers demonstrate that CopyDataSet triggers the insert and post events on the destination dataset, and this happens anytime a CopyDataSet call results in the transfer of data from the source dataset to the destination FDMemTable, which, as you have already

learned, happens anytime the coAppend flag is included in AOptions. Most of the remaining, but not all, of the AOptions flags are pretty straightforward. If the source dataset is in cached updates mode, and there are changes in the cache, and you are using coAppend to add data, the coRefresh

flag determines whether the destination records will maintain the status of the corresponding records in the source dataset.

For example, if a record was updated in the source dataset, meaning that its UpdateStatus property is usModified, the destination record UpdateStatus will also be usModified, so long as the coRefresh flag is absent from AOptions. If coRefresh is present, the destination dataset record's UpdateStatus state will be usUnmodified.

If the coCalcFields flag is present in the AOptions parameter, the values of calculated fields will be included in the CopyDataSet operation. In Figure 12-12, both the coStructure and the coCalcFields flags are present in the AOptions parameter. As a result, all fields, including the one calculated field, appear in the destination FDMemTable. Note that the button labeled Clear Destination Field Structure needed to be clicked prior to calling CopyDataSet with the coStructure flag, since coStructure cannot be used on an open destination FDDataSet.



## **Figure 12-12: All fields, including calculated fields, are copied to the destination FDMemTable**

The flags associated with aggregates, constraints, and indexes all work in a similar fashion. Each of these flags are part of a pair. For example for aggregates, there are coAggregatesCopy and coAggregatesReset flags. When the coAggregatesCopy flag is in the AOptions parameter, any Aggregate fields defined in the source dataset will be copied to the destination FDDataSet. By including the coAggregatesReset flag, any existing Aggregate fields in the destination FDDataSet will first be removed before source Aggregate fields are added. The constraint and index related pairs work in the same fashion.

Aggregate fields are discussed in *Chapter 10, Creating and Using Virtual Fields*.

Chapter 12: Understanding FDMemTables 359

This brings us to the two remaining flags, coEdit and coDelete. These AOptions flags are designed to work in situations where the destination dataset already contains data. According to Delphi's documentation, if the coEdit flag appears in AOptions, FireDAC will attempt to locate each source dataset record in the destination FDDataSet using a primary key, and, if found, will update the

destination record if the records do not match. Similarly, if coDelete appears in AOptions, FireDAC will attempt to locate records marked for deletion in the source (this assumes the source is in the cached updates mode), and will delete them from the destination table.

*Note: At the time of this writing (Delphi 10.2 Tokyo has just been released), I could not confirm the preceding descriptions of the coEdit and coDelete flags.*

*In my tests it appeared that these flags had no effect. I reported this results to the RAD Studio team, and they confirmed that there are issues. My understanding is that these issues will be addressed in the update 1 for RAD Studio 10.2 Tokyo.*

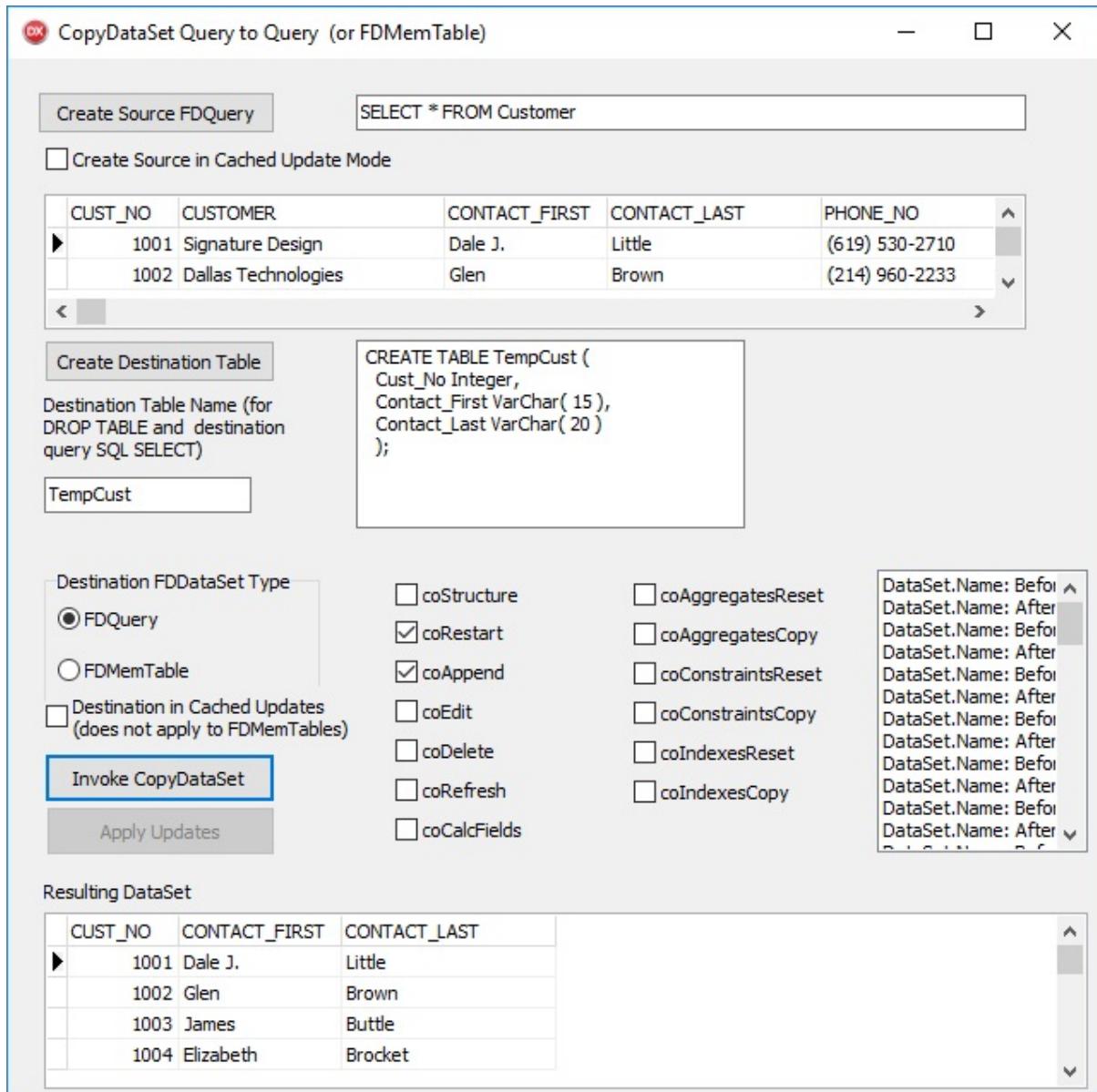
## **OTHER FDdatasets AND COPYDATASET**

While I've focused my attention on using CopyDataSet with FDMemTables as

the destination datasets, this method is available for all FDDataSets, including FDQueries and FDTables.

When CopyDataSet is called from an FDDataSet other than an FDMemTable, and the FDDataSet is not in cached updates mode, the inserts made to the destination dataset are immediately applied to the underlying database on a record by record basis.

Using CopyDataSet with an FDQuery is demonstrated using the CopyDataSetQuery2Query project. In Figure 12-13, a dataset containing a SELECT \* FROM Customer query has been invoked, and a new table named TempCust has been created in the database. The destination FDQuery contains a SELECT \* FROM TempCust query, which, if executed, would return an empty dataset (since this table has only just been created in the database). When the FDQuery executes the CopyDataSet method, using coRestart and coAppend, the inserted records are immediately inserted into the underlying TempCust table.



## 360 Delphi in Depth: FireDAC

**Figure 12-13: The TempCust table in the database is initially empty before CopyDataSet inserts records into it**

*Code: The CopyDataSetQuery2Query project is in the code download.*

Because the CopyDataSet method inserts records into the FDQuery object whose query references the SELECT \* FROM TempCust query, the inserted records are immediately posted to the underlying database table, as shown in Figure 12-14.

The screenshot shows a database properties window for a table named 'TEMPCUST'. The window has tabs for Properties, Metadata, Permissions, Data, and Dependencies. The Data tab is selected, displaying a grid of data with three columns: CUST\_NO, CONTACT\_FIRST, and CONTACT\_LAST. The data consists of 15 rows, each containing a customer number and their first and last names. The first few rows are: 1001 Dale J. Little, 1002 Glen Brown, 1003 James Buttle, 1004 Elizabeth Brocket, 1005 Tai Wu, 1006 Tomas Bright, 1007 <null> Mrs. Beauvais, 1008 Leilani Briggs, 1009 Max <null>, 1010 Miwako Miyamoto, 1011 Victor Granges, 1012 Michelle Roche, 1013 Andreas Lorenzi, 1014 Greta Hessels, 1015 K.M. Neppelenbroek. Below the grid is a toolbar with various icons for database operations like insert, update, delete, and refresh. The status bar at the bottom shows the path: ...\\Public\\Documents\\Embarcadero\\Studio\\19.0\\Samples\\Data\\EMPLOYEE\\Tables.

CUST_NO	CONTACT_FIRST	CONTACT_LAST
1001	Dale J.	Little
1002	Glen	Brown
1003	James	Buttle
1004	Elizabeth	Brocket
1005	Tai	Wu
1006	Tomas	Bright
1007	<null>	Mrs. Beauvais
1008	Leilani	Briggs
1009	Max	<null>
1010	Miwako	Miyamoto
1011	Victor	Granges
1012	Michelle	Roche
1013	Andreas	Lorenzi
1014	Greta	Hessels
1015	K.M.	Neppelenbroek

## Chapter 12: Understanding FDMemTables 361

### Figure 12-14: Newly inserted data resulting from a CopyDataSet invocation

By comparison, if the destination FDDataset is in cached updates mode, and the coRefresh flag is not in AOptions, the copied data contains all of the

information necessary to apply those updates to the underlying database, and will attempt to do so if the ApplyUpdates method is subsequently called for the destination FDDataset. This is demonstrated using the

CopyDataSetQuery2Query project. To demonstrate this, use the following steps:

1. Create the source query in a cached updates mode.

2. Next, create a suitable destination table in the underlying database by clicking the Create Destination Table button (using the provided SQL, or something similar to it).

3. Set Destination Table Type to FDQuery.

4. Place a check mark in the Destination in Cached Updates checkbox.

362 Delphi in Depth: FireDAC

5. Click Invoke CopyDataSet.

6. Finally, click Apply Updates. It is at this point that the records copied to the destination FDQuery are inserted into the database table referenced by the Destination Table Name field.

So, we've established that we can use non-FDMemTable FDDatasets as the destination dataset for the CopyDataSet method. And while this technique may sound appealing, there are better alternatives in most cases.

Consider this — using CopyDataSet with a non-FDMemTable destination FDDataset is designed to update an underlying database table. There is, however, a much better solution for updating underlying database tables.

Specifically, a FireDAC feature called *Array DML*.

*Array DML* permits you to execute a series of parameterized INSERT, UPDATE, or DELETE queries. Importantly, this mechanism can leverage the batch processing capability available in many different remote database servers, and FireDAC will emulate these batch operations in databases that do not

support batch processing. With *Array DML*, a parameterized query, along with an array of query parameters, are passed to the server in a single operation. The results are database updates that are blindingly fast, providing performance far superior to that possible using CopyDataSet, which necessarily applies its updates on a record-by-record basis. *Array DML* is discussed in detail in

*Chapter 15, Array DML*.

## **Copying Data Using the FDDataset Data Property**

If your goal is to use CopyDataSet to initialize an FDMemTable and move data into it from another dataset, there is a second mechanism worth considering —

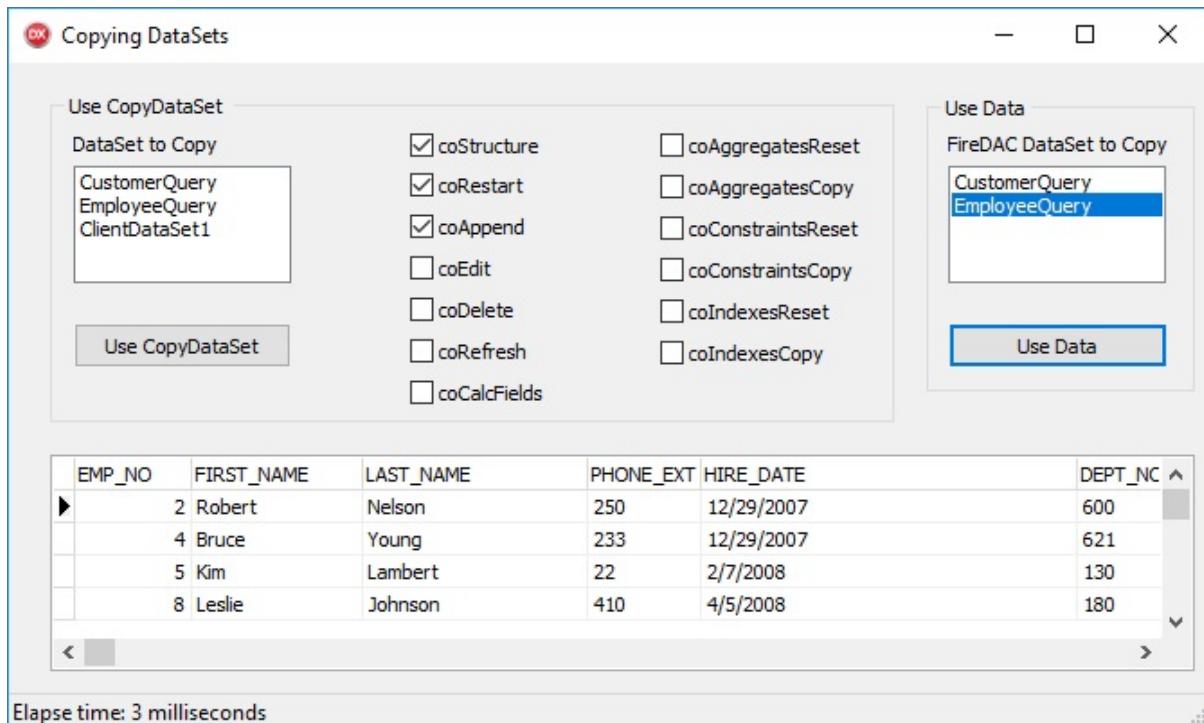
the Data property. The Data property is available on all FireDAC datasets, and you can initialize an FDMemTable with another FDDataset's structure and data simply by assigning the Data property of a source FDDataset to the Data

property of a destination FDMemTable.

This technique is demonstrated in the FDCopyingDatasets project that we used earlier (see Figure 12-10). This project, whose main form is displayed in Figure 12-15, permits you to compare the CopyDataSet method to using the

## Data

property. When this project first loads, all TDataSet descendants that appear on the form (with the exception of FDMemTable1) are listed in the list box on the left side of the form, while only FDDataSets (again, with the exception of FDMemTable1) are listed in the list box on the right. Selecting one of these datasets and clicking the corresponding button (Use CopyDataSet or Use Data) will define the structure of FDMemTable1 and populate it with the data



## Chapter 12: Understanding FDMemTables 363

appearing in the selected dataset (assuming that the coStructure, coRestart, and coAppend checkboxes are checked when using CopyDataSet).

Figure 12-15 shows the FDMemTable1 populated by assigning to it the results of the EmployeeQuery FDQuery as exposed by the Data property. What you see here is similar in almost every way to the results produced by clicking the Use CopyDataSet button, shown back in Figure 12-10.

Please note that the Data property is not configurable in the same way as CopyDataSet. As a result, the operations invoked by clicking the Use Data button ignore the configurations represented by the checkboxes that appear to the left of the button.

**Figure 12-15: FDMemTable1 has been given structure and data by assigning the Data property of EmployeeQuery to the Data property of FDMemTable1**

While the results shown in Figure 12-15 and Figure 12-10 appear to be identical, there are very big differences. The first is that the Data property will always copy both structure and all data. CopyDataSet, on the other hand, can copy only the structure, or it can copy only data into an existing structure.

### 364 Delphi in Depth: FireDAC

The second difference is that CopyDataSet can work with any Delphi TDataSet.

The Data property can only be used to move structure and data from one FDDataset to another FDDataset.

The third difference is that CopyDataSet may trigger insert and post events, on a record-by-record basis. By comparison, copying the Data property from one

FDDataset to another does not trigger insert, update, or delete events at the FDDataset level.

The final difference is that copying data using the Data property is faster than copying using CopyDataSet. Comparing Figure 12-10 to Figure 12-15, at least with this small result set, using the Data property is about three times faster than using the CopyDataSet method, and I suspect that this difference will increase with the amount of data being copied. This speed results from FireDAC not

having to post updates to the destination FDDataset on a record-by-record basis.

*Note: The speed of CopyDataSet versus using Data was obtained using the Start and Complete custom methods introduced in Chapter 6, Navigating and Editing.*

*For a discussion of these methods, please refer back to that chapter.*

While I've emphasized that using the Data property to copy data and structure into a FireDAC dataset requires that the source datasets also be a FireDAC

dataset, this fact is worth mentioning again before I leave this topic. The reason why is that the ClientDataSet component also has a Data property, and this property also represents data and metadata associated with the ClientDataSet.

However, the ClientDataSet.Data property is incompatible with the Data property of FireDAC datasets, which is why a ClientDataSet cannot be the source dataset when copying data to a FireDAC dataset using the Data

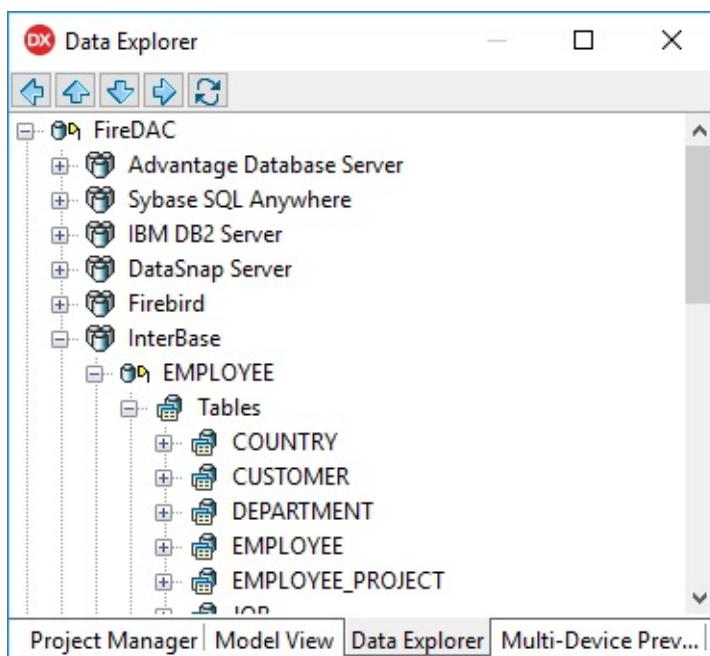
property.

## Editing Data at Design Time

There's one additional thing that an FDMemTable's can do that other FireDAC

datasets cannot, and this feature was added in Delphi 10.2 Tokyo, which is the latest version of Delphi at the time of this writing. The FDMemTable can edit its data at design time.

This feature really only makes sense when you can load data into an FDMemTable at design time, edit that data, and then save that data. And in this case, the only option for saving the data is to write it to a file or save it to the form, frame, or data module's resource file (\*.dfm or \*.fmx). Any other use of



## Chapter 12: Understanding FDMemTables 365

this feature is a waste of time, since if you cannot save the data, editing it is a pointless exercise.

The following steps demonstrate editing data at design time. In this instance, I am going to load data derived from an FDQuery, edit it, and then save it to a file. Use the following steps to demonstrate this new capability.

1. Select File | New | VCL Forms Application to create a new application.
2. Using the Data Explorer, expand the FireDAC node, then expand the InterBase node, then expand the Employee, and finally, expand the

Tables node. Your Data Explorer should look something like this:

3. Select the Customer node and drop it onto your new form. Delphi will respond by creating an FDConnection named EmployeeConnection, and an FDQuery named CustomerTable.
4. Set the Active property of the CustomerTable FDQuery to True.
5. Using the Tool Palette, drop an FDMemTable onto the new form.
6. Right-click the FDMemTable and select Assign DataSet. Select CustomerTable from the displayed dialog box and click OK.

CUST_NO	CUSTOMER	CONTACT_FIRST	CONTACT_LAST
1001	Signature Design	Dale J.	Little
1002	Dallas Technologies	Glen	Brown
1003	Buttle, Griffith and Co.	James	Buttle
1004	Central Bank	Elizabeth	Brocket
1005	DT Systems, LTD.	Tai	Wu
1006	DataServe International	Tomas	Bright
1007	Mrs. Beauvais		Mrs. Beauvais
1008	Anini Vacation Rentals	Leilani	Briggs
1009	Max	Max	

## 366 Delphi in Depth: FireDAC

7. Right-click the FDMemTable once again and select Edit Dataset. Delphi responds by displaying a form on which a DBNavigator, a DBGrid, and several buttons appear, as shown in Figure 12-16.
8. Make your edits to the contents of the FDMemTable, and select OK to save these changes and close the displayed form.
9. Now, you must take one more step or any edits that you made will be lost. You must save your data. This data can either be saved in the form's resource file, or you can save the data to a file. To save the data to a file, right-click the FDMemTable and select Save To File. Use the browser to select a file to save the data to and then select Save.

The alternative is to save the data as a property of the FDMemTable in the form's resource file. To do this, save the form without closing the FDMemTable. This will cause the FDMemTable to stream its properties, including the data, into the form file (\*.dfm or \*.fmx). The next time you open the form, the FDMemTable will load its data (and other published properties) from the form file, and will be populated, assuming that you left the `auDesignTime` flag in the `ActiveStoredUsage` property the last time you saved the form.

**Figure 12-16: Editing an FDMemTable at design time**

Chapter 12: Understanding FDMemTables 367

In the next chapter, I take a look at the unique capabilities that FDMemTables introduce: cloned cursors and nested datasets.

Chapter 13: More FDMemTables 369

# Chapter 13

## More FDMemTables:

### Cloned Cursors and Nested DataSets

There are certain features that are especially well suited for FDMemTables, and two of these are cloned cursors and nested datasets. As mentioned in the preceding chapter, while all FireDAC datasets expose a public `CloneCursor` method, this method is intended for FDMemTable use, and tends to raise exceptions when you attempt to use other FireDAC datasets to invoke this method.

Similarly, nested datasets are a natural feature for FDMemTables. Other FireDAC datasets, such as FDQueries, support nested datasets when the FireDAC driver and the underlying database also support nested datasets, such as Oracle and PostgreSQL, but those situations are the exception. As a result, nested datasets are discussed here in the context of FDMemTables.

*Note: If you find yourself using nested datasets with FDQueries and a compatible database, you can use the techniques described in this chapter, such as using the `NestedDataSet` property of a `DataSetField` instance or the `DataSetField` property of an FDMemTable, to refer to and work with those nested datasets.*

### Cloning and FDMemTables

What do you do when you need two different views of the same data at the same time?

#### 370 Delphi in Depth: FireDAC

One alternative is to load two copies of the data into memory. This approach, however, results in an unnecessary increase in network traffic (or disk access) and places redundant data in memory.

In some cases, a better option is to clone the cursor of an already populated FDDataset. When you clone a cursor, you create a second, independent pointer to an existing FDDataset's memory store, including Delta (the change cache, if cached updates are being employed). Importantly, the cloned

FDDDataSet has an independent view, including, but not limited to, current record, filter, index, and range.

It is difficult to appreciate the power of cloned cursors without actually using them, but an example can help. Imagine that you have loaded 25,000 records into an FDMemTable, and you want to compare two separate records in that FDMemTable programmatically.

One approach is to locate the first record and save some of its data into local variables. You can then locate the second record and compare the saved data to that in the second record.

Yet another approach is to load a second copy of the data in memory. You can then locate the first record in one FDMemTable, the second record in the other FDMemTable, and then directly compare the two records.

A third approach, and one that has advantages over the first two, is to utilize the one copy of data in memory, and clone a second cursor onto this memory store.

The cloned FDMemTable cursor appears as if it were a second copy of the data in memory, in that you now have two cursors (the original and the clone), and each can point to a different record and utilize a different index.

Importantly, only one copy of the data is stored in memory, and the cloned cursor provides a second, independent pointer into it. You can then point the original cursor to one record, the cloned cursor to the other, and directly compare the two records.

The CloneCursor method, like the CopyDataSet method (described in the last chapter), is introduced in the FDDDataSet class, which means that it can be called by any FDDDataSet, not just FDMemTables. For the most part, however, you will not call CloneCursor with any FDDDataSet other than FDMemTables. Trying to

do so often leads to access violations and other issues. When I asked the author of FireDAC about these issues, he explained that CloneCursor was intended for use only from FDMemTable. For this reason, I am including the discussion of CloneCursor here in this FDMemTable chapter.

The following is the declaration of CloneCursor:

Chapter 13: More FDMemTables 371

**procedure** CloneCursor(ASource: TFDDDataSet;

AReset: Boolean = False;

AKeepSettings: Boolean = False); **virtual**;

When you invoke `CloneCursor`, the first argument that you pass is a reference to an active `FDDataset` whose cursor you want to clone. The `AReset` and `AKeepSettings` are used to either keep or discard the original `FDDataset`'s view.

If you pass a value of `False` in both of these parameters, the cloned cursor will adopt the values of the `DetailFields`, `Filter`, `Filtered`, `FilterOptions`, `FilterChanges`, `IndexName`, `IndexFieldNames`, `MasterSource` and `MasterFields`

properties, as well as the `OnFilterRecord` event handler, of the source dataset. If `AReset` is `True`, these properties will assume the default values. If `AReset` is `False` and `AKeepSettings` is `True`, the clone will assume these properties, but they may or may not be entirely valid. For example, the clone may be set to an index that exists on the source but not on the destination.

Once you have cloned the cursor of an existing `FDDataset` to an `FDMemTable`, there are two specific uses. The first, and most general, is to use the clone as an additional, readonly view of the data. This view can have a different current record, sort order (index), range, and filter.

The second use is as an editable cursor into the cloned `FDDataset`'s data store.

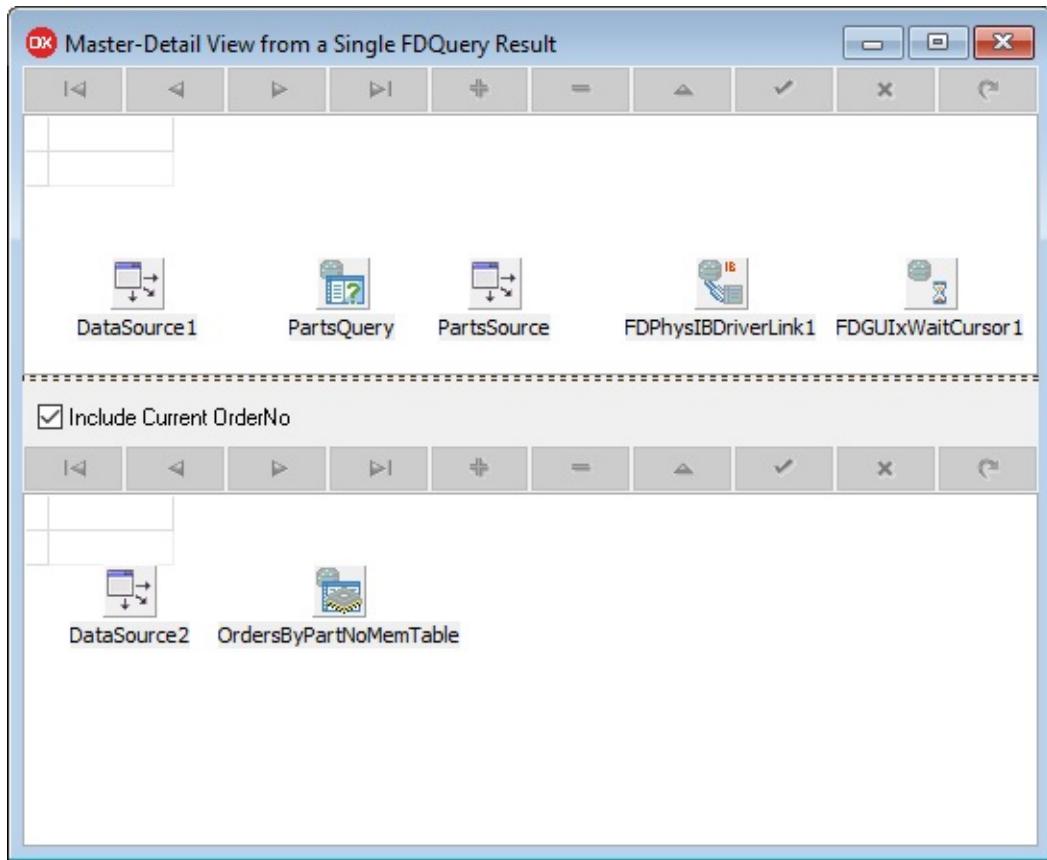
This use, however, has some important limitations. First, the cloned `FDDataset` must be in a cached updates mode. Second, you can only apply those updates by calling the `ApplyUpdates` method of the originally cloned `FDDataset`. Failure to call `ApplyUpdates` on the original dataset will result in the loss of updates applied either by the original `FDDataset` or by any of its clones.

In the following two examples, I demonstrate these two uses for cloned cursors.

The first example demonstrates a read-only clone that provides a master-detail view on a single table. The second example demonstrates an editable clone of an `FDDataset` in cached updates mode.

### **Master with Detail Clone**

This example, one that I really like, is based on a cloned cursor example that I originally included in my two previous `ClientDataSet` books, *Delphi in Depth: ClientDataSets* (first and second editions) . In this example, I use a single query result set to display a self-referencing master-detail relationship. The main form of this project is shown in Figure 13-1.



372 Delphi in Depth: FireDAC

**Figure 13-1: The FDMasterDetailClone project main form**

*Code: The FDMasterDetailClone project can be found in the code download. See Appendix A for more information.*

FDQuery1 holds a simple query of the Items table from the dbdemos.gdb database that is found in Delphi's sample database. Here is the SQL associated with this query:

`SELECT * FROM Items`

The OnCreate event handler of the form modifies the connection found on the SharedDMVcl data module to point to the dbdemos.db database (note that I'm not using the employee.gdb database that is used in most of the code samples in

Chapter 13: More FDMemTables 373

this book). This is the connection that the FDQuery, named PartsQuery, is connected to. Next, the FDMemTable, named OrdersByPartNoMemTable, is used to clone the query, after which OrdersByPartNoMemTable is configured as a detail table of PartsQuery. The final step in this OnCreate event handler is

to set the Filtered property of the FDMemTable to True, which will initially have no effect since the Filter property is blank. Here is that event handler:

```
procedure TForm1.FormCreate(Sender: TObject);  
var  
DBIndex: Integer;  
begin  
//Set the connection to use the dbdemo.gdb database  
DataPaths.ValidatePath( IBDemoPath );  
// Switch database for  
DBIndex :=  
SharedDMVcl.FDConnection.Params.IndexOfName('Database');  
if DBIndex > -1 then  
SharedDMVcl.FDConnection.Params[DBIndex] :=  
'Database=localhost:' + DataPaths.IBDemoPath;  
// Get the Items table records  
PartsQuery.Open;  
//Clone the detail cursor.  
OrdersByPartNoMemTable.CloneCursor(PartsQuery, False);  
//Setup the master-detail relationship  
OrdersByPartNoMemTable.MasterSource := PartsSource;  
OrdersByPartNoMemTable.MasterFields := 'PartNo';  
OrdersByPartNoMemTable.IndexFieldNames := 'PartNo';  
OrdersByPartNoMemTable.IndexesActive := True;  
//The PartsSource.OnDataChange event needs Filtered to be True  
OrdersByPartNoMemTable.Filtered := True;  
end;
```

Figure 13-2 shows the running form. The cloned cursor is shown in the lower DBGrid. Since this cloned cursor has been configured to act as the detail table in a master-detail relationship, the clone only displays those orders where the part number is the same as that associated with the current record in the upper DBGrid.

Master-Detail View from a Single FDQuery Result

The screenshot shows a Delphi application window titled "Master-Detail View from a Single FDQuery Result". It contains two DBGrid components. The top DBGrid has columns: ORDERNO, ITEMNO, PARTNO, QTY, and DISCOUNT. The bottom DBGrid has the same columns. A checkbox labeled "Include Current OrderNo" is checked. The current record in the master grid is highlighted with a blue border. The detail grid shows all records where PARTNO is 12310.

ORDERNO	ITEMNO	PARTNO	QTY	DISCOUNT
1003	1	1313	5	0
1004	1	1313	10	50
1004	2	12310	10	0
1004	3	3316	8	0
1004	4	5324	5	0
1005	1	1320	1	0

ORDERNO	ITEMNO	PARTNO	QTY	DISCOUNT
1004	2	12310	10	0
1074	1	12310	5	0

Include Current OrderNo

## 374 Delphi in Depth: FireDAC

**Figure 13-2: A single in-memory data store is used to display both the master records as well as the detail records**

If you inspect Figure 13-2, you will see that the current record in the master table is associated with PartNo 12310. Furthermore, the detail table is displaying all records in the dataset in which the PartNo is 12310.

The main form also includes a checkbox, whose caption is Include Current OrderNo. In Figure 13-2, you can see that the current record in the upper DBGrid is associated with order number 1004, and the detail records include that order number (as well as order number 1074). If you uncheck the checkbox, the detail dataset will include only the other orders that contain the selected part number in the master table, which in this case, would cause order number 1004

to be omitted from the detail view.

This effect is controlled by the `OnDataChange` event handler of `DataSource1`, whose `DataSet` property points to `PartsQuery`. Each time you navigate to another record in the master table, this event handler triggers to update the `Filter` property of the detail table, as shown in the following code:

Chapter 13: More FDMemTables 375

```
procedure TForm1.DataSource1DataChange(Sender: TObject;
  Field: TField);
```

```
begin
if not IncludeCurrentOrderCbx.Checked then
  OrdersByPartNoMemTable.Filter := 'OrderNo <> ' +
    QuotedStr(PartsQuery.FieldByName('OrderNo').AsString)
else
  OrdersByPartNoMemTable.Filter := '';
end;
```

Clicking the checkbox also results in a change to the Filter property of the cloned dataset. However, in this case, it does so by directly calling the

OnDataChange event handler of DataSource1, as shown in the following code: **procedure TForm1.IncludeCurrentOrderCbxClick(Sender: TObject);**

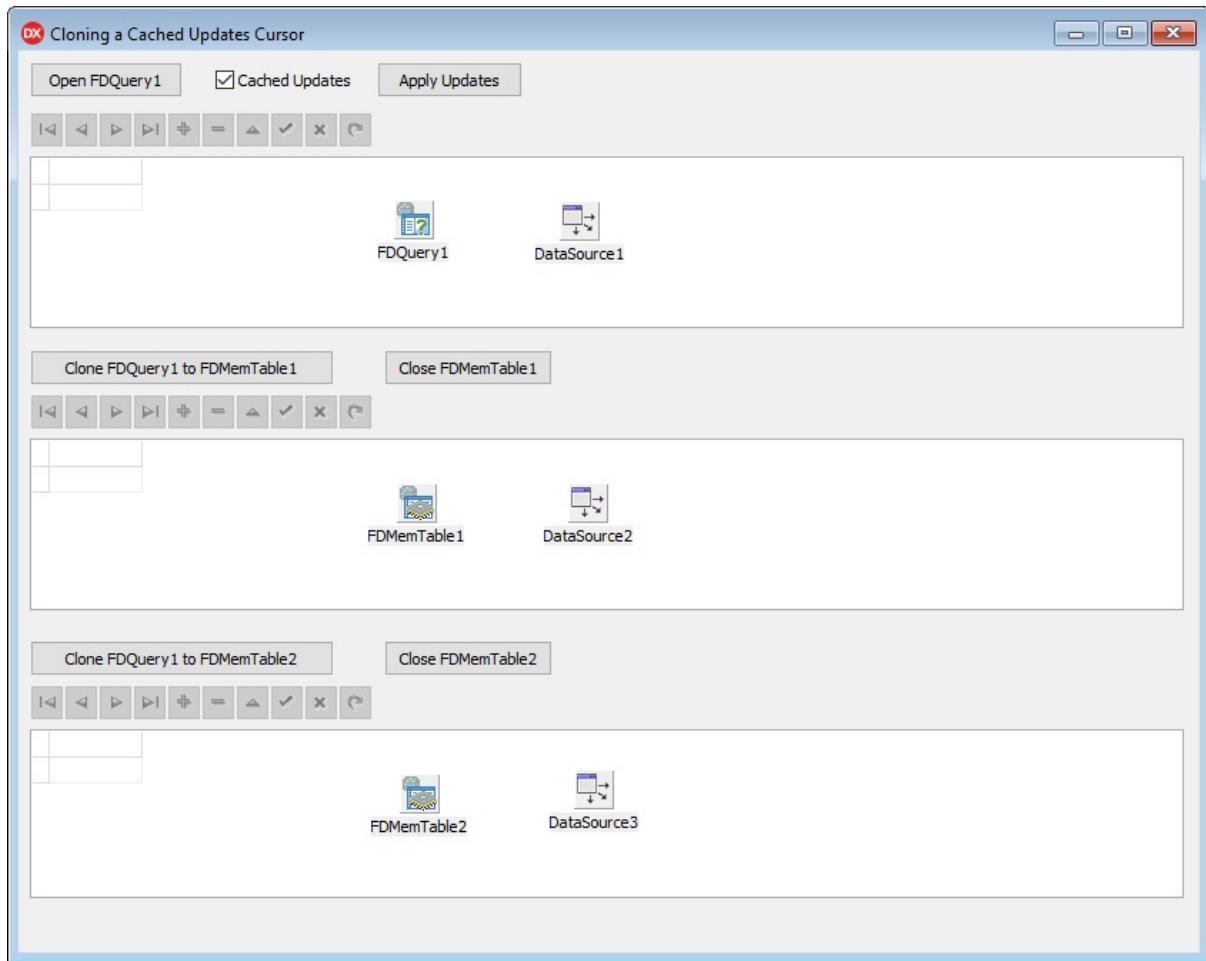
```
begin
  DataSource1DataChange(Self, PartsQuery.Fields[0]);
end;
```

### **Editing with Cloned Cursors and Cached Updates**

Cached updates is a special mode supported by FDDatasets. When in the cached updates state, edits made to an FDDataset are not immediately written to the underlying database. Instead, they are held in memory until that memory is cleared, or until there is an explicit call to ApplyUpdates, at which point the FDDataset will attempt to write all of the changes in cache to the underlying database.

When using cached updates, it is possible to create two or more cloned cursors that can write and manage changes to the cached updates state of the FDDataset that is cloned. This is a powerful capability, and one that suits FDTables very well.

In this section, I am going to demonstrate how to provide two or more editable cursors into a single FDQuery using cloned cursors and cached updates. Cached updates provide you with a rich set of options for editing data, and an entire chapter is devoted to this feature later in this book (*Chapter 16, Using Cached Updates*). For that reason, I am going to limit the discussion of cached updates in this chapter to entering the cached updates state and applying the cached updates to the underlying database, with my primary focus on the cloning of cursors.



## 376 Delphi in Depth: FireDAC

Editing with cloned cursors and cached updates is demonstrated in the project named FDCloningCachedCursors. The main form of this project is shown in Delphi's designer in Figure 13-3.

**Figure 13-3: The main form of the FDCloningCachedCursors project**

*Code: The FDCloningCachedCursors project is included in the code download.*

This project contains one FDQuery and two FDMemTables. The FDQuery contains a simple SELECT \* FROM Employee SQL statement, and is opened by clicking on the button whose default caption is Open FDQuery1. This button, when clicked, will determine whether the query is active or not. If currently closed, the query will be opened and will be placed in the cached updates state, so long as the checkbox labeled Cached Updates is checked. If the query is already active, it is closed. Here is the code associated with the OnClick event of this button:

```

procedure TMainform.btnOpenQuery1Click(Sender: TObject);
begin
if not FDQuery1.Active then
begin
  FDQuery1.Open();
  FDQuery1.CachedUpdates := cbxCachedUpdates.Checked;
  btnApplyUpdates.Enabled := cbxCachedUpdates.Checked;
  btnOpenQuery1.Caption := 'Close FDQuery1';
end
else
begin
  FDQuery1.Close();
  btnApplyUpdates.Enabled := FDMemTable1.Active;
  btnOpenQuery1.Caption := 'Open FDQuery1';
end;
end;

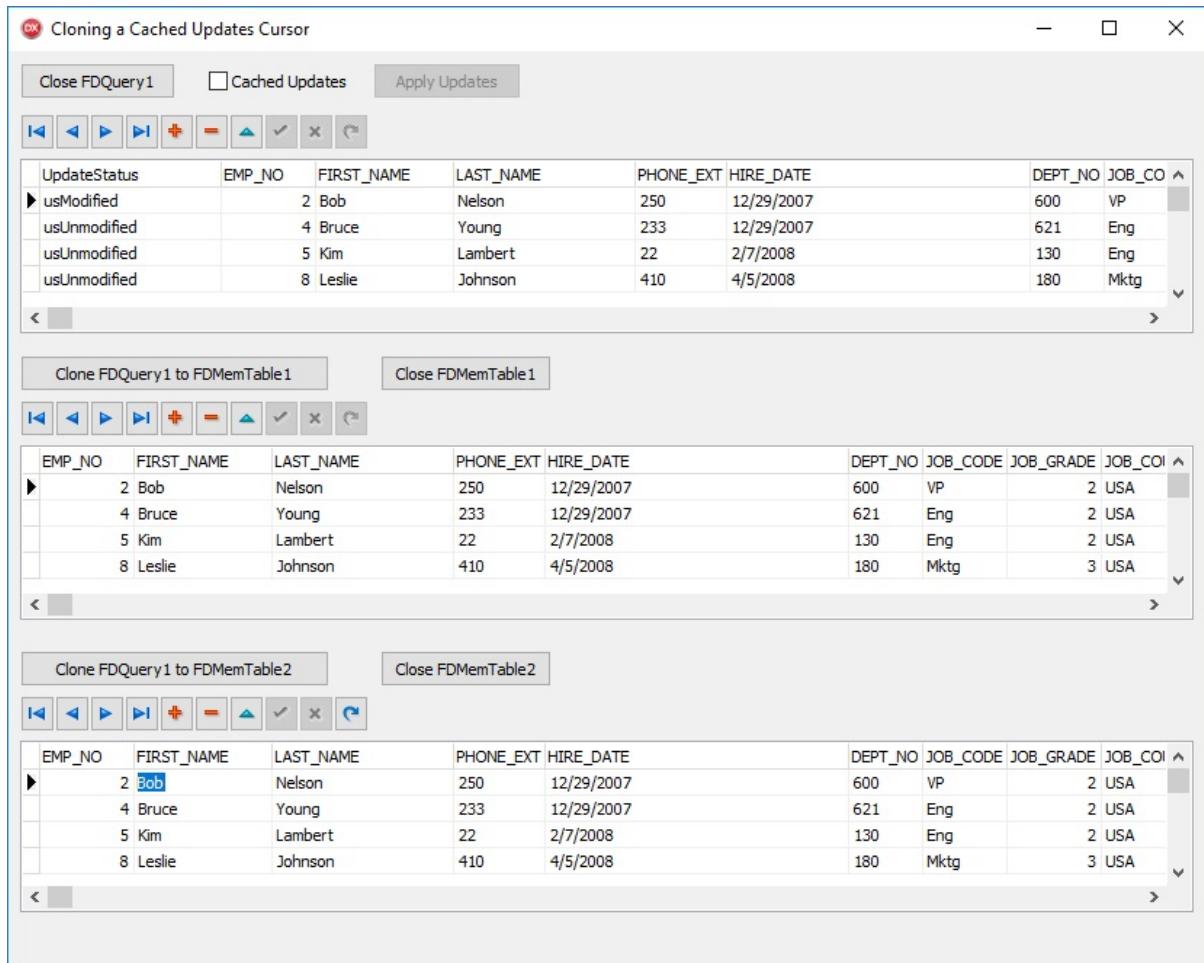
```

Once FDQuery1 has been opened, FDMemTable1 or FDMemTable2 or both can clone the FDQuery1 cursor. At any time, the data originally retrieved through the FDQuery1 SQL statement can be viewed, and even edited, by any of the active datasets.

This can be seen in Figure 13-4. Here FDQuery1 has been opened, and both FDMemTable1 and FDMemTable2 have cloned the FDQuery1 cursor. In Figure

13-4, the lower DBGrid which is associated with FDMemTable2, has been edited, changing the value in the first name field of the first record from Robert to Bob. Once this change has been posted to FDMemTable2, the update can

immediately be seen in all three grids, and the calculated field in FDQuery1, which displays the value of the UpdateStatus property for each record in memory, now shows that this record has been modified.



## 378 Delphi in Depth: FireDAC

**Figure 13-4: FDMemTable2, a cloned cursor of FDQuery1, has been edited.**

**The posted changes are now visible in both FDMemTable1 (another clone)**

**and FDQuery1**

If you take a close look at Figure 13-4, you might notice that the Cached Updates checkbox is not checked. As a result, although FDQuery1 displays the updated data, and the status of the record appears as usModified, the query was not aware that a change has been posted, and so it did not write this change to the underlying database upon posting. Unless FDQuery1 is in cached updates mode when a clone edits the data, the changes made by the clone will not be respected.

This is a bit complicated, so let me try to be as clear as possible. If FDQuery1 is open in cached updates mode, and then its cursor is cloned, changes made by the clone will appear in the change log of FDQuery1, and will be applied when

ApplyUpdates is called on FDQuery1.

## Chapter 13: More FDMemTables 379

If FDQuery1 is not in cached updates mode, edits made by a cloned cursor will appear as though they are in FDQuery1, but will not be written to the database.

Similarly, if FDQuery1 is placed into cached updates mode after a clone has edited the data, the clone's changes made prior to FDQuery1's entry into cached updates mode will not be in the change cache, and therefore will not be seen as an edited record that needs to be applied.

The only way that a clone can update the shared data store is to post an edit to the data while FDQuery1 is in cached updates mode, after which a call to FDQuery1's ApplyUpdates method will attempt to write that change to the underlying database. Only under that condition will FDQuery1's records edited by a clone be marked with the update status information necessary for ApplyUpdates to apply the clone's edits.

Having said that, any changes made directly to FDQuery1, either programmatically or through data binding, when it is not in cached updates mode are immediately written to the underlying database once that change is posted to FDQuery1, whether or not any clones of FDQuery1 exist. And, any changes made directly to FDQuery1 while it is in cached updates mode will reside in the change cache, and a subsequent call to ApplyUpdates will attempt to write that data to the underlying database.

Furthermore, only the original FDDataset can write the cached updates to the underlying database. Calling ApplyUpdates on a cloned cursor will have no effect.

Being able to edit a common data store using one or more cloned cursors and cached updates is a powerful capability, one that can provide the basis for sophisticated features within your applications. However, this is a complicated topic. Playing around with the FDCloningCachedCursors project can be a good way to better understand how this mechanism works.

## **Creating Nested DataSets**

A nested dataset is what it sounds like — a dataset within a dataset. More specifically, a nested dataset is a Field of type DataSetField, and this Field can hold the row-column structure of a dataset.

A given FDMemTable can include one or more nested dataset fields, and each of those may include one or more nested dataset fields. As a result, nested datasets make it possible to represent a complex set of data relations in a single FDMemTable.

## 380 Delphi in Depth: FireDAC

If you've never used nested datasets before, you might not see the value in this type of structure. If that is the case, an example from a real world application should help.

A trucking company has an application that their drivers use to manage the data associated with deliveries. Using the application, a driver receives a single FDMemTable streamed over the Internet to the driver's laptop, tablet, or phone.

This FDMemTable contains all of the information that the driver may need during the course of a scheduled delivery route.

The FDMemTable contains one record for each stop (location) on the driver's route, and includes the name and address of the drop-off location, and other information, including a BLOB containing a PDF of the location map. This record also contains a nested dataset. This dataset includes one record for each of the deliveries to be made at that location. For example, there might be one or more customers at a particular address. The record for each customer includes another nested dataset that contains a list of packages for that customer. The customer-level nested dataset also includes BLOB fields containing PDFs of the invoice, bill of lading, and special handling instructions.

Actually, there are two specific reasons for using nested datasets. The first is that they permit you to use an FDMemTable as a standalone database. In those cases, nested datasets permit you to store master-detail relationships about numerous entities in a single FDMemTable that can be written to, and read

from, a file, or be written to a stream and transmitted over the internet to an endpoint where it is loaded from the stream. In a situation like this, the ability to persist that data is extremely valuable. For a detailed look at FireDAC dataset persistence, please refer to *Chapter 11, Persisting Data*.

The second reason is that it permits you to work with related data from an underlying relational database in a manner that permits the FDMemTable to efficiently manage those relationships. For example, the trucking application gets its data from a traditional database server. However, by moving that data

into an FDMemTable with nested datasets, that same information can be efficiently communicated to the application used by the truck drivers.

## Defining Nested DataSets at Design Time

This section walks you through the creation of nested datasets at design time.

When I began to write this section, I had originally intended to demonstrate the creation of nested datasets using two different techniques, similar to the design-time creation of FDMemTable structures I described at the beginning of

*Chapter 12, Understanding FDMemTables.* Unfortunately, I found that the Chapter 13: More FDMemTables 381

creation of nested datasets using FieldDefs was problematic at design time (something I had also observed in the past). As a result, I am only going to demonstrate the design-time definition of nested datasets using fields.

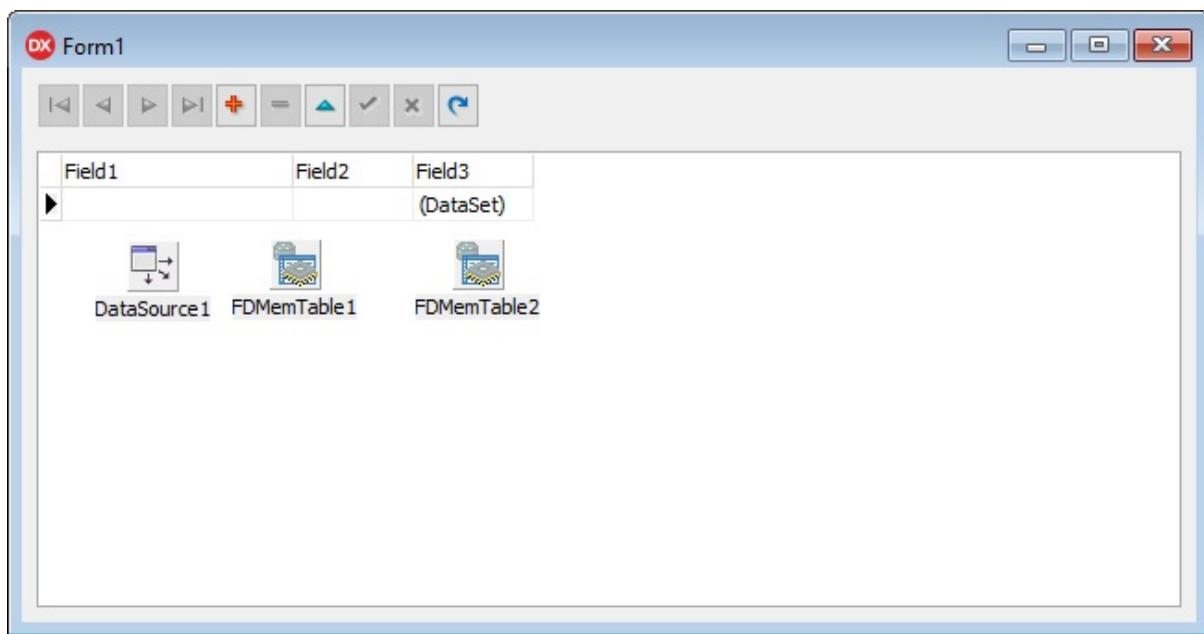
When you define an FDMemTable using fields, you are specifically defining persistent fields, with one data field for each field that the FDMemTable can hold.

*Code: The FDDesigntimeNestedFields project is included in the code download.*

Use the following steps to define an FDMemTable's structure that includes a nested dataset at design time using fields:

1. Create a new VCL Forms Application.
2. Add to the main form a DBNavigator, a DBGrid, a DataSource, and an FDMemTable.
3. Position the DBNavigator in the top left of the form, and position the DBGrid to occupy most of the rest of the form. Also, set the DataSource properties of both the DBNavigator and the DBGrid to DataSource1.
4. Set the DataSource's DataSet property to FDMemTable1.
5. Right-click the FDMemTable and select Fields Editor from the displayed context menu.
6. Right-click the Fields Editor and select New Field.
7. Using the New Field dialog box, set Name to Field1, Type to String, Size to 20, and Field type to Data. Click OK to save the new field.

8. Right-click the Fields Editor and select New Field again.
9. Set Name to Field2, Type to Integer, and Field type to Data. Click OK to save this field.
10. Add one more field. This time set Name to Field3, Type to DataSetField, and Field type to Data. Click OK to continue.
11. At this point, we have our structure for the top-level FDMemTable, but Field3, being a DataSetField, needs to have structure as well. To define the structure for Field3, we need another FDMemTable. Add a second FDMemTable to the main form. This FDMemTable should be named FDMemTable2.



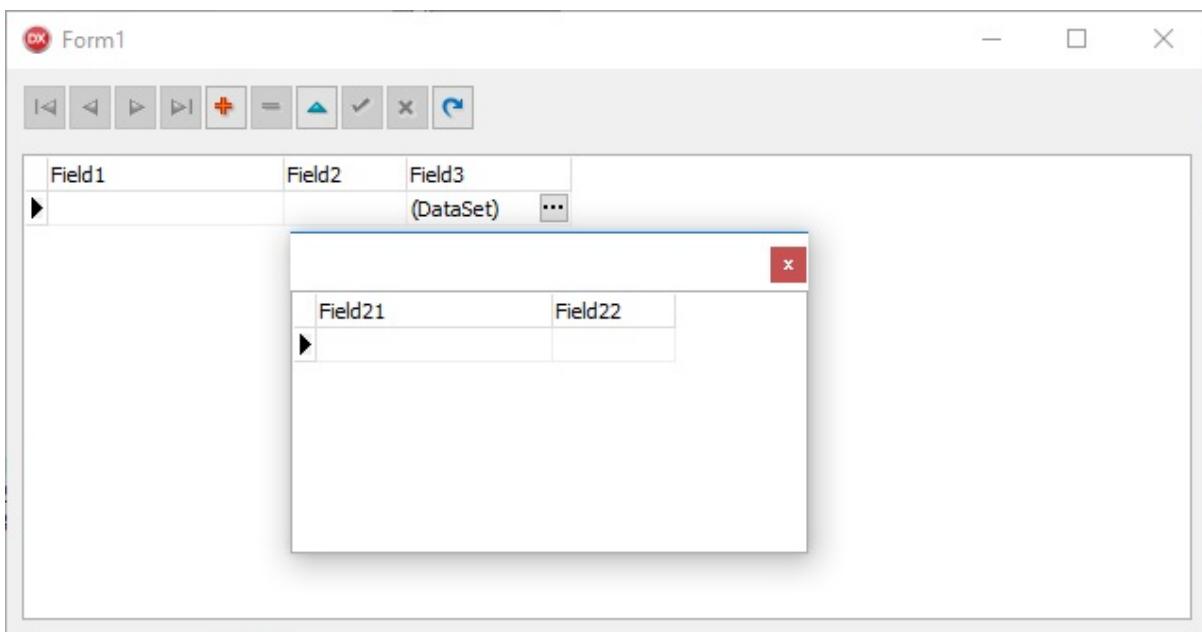
## 382 Delphi in Depth: FireDAC

12. Select FDMemTable2 and set its DataSetField property to FDMemTable1Field3. This is the full name of the persistent field that was created by the designer when you created the nested dataset field.
13. Now, right-click FDMemTable2 and select Fields Editor.
14. Using the Fields Editor, add a new field. Set Name to Field21, Type to String, Size to 20, and Field type to Data. Click OK to continue.
15. Create another new field. Set Name to Field22, Type to DateTime, and Field type to Data. Click OK one last time.
16. You have now completed the structure. Select FDMemTable1 and set its

Active property to True. The FDMemTable structure is created, and the form now looks like that shown in Figure 13-5.

**Figure 13-5: An FDMemTable with a nested dataset has been created using persistent fields**

If you run this project, you will find that you can enter data into the nested dataset by clicking on Field3 to display an ellipsis button. When you click the ellipsis button, an automatically generated form is created, which displays a grid into which you can enter data, as shown in Figure 13-6.



Chapter 13: More FDMemTables 383

**Figure 13-6: Opening a nested dataset field in a DBGrid displays that nested dataset in an automatically created grid**

You don't have control over how the automatically created grid looks, so in most cases, you will provide your own interface to the nested datasets. You do this the same way you do with any other dataset: You associate data-aware controls, or use LiveBindings, to connect a user interface element to the dataset, the dataset being FDMemTable2 in this case.

You can demonstrate this by using the following steps:

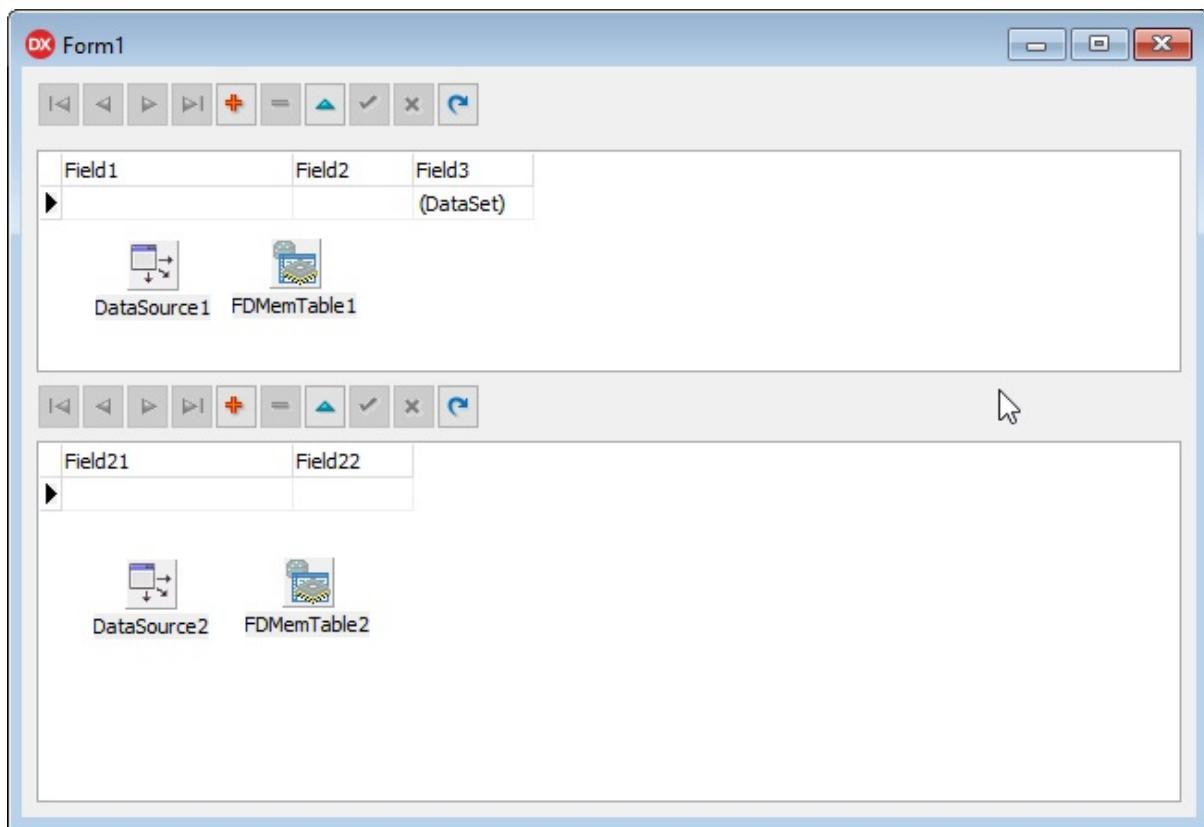
1. Return to the project in Delphi's designer. Adjust the position of DBGrid1, and then place another DBGrid, a DBNavigator, and a DataSource onto the form.
2. Place the second DBNavigator beneath DBGrid1, and position DBGrid2

beneath DBNavigator2.

3. Assign DataSource2 to the DataSource properties of both DBNavigator2 and DBGrid2.

4. Finally, point the DataSet property of DataSource2 to FDMemTable2.

When you are done, your form will look something like that shown in Figure 13-7.



## 384 Delphi in Depth: FireDAC

### **Figure 13-7: The lower grid in this figure is pointing to a nested dataset**

When you run this project, notice that you can enter data into the top grid, as well as into the bottom grid. Importantly, the data entered into the bottom grid will be associated with the nested dataset corresponding to the record appearing in the top grid.

*Note: If you want to suppress the display of the nested dataset field in the top grid (DBGrid1), you can double click the DBGrid to display the Columns editor, click Add All Fields to create one TColumn for each field in the connected dataset, and then select the column associated with the nested dataset and click the Delete Selected button.*

### **Defining Nested DataSets at Runtime**

Creating FDMemTable structures that support nested datasets at runtime can be performed using both FieldDefs and Fields. The trick to writing the runtime code is to mimic exactly the steps that you take when you configure an

## Chapter 13: More FDMemTables 385

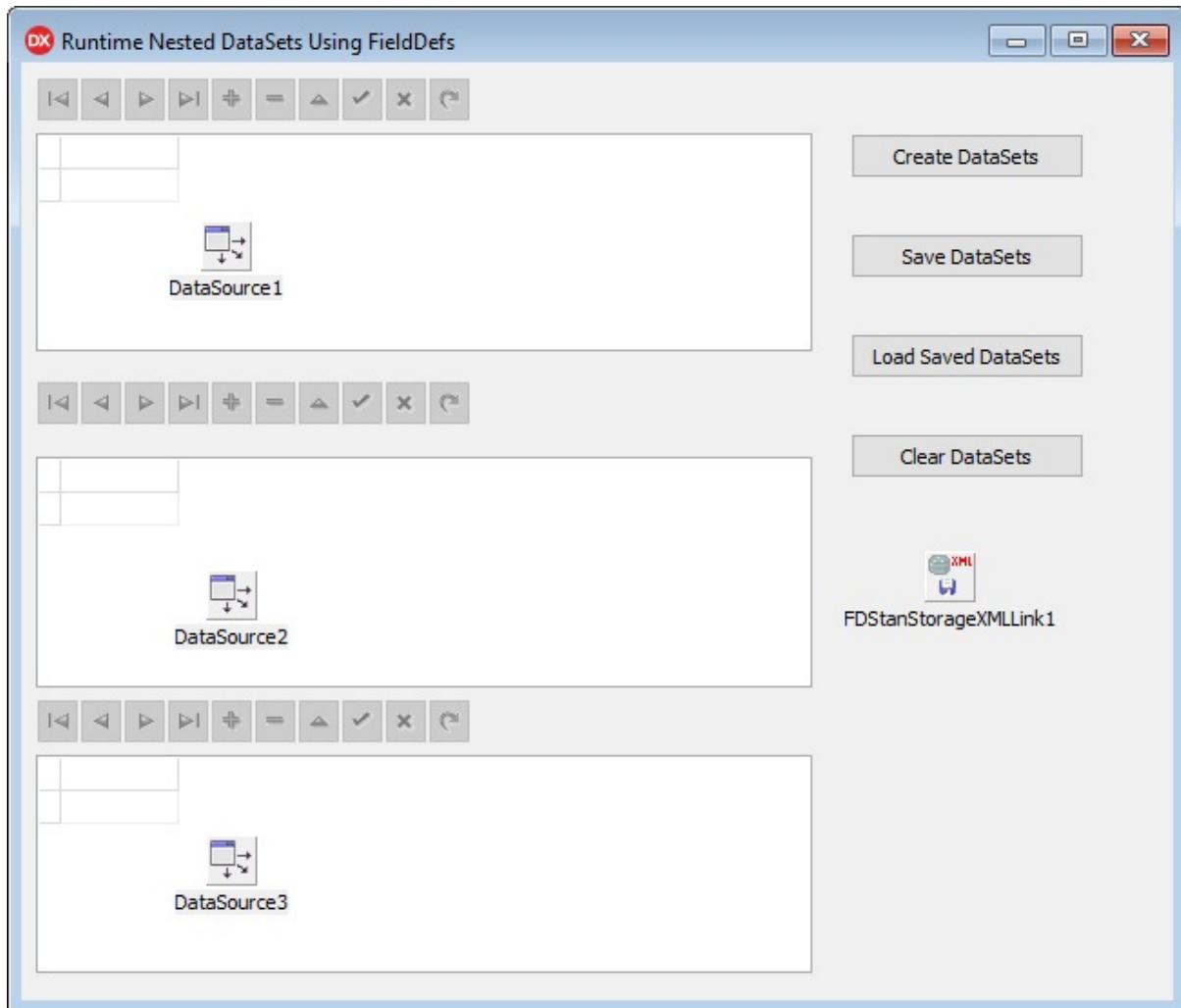
FDMemTable manually, and in the same order. So long as you get that right, your code should work fine.

Unfortunately, writing code that repeats the steps that you take manually is not as easy as it sounds. As a result, the two example projects that I show in the following sections are a bit more complicated than their design-time counterparts, in that they both create a nested, nested dataset — a top-level FDMemTable that contains one nested dataset which in turn contains another nested dataset.

### USING FIELDDEFS AT RUNTIME

Defining an FDMemTable's structure to include nested datasets at runtime using FieldDefs is demonstrated in the FDRuntimeNestedFieldDefs project. The main form of this project is shown in Figure 13-8.

*Code: The FDRuntimeNestedFieldDefs project is in the code download.*



## 386 Delphi in Depth: FireDAC

**Figure 13-8: The main form of the FDRuntimeNestedFieldDefs project**

The actual creation of the FDMemTable structure is performed in a method named CreateNestedDataSets. This method is called from the OnClick event handler of the button labeled Create DataSets. The CreateNestedDataSets method is shown in the following code segment:

```
procedure TForm1.CreateNestedDataSets;
```

```
begin
```

```
  TopLevelMemTable := TFDMemTable.Create(Self);
```

```
  MidLevelMemTable := TFDMemTable.Create(TopLevelMemTable);
```

```
  ThirdLevelMemTable := TFDMemTable.Create(TopLevelMemTable);
```

```
  with TopLevelMemTable.FieldDefs do
```

```
  begin
```

Chapter 13: More FDMemTables 387

```
Add('TopID', ftInteger);
Add('TopName', ftString, 40);
Add('TopComments', ftMemo);
Add('TopDateInitiated', ftDate);
end;
with TopLevelMemTable.FieldDefs.AddFieldDef do
begin
  Name := 'TopNested';
  DataType := ftDataSet;
  with ChildDefs do
    begin
      Add('MidID', ftInteger);
      Add('MidName', ftString, 30);
      with AddChild do
        begin
          Name := 'MidNested';
          DataType := ftDataSet;
          with ChildDefs do
            begin
              Add('ThirdID', ftInteger);
              Add('ThirdName', ftString, 25);
              Add('ThirdActive', ftBoolean);
            end;
            end;
            end;
        end;
    //Create the FDMemTable and its nested datasets
    TopLevelMemTable.Active := True;
```

```

//Hook up the other FDMemTable
MidLevelMemTable.DataSetField :=
TDataSetField(TopLevelMemTable.FieldByName('TopNested'));
ThirdLevelMemTable.DataSetField :=
TDataSetField(MidLevelMemTable.FieldByName('MidNested'));
//Configure the DataSources
DataSource1.DataSet := TopLevelMemTable;
DataSource2.DataSet := MidLevelMemTable;
Datasource3.DataSet := ThirdLevelMemTable;
end;

```

As you can see from this code, after creating the three FDMemTable objects, five FieldDefs are added to the top-level FDMemTable using the FieldDefs Add and AddFieldDef methods.

**Runtime Nested DataSets Using FieldDefs**

The application window displays three nested datasets:

- TopLevelMemTable:**

TopID	TopName	TopComments
1	One	(Memo)
2	Two	(Memo)
3	Three	(Memo)
- MidLevelMemTable:**

MidID	MidName	MidNested
21	TwoOne	(DataSet)
22	TwoTwo	(DataSet)
23	TwoThree	(DATASET)
- ThirdLevelMemTable:**

ThirdID	ThirdName	ThirdActive
31	ThreeOne	
32	ThreeTwo	

Buttons for managing datasets include:

- Create DataSets
- Save DataSets (highlighted)
- Load Saved DataSets
- Clear DataSets

Once the structure of the top-level FDMemTable has been defined, each of the nested dataset structures needs to be defined. After completing all of the necessary FieldDef configurations, the Active property of the top-level FDMemTable is set to True, which creates both the top-level FDMemTables and the nested datasets.

Finally, this code hooks up the second-tier and third-tier FDMemTables to their appropriate DataSetFields. Figure 13-9 shows the main form of the FDRuntimeNestedFieldDefs project at runtime, after the Create DataSets button has been clicked.

**Figure 13-9: A three-tier nested FDMemTable structure has been created at runtime using FieldDefs**

Chapter 13: More FDMemTables 389

Another interesting aspect of this project, one that it shares with the project covered in the next section, is that it permits the three-tier FDMemTable to be saved to disk as well as loaded from a previously saved FDMemTable. When an FDMemTable includes nested datasets, saving the FDMemTable saves the nested metadata, data and the change cache (if cached updates are enabled).

## USING FIELDS AT RUNTIME

Defining an FDMemTable's structure to include nested datasets at runtime using persistent fields is demonstrated in the FDRuntimeNestedFields project. The main form of this project looks exactly like the one shown for the FDRuntimeNestedFieldDefs project shown in Figure 13-9. In fact, the only difference between these two projects is the code found on the CreateNestedDataSets method.

*Code: The FDRuntimeNestedFields project can be found in the code download.*

The following code is from the CreateNestedDataSets method found in the FDRuntimeNestedFields project:

```
procedure TForm1.CreateNestedDataSets;
begin
  TopLevelMemTable := TFDMemTable.Create(Self);
```

```
MidLevelMemTable := TFDMemTable.Create(TopLevelMemTable);
ThirdLevelMemTable := TFDMemTable.Create(TopLevelMemTable);
with TIntegerField.Create(Self) do
begin
  Name := 'TopID';
  FieldKind := fkData;
  FieldName := 'ID';
  DataSet := TopLevelMemTable;
  Required := True;
end;
with TStringField.Create(Self) do
begin
  Name := 'TopName';
  FieldKind := fkData;
  FieldName := 'Name';
  Size := 40;
  DataSet := TopLevelMemTable;
end;
with TMemoField.Create(Self) do
  390 Delphi in Depth: FireDAC
begin
  Name := 'TopComments';
  FieldKind := fkData;
  FieldName := 'Comments';
  DataSet := TopLevelMemTable;
end;
with TDateField.Create(Self) do
begin
  Name := 'TopDateInitiated';
```

```
FieldKind := fkData;
FieldName := 'Date Initiated';
DataSet := TopLevelMemTable;
end;
//Note: For TDataSetFields, FieldKind is fkDataSet by default
with TDataSetField.Create(Self) do
begin
Name := 'TopNested';
FieldName := 'NestedDataSet';
DataSet := TopLevelMemTable;
end;
//MidLevelMemTable
MidLevelMemTable.DataSetField :=
TDataSetField(FindComponent('TopNested'));
with TIntegerField.Create(Self) do
begin
Name := 'MidID';
FieldKind := fkData;
FieldName := 'MidID';
DataSet := MidLevelMemTable;
Required := True;
end;
with TStringField.Create(Self) do
begin
Name := 'MidName';
FieldKind := fkData;
FieldName := 'MidName';
DataSet := MidLevelMemTable;
Size := 30;;
```

```
end;
with TDataSetField.Create(Self) do
begin
  Name := 'MidNested';
  FieldName := 'NestedNestedDataSet';
  Chapter 13: More FDMemTables 391
  DataSet := MidLevelMemTable;
end;
//Third Level
ThirdLevelMemTable.DataSetField :=
  TDataSetField(FindComponent('MidNested'));
with TIntegerField.Create(Self) do
begin
  Name := 'ThirdID';
  FieldKind := fkData;
  FieldName := 'ThirdID';
  DataSet := ThirdLevelMemTable;
  Required := True;
end;
with TStringField.Create(Self) do
begin
  Name := 'ThirdName';
  FieldKind := fkData;
  FieldName := 'ThirdName';
  DataSet := ThirdLevelMemTable;
  Size := 25;
end;
with TBooleanField.Create(Self) do
begin
```

```

Name := 'ThirdActive';
FieldKind := fkData;
FieldName := 'ThirdActive';
DataSet := ThirdLevelMemTable;
end;

//Create the FDMemTable and its nested datasets
TopLevelMemTable.Active := True;
//Configure the DataSources
DataSource1.DataSet := TopLevelMemTable;
(**)
DataSource2.DataSet := MidLevelMemTable;
Datasource3.DataSet := ThirdLevelMemTable;
(**)

(* This is an alternative way of doing the above
DataSource2.DataSet :=
DataSetField(FindComponent('TopNested')).NestedDataSet;
Datasource3.DataSet :=
TDataSetField(FindComponent('MidNested')).NestedDataSet;
(**)
end;

```

## 392 Delphi in Depth: FireDAC

There are two rather slight differences between this code and the corresponding method in the FDRuntimeNestedFieldDefs project. One is that the DataSetField properties of the MidLevelMemTable and ThirdLevelMemTable components

are assigned prior to defining the structure of those FDMemTables. This step is required earlier in the code since each persistent field that you create must specifically be assigned to an FDMemTable, and that must be a valid

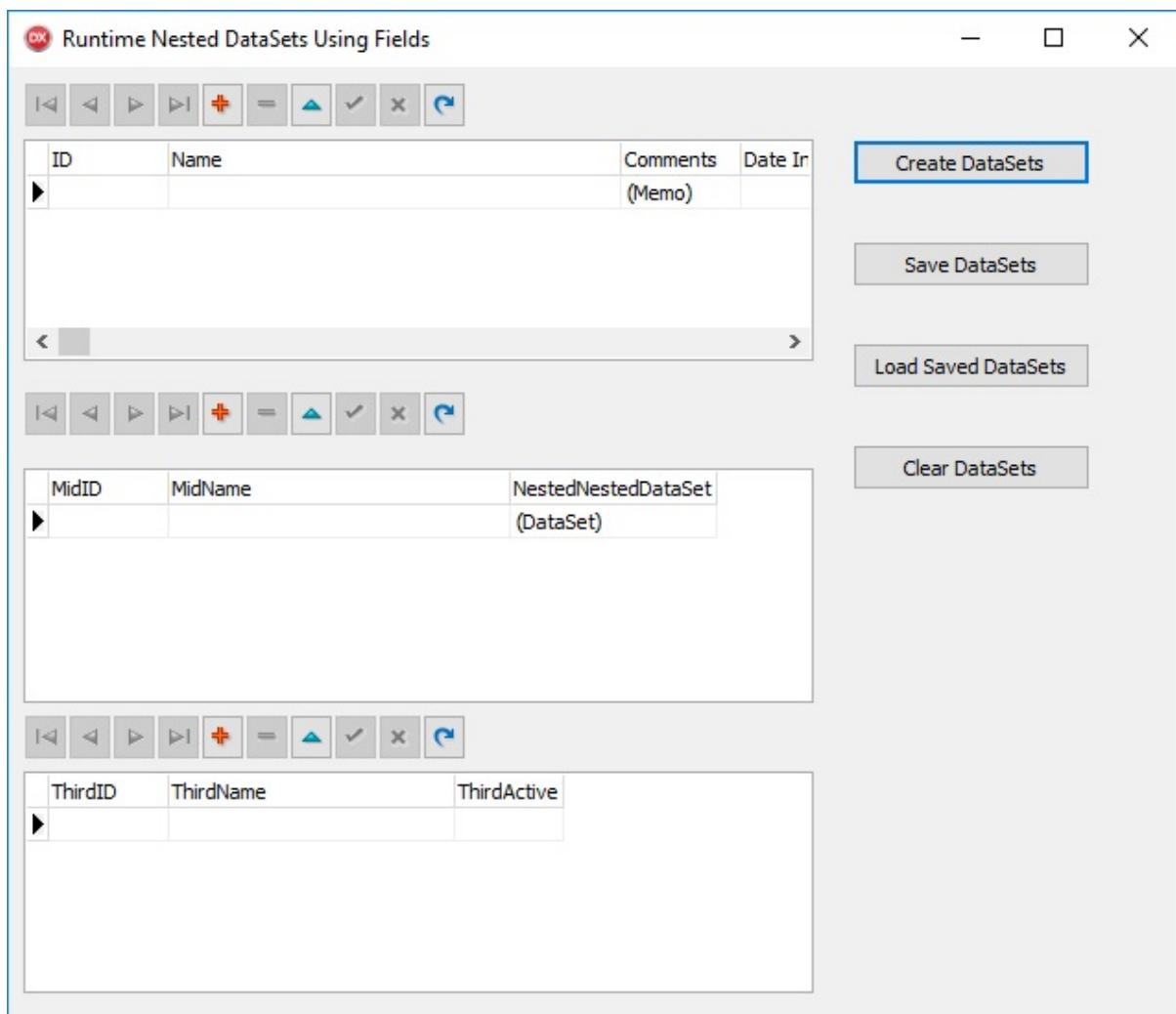
FDMemTable at the time that the Active property of the top-level FDMemTable is set to True.

The second difference is that this code calls the constructors of TField

descendants, such as `TIntegerField`, `TStringField`, and `TDataSetField`. The use of those constructors is more verbose than the `Add` methods associated with `FieldDefs` and `ChildDefs`. As a result, this version of the method is much longer than the previous one.

Figure 13-10 shows what this project looks like when it is running and the Create button has been clicked. As you can see, there is no real discernable difference between what you see in this figure and what is shown in Figure 13-9

(other than no data has yet been added to the datasets in Figure 13-10).



Chapter 13: More FDMemTables 393

**Figure 13-10: A three-tier nested FDMemTable structure has been created at runtime using persistent Fields**

### Final Thoughts About Nested DataSets

Before continuing on to the next chapter, I want to make a couple of observations about nested datasets. The first is how the nested datasets are

referenced. As you have seen in the preceding walk-throughs and code samples, you refer to a nested dataset using a dataset reference. For example, in the FDRuntimeNestedFields project, the references to the nested datasets were

FDMemTable variables labeled MidLevelMemTable and ThirdLevelMemTable.

Once you have a reference like this you can use that reference to invoke FireDAC datasets methods, such as Edit, Next, and Locate, as well as work with properties of the nested dataset using RecordNo, UpdatesPending, and State.

### 394 Delphi in Depth: FireDAC

What you cannot do is directly work with the nested dataset through its owner record. For example, in the FDRuntimeNestedFields you cannot do something like the following:

```
TopLevelMemTable.MideLevelMemTable.Next; // INVALID STATEMENT!
```

If you want to refer to a nested dataset by way of the current record, you must explicitly cast the associated field as a TDataSetField, after which you can access the NestedDataSet property, a TDataSet reference, of that field. For example, imagine that TopTab is an FDMemTable containing a nested dataset field named DataSet. Here is how you can refer to the nested dataset of the current record of TopTab through its FieldByName method:

```
var
```

```
NestedData: TDataSet;
```

```
begin
```

```
if not TopTab.FieldByName('DataSet').IsNull then
```

```
begin
```

```
NestedData :=
```

```
TDataSetField(TopTab.FieldByName('DataSet')).NestedDataSet;
```

```
if NestedData.State in dsEditMode then
```

```
NestedData.Post;
```

```
end;
```

```
...
```

The second observation is that, while nested dataset are powerful and useful,

they do introduce limits on how you can work with your data when compared to those situations where you load data from related tables into two or more separate FDDatasets.

The primary limit introduced by nested datasets is that they make searching for data across master table records slow and inconvenient. For example, imagine that you have written a contact management system using FDMemTables as the datasets, and want to employ a master-detail relationship between contacts and their various phone numbers.

One way to do this would be to utilize two tables, one for contacts, and one for contact phone numbers. Your application would then have to separately load contacts and contact phone numbers into two separate FDMemTables, and

employ user interface techniques (such as dynamic master-detail links or filters

### Chapter 13: More FDMemTables 395

or ranges, see *Chapter 9, Filtering Data* for details) to display the master-detail relationship between a given contact and their phone numbers.

An alternative would be to create a single contact FDMemTable, and include a nested dataset for contact phone numbers. The advantage of this approach is that you can easily create master-detail views that limit the display of a contact's phone numbers automatically.

The drawback to this second technique involves your options for working with the nested data. Specifically, imagine that you get a phone bill and want to know which contact is associated with a rather expensive call that appears on that bill.

If you load contact phone numbers into a separate FDMemTable, you can use one of the search techniques described in *Chapter 9, Searching Data*, to locate the phone number in the contact phone numbers table, which will quickly reveal the contact associated with that phone number.

By comparison, if you have embedded contact phone numbers as a nested dataset in the contacts table, no global search is possible. Your only solution is to navigate (scan) record-by-record through the contacts table, performing a separate search on each nested dataset. Such a search would be significantly slower, on average, than a search on a separate contact phone numbers table.

Does this affect your decision to employ nested datasets or not? The answer is that it depends on your reasons for using nested datasets. In most cases,

nested datasets are used to represent master-detail data in a shared structure. Nonetheless, I wanted to bring up this issue because it makes a difference in some cases.

In the next chapter, I discuss the SQL command preprocessor.

Chapter 14: The SQL Command Preprocessor 397

# **Chapter 14**

## **The SQL Command**

### **Preprocessor**

This feature, which in the past has been called Dynamic SQL, involves a preprocessor that examines your SQL statements, and under the right conditions, replaces text within those statements before passing the SQL to the underlying database. There are two roles played by the SQL command preprocessor. The

primary role of this preprocessor is to modify a somewhat generic SQL statement in order to accommodate the idiosyncratic syntax of the underlying database server. Although SQL is ‘standardized,’ each database supports its own dialect of SQL, reflecting its particular origin and feature set.

The value of the SQL command preprocessor will be apparent to any developer who has had to build applications that must work with more than one underlying database. For example, developers who write vertical market applications, those designed for a particular industry, and which will be used by many different companies, often have to support two or more popular databases, such as Oracle and MS SQL Server. Doing so permits the client companies to use their current database server for the application’s data, rather than requiring those companies to possibly have to support and maintain (and license) an additional database server.

The second role of the SQL command preprocessor is to provide you with enormous flexibility in writing your SQL statements. For example, the preprocessor permits you to embed macros anywhere within your SQL statements, so long as you assign valid values to those macros before attempting to execute the resulting SQL. Similarly, the preprocessor lets you include FireDAC’s scalar functions within your SQL statements, permitting you to

perform calculations or transformations that might otherwise be impossible.

Without involving FireDAC’s SQL command preprocessor, databases generally

support limited flexibility in the SQL that they execute. This flexibility is provided in the form of parameters, which are placeholders for values that can

appear in the predicates of the WHERE clause of SQL SELECT and DELETE

## 398 Delphi in Depth: FireDAC

queries, the SET clause of UPDATE queries, and the VALUES part of INSERT

queries. In general, parameters come in two forms, *named* and *unnamed*. A query that employs at least one parameter is referred to as a *parameterized query*.

In addition to providing some flexibility, parameterized queries can provide a performance advantage. Specifically, when you are going to execute the same basic query, albeit with different values in the parameters, the query only needs to be prepared once. For subsequent executions, only the new parameter values need to be communicated to the underlying database, and this saves time.

While parameterized queries offer some flexibility and potential performance enhancements, they are limited. Specifically, parameters can be used in place of literal values in SQL statements (such as the predicates in WHERE clauses), but cannot be used in place of identifiers, such as table names or column names.

By comparison, the FireDAC's SQL preprocessor permits a great deal of flexibility (though no specific performance benefits are realized beyond those gained through traditional SQL parameters). For example, features supported by the SQL preprocessor permit flexibility in both the SELECT clause and the FROM clause of SELECT statements, the UPDATE clause of UPDATE statements, and the target of DROP statements.

When the ResourceOptions.MacroCreate and ResourceOptions.MacroExpand properties of FireDAC datasets is set to True (the default), the SQL command preprocessor searches the SQL statement for macros and special escape characters that identify an operation it needs to perform. When it detects a macro or an escape sequence, it replaces the identified part of the SQL statement with new text, which it then passes along to the underlying database.

Importantly, the underlying database never sees the original text that includes the macros or escape sequences.

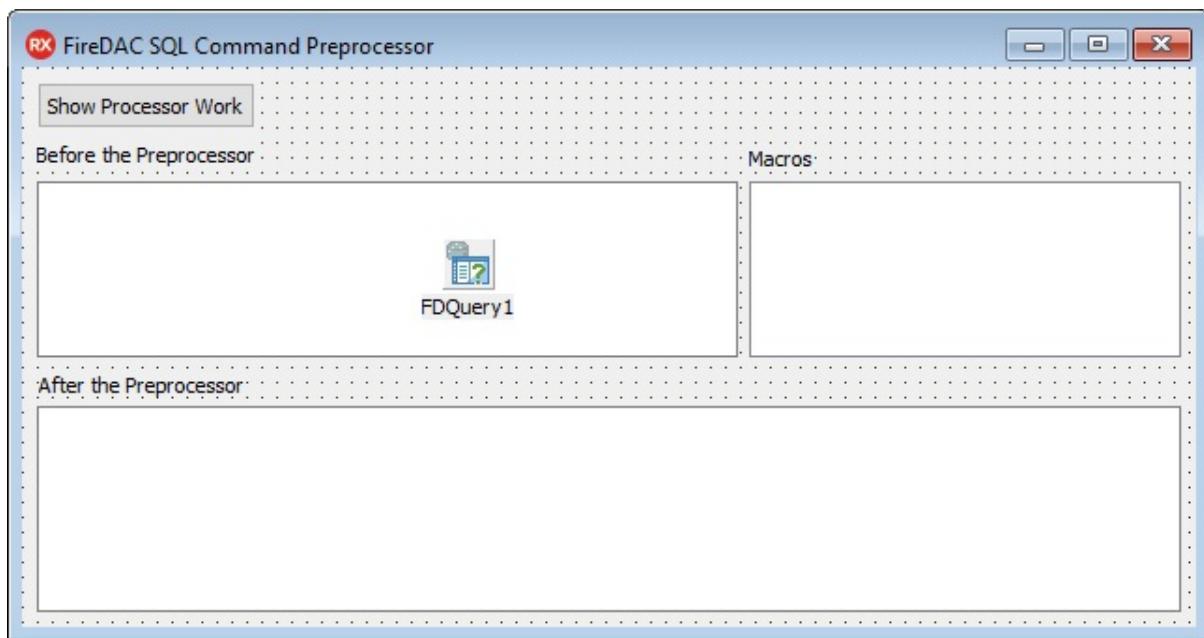
The SQL preprocessor supports two distinct types of operations. These are:

- Macro substitution

- Escape sequences

These operations are discussed in the following sections.

*Note: Parameterized queries are discussed in detail in Chapter 5, More Data Access. Refer to that chapter for more information about parameterized queries.*



## Chapter 14: The SQL Command Preprocessor 399

In order to view the workings of the SQL command preprocessor, I have built a little application that will display the original query, as well as the query as it appears once the preprocessor does its job. The main form of this project is shown in Figure 14-1.

**Figure 14-1: The FDSQLPreprocessor project main form**

*Code: The FDSQLPreprocessor project can be found in the code download. See Appendix A for details.*

Unlike almost every other project described in this book, this particular project is one that you have to configure at design time. Specifically, you need to define the query and assign any required values to variables that you have declared, all at design time. You then run the project and click the button labeled Show Processor Work. The event handler on this button displays the original query in the upper memo field, and prepares the query. After this, it displays any defined macros in the provided list box, as well as the post-processed query in the lower memo field. The post-processed query can be read from the FDQuery's Text

property. This event handler is shown here:

```
procedure TForm1.btnShowProcessorClick(Sender: TObject);
var
  i: Integer;
  Macro: TFDMacro;
begin
  Memo1.Lines.Clear;
  Memo2.Lines.Clear;
  Memo1.Lines.Add( FDQuery1.SQL.Text );
  FDQuery1.Prepare;
  ListBox1.Clear;
  for i := 0 to FDQuery1.MacroCount-1 do
  begin
    Macro := FDQuery1.Macros[i];
    ListBox1.Items.Add( 'Name: ' + Macro.Name + ', Type : ' +
      GetEnumName(TypeInfo(TFDMacroDataType),Ord(Macro.DataType)) +
      ', Value: ' + Macro.Value );
  end;
  Memo2.Lines.Text := FDQuery1.Text;
end;
```

## **Macro Substitution**

When you employ macro substitution, you embed macros in your SQL statements, even in those places where traditional SQL parameters are not allowed. For example, the use of macro symbols in your SQL statements permits you to write queries whose tables, fields, or WHERE clause predicates are not known until runtime. As mentioned earlier, your entire SQL statement may consist of a single macro. All you need to do is ensure that you have bound a valid value to each macro appearing in your SQL statement before executing the query, and the command preprocessor will take care of updating the

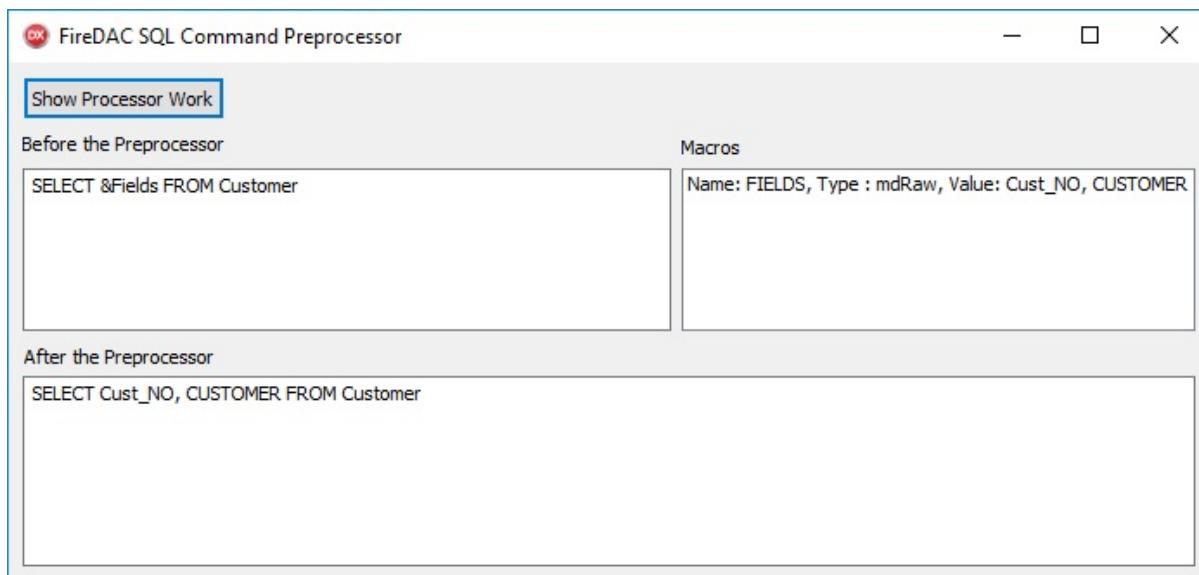
resulting SQL.

In the case of an SQL statement consisting of a single macro, that macro must be replaced by a valid and complete SQL statement before execution, which really makes no sense. If you are going to the trouble of replacing a single macro with an entire SQL statement, you might as well just assign a valid SQL

statement to your FDQuery's SQL property at runtime, instead of messing around with a macro.

Macros are identified by either the ! character or the & character, which you follow with the macro name. Prior to executing a query, you must provide a valid value for each of its macros using the Macros collection property of the dataset.

For example, consider the following SQL statement, which includes one macro: SELECT &Fields FROM Customer;



## Chapter 14: The SQL Command Preprocessor 401

Prior to executing this query, a statement similar to the following can be used to replace the &Fields macro with a suitable SELECT clause:

```
FDQuery1.Macros[0].AsRaw := 'CustNo, Company';
```

The use of the indexed Macros property in the preceding statement require you to know that the macro to which you were assigning a value was the first macro appearing in the SQL statement. Alternatively, you can use the MacroByName method to assign a value to a macro. When using MacroByName, you omit the

& or ! character, the character that identifies the macro, as shown here:

FDQuery1.MacroByName('Fields').AsRaw := 'CustNo, Company';

Figure 14-2 shows how this query, and its processed result, appear in the FDSQLPreprocessor project.

### **Figure 14-2: The effect of macro substitution is shown in the FDSQLPreprocessor project**

In the preceding examples, the macro name was preceded by the ampersand character (&). In this mode, referred to as *SQL substitution*, how the data is inserted into the SQL is typed, meaning that the macro can represent itself according to its data type in the SQL statement. This typing can be done

402 Delphi in Depth: FireDAC

programmatically, but is most obvious when you view the Macro tab of the SQL

Editor, where you can specify a data type for a macro.

By comparison, when you precede the macro name with the exclamation point

character (!), a mode referred to as *string substitution*, the value assigned to the macro is inserted as-is, where the value being concatenated into the SQL string is a string, without any conversion based on the underlying database.

While SQL substitution and string substitution were originally intended to be different, it turns out that they work the same. If you use the macro data type of mdRaw at design time, or assign to the AsRaw property of the macro at runtime, you get the intended behavior of string substitution, regardless of which

character (! or &) you use to define the macro. Likewise, using one of the other data types will perform the proper conversion, based on your connected database, whether you use ! or &.

Macro substitution is demonstrated in a second project named

FDMacroSubstitution. This project generates an SQL statement whose field list and table name is created dynamically at runtime. The main form of this project is shown in the Figure 14-3. The table shown on the right side of this figure is the result of the SQL statement execution.

Dynamic SQL - Macro Substitution

Tables	Show Data				
ANIMALS	CUSTNO	COMPANY	STATE	ZIP	COUNTRY
BIOLIFE	1221.00	Kauai Di...	HI	94766-1...	US
COUNTRY	1231.00	Unisco			Bahamas
CUSTOLY	1351.00	Sight Div...			Cyprus
<b>CUSTOMER</b>	1354.00	Cayman...			British W...
EMPLOYEE	1356.00	Tom Saw...	St. Croix	00820	US Virgin...
INDUSTRY	1380.00	Blue Jack...	HI	99776	US
ITEMS	1384.00	VIP Diver...	St. Croix	02800	US Virgin...
MASTER	1510.00	Ocean P...	HI	94756	US
NEXTCUST	1513.00	Fantastiq...			Columbia
	1551.00	Marmot...	Ontario	G3N 2E1	Canada
	1560.00	The Dept...	FL	35003	US
	1563.00	Blue Spo...	OR	91187	US
	1624.00	Makai SC...	HI	94756	US
	1645.00	Action Cl...	FL	32274	US
	1651.00	Jamaica...	Jamaica		West Ind...
	1680.00	Island Fi...	GA	32521	US
	1984.00	Adventur...			Belize

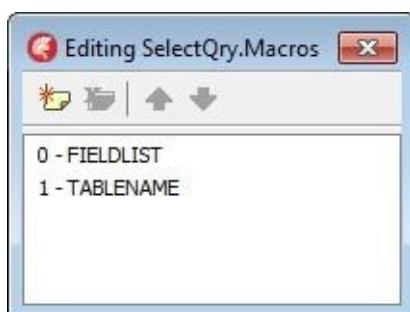
## Chapter 14: The SQL Command Preprocessor 403

**Figure 14-3: The fields that appear in the result set are selected at runtime** The actual query that is executed to create the tabular view on the right side of this form is shown here. As should be obvious, &FieldList and &TableName are macros:

SELECT &FieldList FROM &TableName

*Code: The FDMacroSubstitution project can be found in the code download.*

These macros can be assigned values at design time or at runtime, though it is the runtime assignment of values that is of particular value.





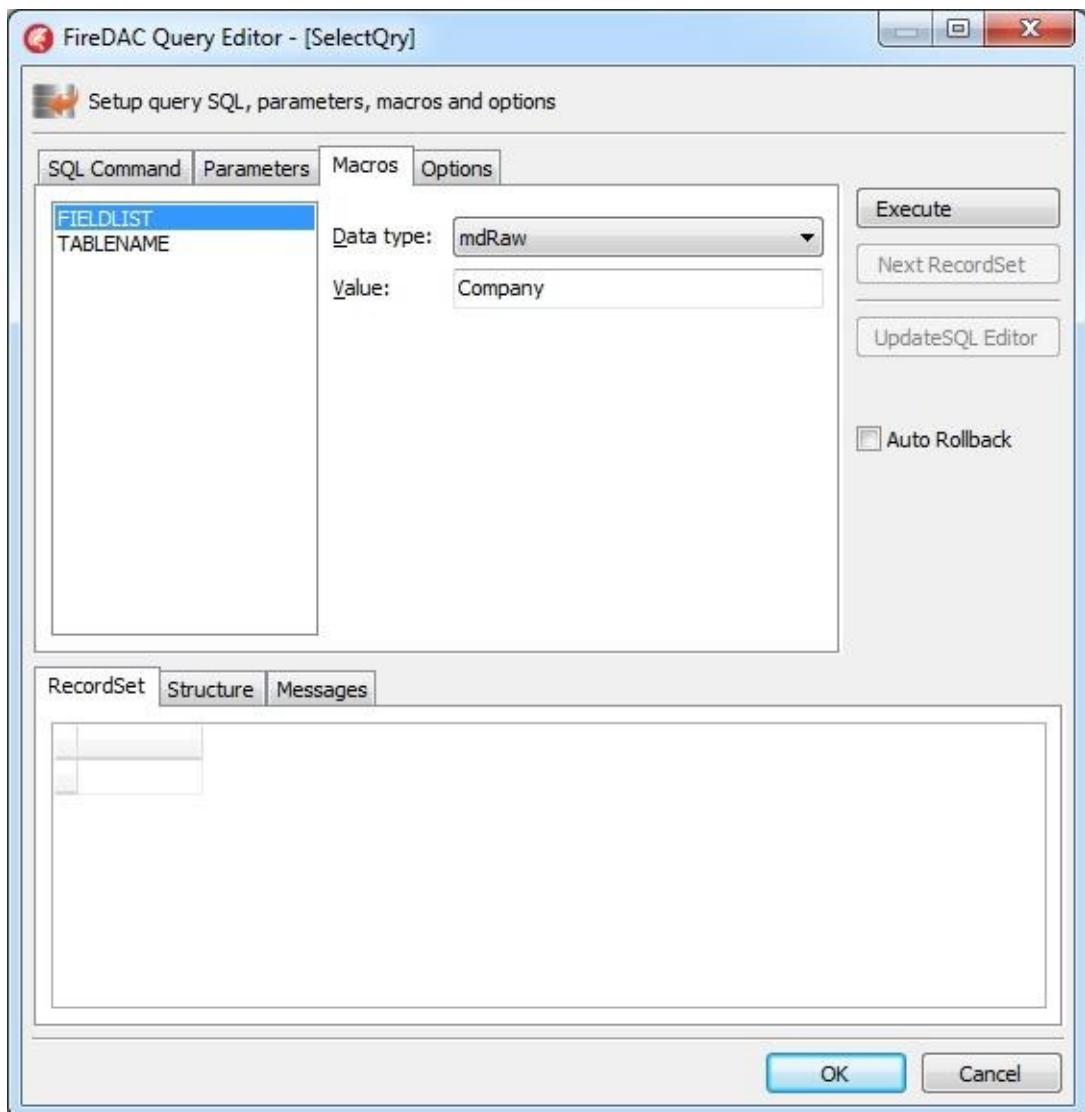
## 404 Delphi in Depth: FireDAC

Once an SQL statement that includes macros is assigned to an FDQuery, you can use the property editor for the Macros property to see the macros that FireDAC detects in your query. The Macros property collection editor for the query in the FDMacroSubstitution project is shown here:

You assign a value to one of these macro at runtime by selecting the macro name in the Macros property editor and then setting the Value property using the Object Inspector, as shown here:

You can also access, and assign values to, macros using the Query Editor.

Figure 14-4 shows the Query Editor with the Macros tab selected.



## Chapter 14: The SQL Command Preprocessor 405

**Figure 14-4: The Macros tab of the FireDAC Query Editor**

While you may set macros at design time in order to create a default query, it is the runtime assignment of data to the macros that creates the greatest flexibility.

The following is the LoadData method, which is called from the button labeled Show Data seen in Figure 14-3. This method generates a field list from the fields selected in the Fields list box, and uses the table name selected in the Tables list box:

```
procedure TForm1.LoadData;
```

```
var
```

```
 406 Delphi in Depth: FireDAC
```

```
  i: Integer;
```

```

FieldList: string;
ListBoxItem: TListBoxItem;
procedure BuildList(Value: string);
begin
if FieldList = " then
  FieldList := Value
else
  FieldList := FieldList + ', ' + Value;
end;
begin
  DataModule2.SelectQry.Close;
  for i := 0 to ListBox2.Items.Count - 1 do
    if ListBox2.ListItems[i].IsSelected then
      BuildList(ListBox2.ListItems[i].Text);
  DataModule2.SelectQry.Macros[0].AsRaw := FieldList;
  DataModule2.SelectQry.MacroByName('TableName').AsRaw :=
    ListBox1.Selected.Text;
  DataModule2.SelectQry.Open;
end;

```

*Note: When using macro substitution, please use great care if your text comes from user input. Macro substitution can open you to many of the same dangers that exist with the SQL injection hack.*

## Escape Sequences

Escape sequences are strings that you embed in your SQL which begin with an open curly brace character ({) preceded by a white space character, and ends with a close curly brace (}). This pair of characters signal the SQL command preprocessor that it should take an action. If you need to include the { or }

characters in some other part of your SQL statement, you must precede that character with an escape character. This process is described in greater detail later in this chapter in the section *Special Character Processing*.

There are four specific types of escape sequences in FireDAC. These are:

- Constant substitution

Chapter 14: The SQL Command Preprocessor 407

- Identifier substitution

- Conditional substitution

- FireDAC scalar functions

Each of these types is described in the following sections.

## **Constant Substitution**

Constant substitution permits you to identify a literal value in your SQL statement. The type of literal is defined by an identifier that precedes the literal, and this identifier is case insensitive. Prior to submitting the processed SQL

statement, the preprocessor ensures that the constant value is formatted appropriately for the underlying database.

FireDAC supports constant substitution for six types. These are shown in Table 14-1.

### **Constant Type Example**

#### **Comments**

Numeric literal

{E 3.14159}

A period (.) must be used as the decimal separate.

Date literal

{D 2014-09-02}

Use the ‘yyyy-mm-dd’ format.

Time literal

{T 13:12:59}

Use the ‘hh:mm:ss’ format, where hh is 00-23.

DateTime literal {DT 2014-09-02}

Use the ‘yyyy-mm-dd hh:mm:ss’

13:12:59}

format.

Boolean (logical) {L True}

Use True or False.

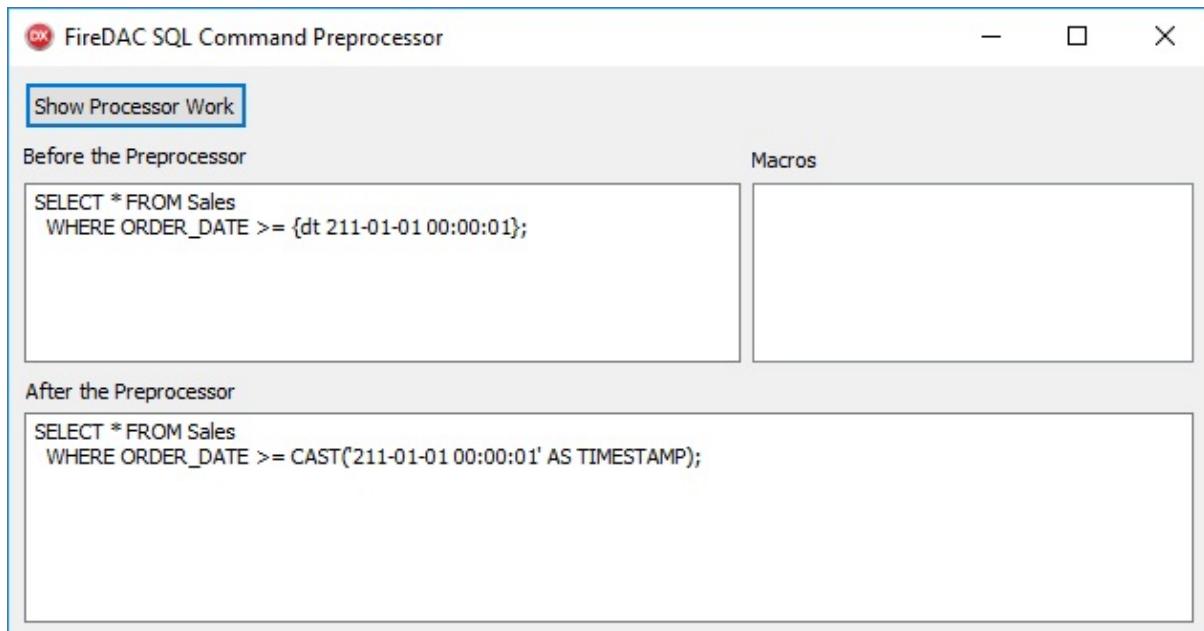
literal

String literal

{S ‘This is a string’} String can be quoted or not.

### Table 14-1: Constant Substitution Supported Types

For example, assuming that the FromTS field is a date/time type field, a timestamp, for instance, the following query will select records where the FromTS field contains values equal to or later than the first minute of the first day of January in the year 2017, and where the ThruTS field is null. This will



The screenshot shows the FireDAC SQL Command Preprocessor window. At the top, there's a button labeled "Show Processor Work". Below it, two sections are visible: "Before the Preprocessor" and "After the Preprocessor". In the "Before the Preprocessor" section, the following SQL query is shown:

```
SELECT * FROM Sales  
WHERE ORDER_DATE >= {dt 211-01-01 00:00:01};
```

In the "After the Preprocessor" section, the resulting SQL query after substitution is shown:

```
SELECT * FROM Sales  
WHERE ORDER_DATE >= CAST('211-01-01 00:00:01' AS TIMESTAMP);
```

### 408 Delphi in Depth: FireDAC

work regardless of the format that the underlying database requires for date/time constants.

```
FDQuery1.SQL.Text := 'SELECT * FROM Privileges ' +  
' WHERE StartTS >= {dt 2017-01-01 00:00:01} ' +  
' AND ThruTS IS null;'
```

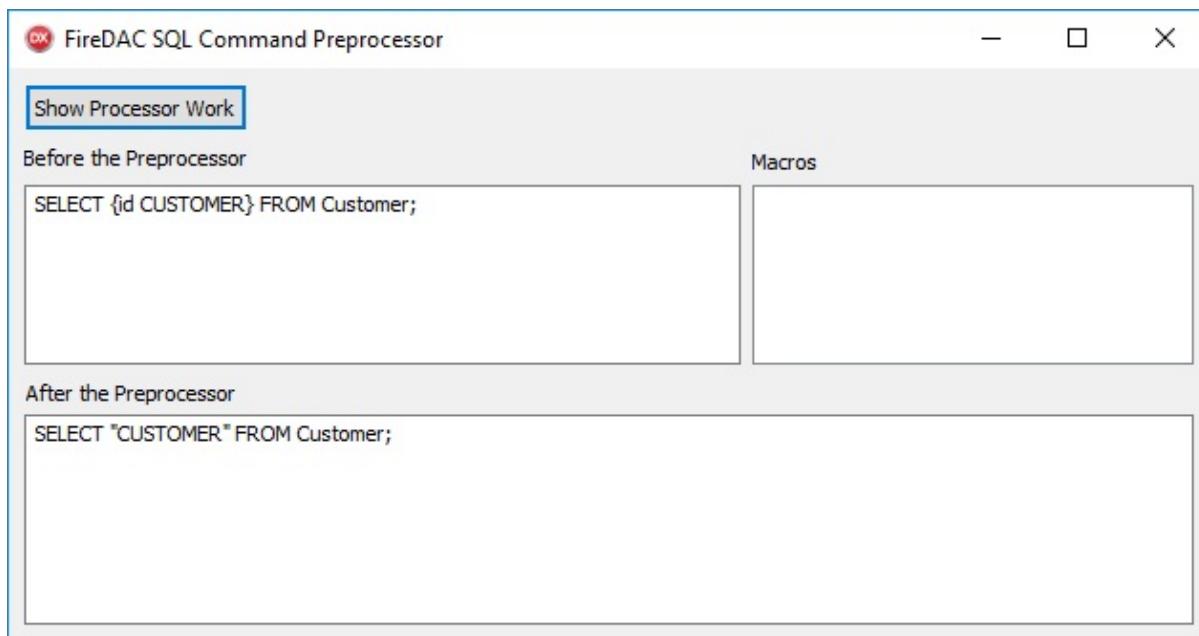
The results of the processing of this query are shown in Figure 14-5.

**Figure 14-5: Constant substitution and the resulting processed query**

## Identifier Substitution

You use identifier substitution to reference a field name or table name in your SQL statement using delimiters appropriate to the underlying database. For example, when a field name or table name includes white space, or some other non-standard characters, you often have to enclose the name in double quotes, square braces, or some other similar delimiter, depending on the database to which you are connected.

To use identifier substitution, enclose the field name or table name within curly braces and preceded by the characters ‘id.’ For example, consider the following SQL statement:



The screenshot shows the FireDAC SQL Command Preprocessor interface. At the top, there's a title bar with the application name and standard window controls. Below the title bar, a button labeled "Show Processor Work" is highlighted with a blue border. The main area is divided into two sections: "Before the Preprocessor" and "After the Preprocessor". In the "Before the Preprocessor" section, the SQL query is displayed as "SELECT {id CUSTOMER} FROM Customer;". In the "After the Preprocessor" section, the query is shown as "SELECT \"CUSTOMER\" FROM Customer;". To the right of the "Before" section, there is a "Macros" panel which is currently empty.

Chapter 14: The SQL Command Preprocessor 409

SELECT {id First Name} FROM Customer;

Since the First Name field includes white space, it must be delimited prior to sending it to the underlying database. The use of identifier substitution ensures that delimiters appropriate for your connected database are substituted. As a result, if the underlying database is InterBase, the SQL statement will look like that shown in Figure 14-6.

**Figure 14-6: The effects of identifier substitution are shown in the**

**FDSQLPreprocessor project**

### Conditional Substitution

Conditional substitution permits you to include statements that FireDAC will substitute during preprocess with SQL syntax appropriate to the underlying database. In some cases, however, the correct syntax can only be known by

you, in which case, you must provide the various alternative SQL segments that will be substituted at runtime.

For instance, imagine that your application supports both InterBase and MS SQL Server. Suppose further that at least one table that your application uses is not under your control (it was created by your client), and that the date field of

The screenshot shows the FireDAC SQL Command Preprocessor interface. At the top, there's a toolbar with a 'Show Processor Work' button, a minimize button, a maximize button, and a close button. Below the toolbar, there are two main sections: 'Before the Preprocessor' and 'After the Preprocessor'. In the 'Before the Preprocessor' section, the SQL code is:

```
SELECT PO_NUMBER,{iif (INTRBASE, ORDER_DATE, MSSQL, ORDDATE) }  
FROM Sales;
```

In the 'After the Preprocessor' section, the SQL code is simplified to:

```
SELECT PO_NUMBER, ORDER_DATE FROM Sales;
```

On the right side of the interface, there is a 'Macros' section which is currently empty.

## 410 Delphi in Depth: FireDAC

this table (Orders) is named OrdDate in your InterBase database, but is named OrderDate in the MS SQL Server database.

FireDAC supports two flavors of conditional substitution. In the first, you list two or more database identifiers, preceded by the characters if or iif, and followed by two or more corresponding SQL statements that you want to substitute conditionally. For example, the following can be used to select the OrderID and order date fields from the underlying database (see Figure 14-7), where the order date field is named ORDER\_DATE in InterBase and ORDDATE in MS SQL Server:

```
SELECT PO_NUMBER,{iif (INTRBASE, ORDER_DATE, MSSQL,  
OrderDate) }  
FROM Sales;
```

### Figure 14-7: The result of conditional substitution

In most cases, you will include one value for each database identifier. However, you can include one extra value, at the end, which will be used in

the case that none of the identifiers match the connected database. For example, in the

following version, which also uses ‘if’ instead of ‘iif,’ the field name ODate will be used if the connected database is neither MS SQL nor InterBase.

```
SELECT OrderID,  
{if (MSSQL, OrdDate, INTRBASE, ORDER_DATE, ODate) }  
FROM Sales;
```

## Chapter 14: The SQL Command Preprocessor 411

The second type of conditional substitution makes use of a statement similar to a simple if statement, where the identified statement is substituted for the escape sequence if the identifier matches the database to which FireDAC is connected.

For example, consider the following statement. Here the OrdDate field will only be returned when the connected database is InterBase. Since two fields, not one, will the statement is querying the InterBase database, the comma is also

included in the conditional part of the escape sequence:

```
SELECT PO_NUMBER{IF INTRBASE}, ORDER_DATE {FI} FROM  
Sales;
```

Table 14-2 lists the database identifiers used by conditional substitution.

### **Identifier**

### **Database**

ADS

Advantage Database Server

ASA

Sybase SQL Anywhere

DB2

IBM DB2

FIREBIRD

Firebird

INFORMIX

Informix

INTRBASE

InterBase

MSACCESS, MSACC

Microsoft Access database

MSSQL

Microsoft SQL Server

MYSQL

MySQL Server

ORACLE, ORA

Oracle Server

OTHER

Some other database (ODBC)

POSTGRESQL, PG

PostgreSQL Server

SQLITE

SQLite database

TERADATA, TDATA

Teradata database

## **Table 14-2: FireDAC Database Identifiers**

412 Delphi in Depth: FireDAC

### **FireDAC Scalar Functions**

FireDAC's SQL command preprocessor supports a large number of scalar functions that you can embed in your SQL, as well as other expressions (evaluated expressions, filter expressions, calculated field expressions, filter-based index expressions). FireDAC supports the followings scalar function categories:

- String/Character
- Numeric
- Date/Time
- System

## 🎬 Convert

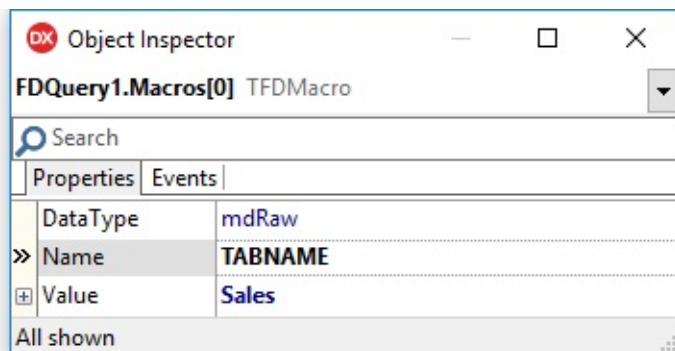
Like other escape sequences, FireDAC scalar functions appear within a pair of curly braces, and may optionally be preceded by the characters fn (within the braces) to make it clear that a function is being invoked. For example, the CURRENT\_DATE() scalar function returns the current date value formatted appropriately for the underlying database. The following SQL statement queries the Sales table for records whose date is prior to today:

```
SELECT * FROM Sales WHERE ORDER_DATE < {CURRENT_DATE};
```

The use of the fn characters is optional, but can be helpful in making it obvious that a scalar function is being called. Consequently, the following statement is functionally identical to the preceding SQL statement:

```
SELECT * FROM Sales WHERE ORDER_DATE < { fn  
CURRENT_DATE};
```

The FDMacros project includes a query that employs macro substitution, constant substitution, and FireDAC scalar functions. The following is the text of the query executed by this project. CURRENT\_DATE() and EXTRACT(), which are enclosed in a pair of matching curly braces, are FireDAC date/time scalar functions. &TabName, as in the preceding example, is a macro. Finally, {e 1988} is a constant substitution escape sequence, which formats the value consistent with the format required by the underlying database:



The screenshot shows a software interface titled "FireDAC and Queries with Macros". At the top is a toolbar with various icons. Below it is a grid view displaying a dataset. The columns are labeled: TODAY, PO\_NUMBER, CUST\_NO, SALES REP, ORDER\_STATUS, ORDER\_DATE, and SH1. The data rows are:

TODAY	PO_NUMBER	CUST_NO	SALES REP	ORDER_STATUS	ORDER_DATE	SH1
3/26/2017	V92E0340	1004	11	shipped	10/16/2011	10/
3/26/2017	V92J1003	1010	61	shipped	7/27/2011	8/5
3/26/2017	V92F3004	1012	11	shipped	10/16/2011	1/1

Chapter 14: The SQL Command Preprocessor 413

```
SELECT {CURRENT_DATE()} as Today, c.*
```

```
FROM &TabName c
```

```
WHERE {Extract(Year, c.ORDER_DATE)} = {e 2011}
```

The following illustration shows the Macros property of the FDQuery that executes this query, depicting the assignment of the value to the &TabName variable.

The result set produced by this query is shown in Figure 14-8.

### **Figure 14-8: A preprocessed query returns a result set**

*Code: The FDMacros project can be found in the code download.*

In addition to being used in SQL statements, most of FireDAC's scalar functions are available for use with the *local expression engine*. The local

414 Delphi in Depth: FireDAC

expression engine is the feature that permits you to include FireDAC scalar functions in expressions, specifically filter expressions, filter-based index expressions, calculated fields, and custom expressions produced by the FireDAC expression evaluator. When FireDAC scalar functions are used in expressions, the use of the curly braces is optional. In other words, the scalar function used in expressions can appear inline with or without being enclosed the curly braces.

The following is an example of the LENGTH string/character scalar function being used in a filter expression. This example was shown in *Chapter 9, Filtering Data*:

```
FDQuery1.Filter := 'LENGTH([Company]) = 12';
```

*Note: Custom expressions created with the FireDAC expression evaluator are beyond the scope of this book. For information on custom expressions, please refer to the section Writing Expressions in the FireDAC help.*

As alluded to above, most of FireDAC's scalar functions are available to the local expression engine, but not all. The few FireDAC scalar functions that are not available to the local expression engine are identified in the following tables with an asterisk following the function prototype.

FireDAC's support for scalar functions typically relies on the database to which you are connected. As a result, the values returned are normally based on the configuration and features of your database. For example, the DAYNAME()

date/time scalar function will return a string based on your database configuration, which will control the language used to define the string.

Likewise, the date format of CURDATE() will rely on the date format configured on the database.

When a scalar function is not dependent on the database, the value returned may be based on your operating system's regional settings. Finally, there are some functions which rely on a database, and depending on the database to which you are connected, may not be supported at all. Under those conditions, your use of that FireDAC scalar function against that database will raise an “Escape

function [%s] is not supported” exception. For example, the TIMESTAMPADD() scalar function is not supported when connected to InterBase.

## Chapter 14: The SQL Command Preprocessor 415

Before I continue, I want to acknowledge that the information in the following tables was drawn largely from the tables that appear in the RAD Studio help system. I have paraphrased the information to some extent, but similarities are unavoidable, given that the help system tables are among the only source of information about these functions. There are cases, however, where the help system contains more detail than I have included here. As a result, if you need additional information about a given function, you should refer to the help, as well as the Embarcadero Wiki, for additional details.

## **FIREDAC STRING/CHARACTER SCALAR FUNCTIONS**

FireDAC's string/character scalar functions return information related to strings or characters within strings. FireDAC's string/character scalar

functions are listed in Table 14-3.

## Function

### Returns

ASCII(string)

The ASCII code value of the leftmost character of the string or character as an integer.

BIT\_LENGTH(string) \*

The length, in bits, of the string expression.

CHAR(8byteInt)

The character that has the ASCII code of the parameter. The value should be between 0 and 255.

CHAR\_LENGTH(string)

The length in characters of the string expression.

*See CHARACTER\_LENGTH*

CHARACTER\_LENGTH( The length in characters of the string expression.  
string)

*See CHAR\_LENGTH.*

CONCAT(string1, string)

The character string that is the result of concatenating string1 and string2, depending on the rules defined by your database.

INSERT(string1,

A string where length characters from string1  
start, length, string2)

have been inserted into string2, beginning at start position.

416 Delphi in Depth: FireDAC

LCASE(string)

The lower-case version of string.

**LEFT(string, count)**

The left count characters of string.

**LENGTH(string)**

The length of string in number of characters.

**LOCATE(string,**

The position of the first occurrence of string1

string2 [, start])

within string2, beginning from position 1 or the

optional start position. If not found, returns 0.

**LTRIM(string)**

String with the left-most blank space removed.

**OCTET\_LENGTH(string)**

The length in bytes of string.

**POSITION(string1, string2)** The position of string1 in string2.

**REPEAT(character, count)** A string composed of count characters.

**REPLACE(string1, string2, string3)**  
A string where instances of string2 within string1  
are replaced by string3.

**RIGHT(string, count)**

The right count characters of string.

**RTRIM(string)**

String with the right-most blank space removed.

**SPACE(count)**

A character of count spaces.

**SUBSTRING(string, start,**

A string segment from string, beginning at start  
count)

position and extending for count characters.

**UCASE(string)**

The upper-case version of string.

\* Not available from the local expression engine

### **Table 14-3: FireDAC's string/character scalar functions**

The use of the UCASE() string scalar function is shown in the FDSQLPreprocessor project in Figure 14-9.

The screenshot shows the FireDAC SQL Command Preprocessor application window. At the top, there is a title bar with the application name and standard window controls. Below the title bar, a button labeled "Show Processor Work" is highlighted with a blue border. The main interface is divided into two main sections: "Before the Preprocessor" and "After the Preprocessor". In the "Before the Preprocessor" section, there is a code editor containing the following SQL query:

```
SELECT { fn UCASE( CONTACT_FIRST ) }, { UCASE(CONTACT_LAST) }  
FROM CUSTOMER
```

In the "After the Preprocessor" section, the code editor shows the result of the preprocessing, where the scalar functions have been expanded:

```
SELECT UPPER( CONTACT_FIRST ), UPPER(CONTACT_LAST ) FROM CUSTOMER
```

Chapter 14: The SQL Command Preprocessor 417

### **Figure 14-9: The results from string/character scalar functions in the FDSQLPreprocessor project**

*Note: In my initial installation of Delphi, some of these scalar functions, including both string and numeric types, did not work with InterBase until I ran an SQL script named ib\_udf.sql, which is located in the examples directory under the InterBase home directory. For more information, please see Appendix A.*

## **NUMERIC SCALAR FUNCTIONS**

FireDAC's numeric scalar functions return information related to numbers and arithmetic functions. FireDAC's numeric scalar functions are listed in Table 14-4.

### **Function**

### **Returns**

**ABS (number)**

The absolute value of number.

**ACOS(number)**

The arccosine of number as an angle, expressed in radians.

418 Delphi in Depth: FireDAC

**ASIN(number)**

The arcsine of number as an angle, expressed in radians.

**ATAN(number)**

The arctangent of number as an angle, expressed in radians.

**ATAN2(number1,**

The arctangent of the x and y coordinates,  
number2)

specified by number1 and number2, as an angle,  
expressed in radians.

**CEILING(number)**

The smallest integer greater than or equal to  
number.

**COS(number)**

The cosine of number, where number is an angle  
expressed in radians.

**COT(number)**

The cotangent of number, where number is an  
angle expressed in radians.

**DEGREES(number)**

Degrees converted from number radians.

**EXP(number)**

The exponential value of number.

**FLOOR(number)**

The largest integer less than or equal to number.

**LOG(number)**

The natural logarithm of number.

**LOG10(number)**

The base 10 logarithm of number.

**MOD(number1, number2)** The remainder (modulus) of number1 divided by number2.

**PI()**

The constant value of pi ( $\pi$ ) as a floating-point value.

**POWER(number1,**

The value of number1 to the power of number2.

**number2)**

**RADIANS(number)**

The number of radians converted from number degrees.

Chapter 14: The SQL Command Preprocessor 419

**RAND([number])**

A random floating-point value using number as the optional seed value.

**ROUND(number1,**

Number1 rounded to number2 places right of the number2)

decimal point. If number2 is negative, number1 is rounded to number2 places to the left of the decimal point.

**SIGN(number)**

An indicator of the sign of number. Returns -1 if

number is less than zero, 0 if number is 0, and 1 if number is greater than zero.

**SIN(number)**

The sine of number, where number is an angle expressed in radians.

**SQRT(number)**

The square root of number.

**TAN(number)**

The tangent of number, where number is an angle expressed in radians.

**TRUNCATE(number1,**

Number1 truncated to number2 places right of number2)

the decimal point. If number2 is negative, number1 is truncated to the left of the decimal point.

#### **Table 14-4: FireDAC's numeric scalar functions**

#### **DATE/TIME SCALAR FUNCTIONS**

FireDAC's date/time scalar functions return information related to dates, times, and periods. FireDAC's date/time scalar functions are listed in Table 14-5.

#### **Function**

#### **Returns**

**CURDATE( )**

The current date.

**CURRENT\_DATE( )**

The current date on the server.

**CURRENT\_TIME[**

The current local time. The precision (precision)]

argument determines the seconds precision of the returned value.

#### 420 Delphi in Depth: FireDAC

CURRENT\_TIMESTAMP[ The current local date and local time as a (precision)]

timestamp value. The precision argument determines the seconds precision of the returned timestamp.

#### CURTIME( )

The current local time.

#### DAYNAME(date)

A string containing name of the day based on date.

#### DAYOFMONTH(date)

The day of the month as an integer base on the month field in date.

#### DAYOFWEEK(date)

The day of the week as an integer based on the week field in date.

#### DAYOFYEAR(date)

The day of the year as an integer based on the date.

#### EXTRACT(extract-field,

The extract-field portion of the extract-source.  
extract-source)

The extract-source argument is a datetime or interval expression. The extract-field argument can be one of the following keywords: YEAR, MONTH, DAY, HOUR,

MINUTE, SECOND.

HOUR(time)

The hour as an integer based on time.

MINUTE(time)

The minute as an integer based on time.

MONTH(date)

The month as an integer based on date.

MONTHNAME(date)

A string containing the name of the month  
based on date.

NOW( )

The current date and time as a timestamp  
value.

QUARTER(date)

The quarter as an integer in date.

SECOND(time)

The second as an integer based on time.

Chapter 14: The SQL Command Preprocessor 421

TIMESTAMPADD(interval, A timestamp calculated by adding integer  
integer, timestamp\_exp)

intervals of type interval to timestamp. Valid  
values of interval include FRAC\_SECOND,  
SECOND, MINUTE, HOUR, DAY, WEEK,  
MONTH, QUARTER, and YEAR where  
fractional seconds are expressed in billionths  
of a second.

TIMESTAMPDIFF(interval, The integer number of intervals of type  
timestamp1, timestamp2)

interval by which timestamp2 is greater than

timestamp1. Valid values of interval include FRAC\_SECOND (\*), SECOND, MINUTE, HOUR, DAY, WEEK, MONTH, QUARTER, YEAR, where fractional seconds are expressed in billionths of a second.

**WEEK(date)**

The week of the year based on the date.

**YEAR(date)**

The year as an integer based on date.

#### **Table 14-5: FireDAC's date/time scalar functions**

#### **SYSTEM SCALAR FUNCTIONS**

FireDAC's system scalar functions return information about the connection, as well as perform specific, miscellaneous functions. FireDAC's string/character scalar functions are listed in Table 14-6.

##### **Function**

##### **Returns**

**DATABASE( )**

The name of the connected database.

**IFNULL(exp, value)**

Value if exp is null. Exp if exp is not null.

**IF(exp, value1, value2)** Value1if exp is True, otherwise value2.

**IIF(exp, value1, value2)** Value1if exp is True, otherwise value2.

**LIMIT([skip,] rows) \***

Allows limiting the result set, by skipping first

<skip> records and returning to more than <rows>

422 Delphi in Depth: FireDAC

records. The function can be put in any place of the SQL command.

**NEWGUID( )**

A new, randomly-generated GUID value.

**USER( )**

The connected user name in the DBMS.

### **Table 14-6: FireDAC's system scalar functions**

#### **THE CONVERT SCALAR FUNCTION**

FireDAC supports the CONVERT scalar function, which is designed to convert one data type to another, compatible data type. The CONVERT scalar function has the following syntax:

`CONVERT(value_exp, data_type)`

As you can see, the CONVERT function takes two parameters, a data value and an identifier that indicates the data type to which you want to convert the value in the first parameter. The value of the first parameter can be a field identifier, a literal value, or the result of an expression evaluation, including expressions that include FireDAC scalar functions.

Valid CONVERT function identifiers are listed in Table 14-7.

BIGINT

BINARY

BIT

CHAR

DATE

DECIMAL

FLOAT

GUID

INTEGER

LONGVARBINARY LONGVARCHAR

NUMERIC

REAL

SMALLINT

TIME

TIMESTAMP

VARBINARY

VARCHAR

WCHAR

WLONGVARCHAR WVARCHAR

### **Table 14-7: FireDAC CONVERT function conversion identifiers**

Chapter 14: The SQL Command Preprocessor 423

The FireDAC CONVERT scalar function plays an important role in coordination with other FireDAC scalar function usage. Specifically, FireDAC

does not necessarily guarantee the data type of data returned by FireDAC scalar function calls. In some cases, the data returned by a FireDAC scalar function call is ambiguous. In those cases, you should use the CONVERT function to

ensure that the returned data from a FireDAC scalar function call is compatible with your needs.

### **Special Character Processing**

FireDAC's SQL command preprocessor identifies variables and escape sequences using the following characters:

& ! { }

In addition, the ? and : characters identify SQL parameters.

If you actually want to use these characters in your SQL statements and not have them interpreted by the SQL preprocessor, you have to take appropriate steps. In most cases, you can pass these characters to the underlying database by using the special character twice in a row. For example, include two consecutive curly braces {{}} to avoid having the preprocessor interpret the opening curly brace as the beginning of an escape sequence.

Another option is turn off the SQL command preprocessor entirely. For example, if you do not want the SQL preprocessor to perform preprocessing, set ResourceOptions.MacroCreate, ResourceOptions.MacroExpand and ResourceOptions.EscapeExpand to False. If you do not want the FireDAC preprocessor to recognize SQL parameter markers, set ResourceOptions.ParamCreate and ResourceOptions.ParamExpand to False.

In the following chapter, you will learn about Array DML.

Chapter 15: Array DML 425

# Chapter 15

## Array DML

This is a short chapter, so I am going to begin it with a true story. A coworker approached me recently, looking for advice on how to most efficiently insert a large number of records into a table. Since we were using FireDAC as our data access mechanism, I suggested Array DML, a feature of FireDAC that supports the batch processing of parameterized queries supported by a large number of databases, including the one that we were using.

I explained how Array DML worked, and provided him with a code sample that demonstrated essentially what he wanted to achieve. Later that day he returned to my office, sporting a big grin. “It worked,” he said. I asked him to be more specific, and he described how he succeeded in inserting more than 18,000

records into a database table in about two seconds.

Now that’s performance. I can’t guarantee that everyone can achieve numbers like these, but if you are going to try, Array DML is the way to go, if it’s an option for you.

Array DML provides a mechanism that supports high-speed data manipulation

using parameterized query and stored procedure execution. The DML part of the name “Array DML” name refers to SQL DML statements, where the initials

DML stand for *Data Manipulation Language*, as opposed to SQL DDL, which stands for *Data Definition Language*.

In its most basic form, there are two parts to using Array DML. The first part is defining a parameterized SQL DML statement, such as a parameterized UPDATE or INSERT statement. Similarly, you can use a parameterized stored procedure call.

The second part involves configuring an array to hold the parameter values. This array has one element for each time you want the query or stored procedure to be executed, and each element consists of a set of one or more parameters, the number of which depends on how many parameters are found in the associated query or stored procedure that you need to execute.

Array DML is something that you will use only when you need to execute the same query or stored procedure repeatedly, albeit with a different set of

## 426 Delphi in Depth: FireDAC

parameter values upon each execution. There are, however, many situations in which you need to do just that. For example, data warehousing applications often involve extracting data from the live database, manipulating that data, and then inserting the processed data into the data warehouse database. These operations are referred to as ETL (extract, transform, and load) operations, and they are perfectly suited for Array DML.

There are two principle advantages of Array DML. For those databases that directly support these types of batch command operations, the amount of communication between your application and the database server can be dramatically reduced, and optimizations on the server can produce exceptional performance.

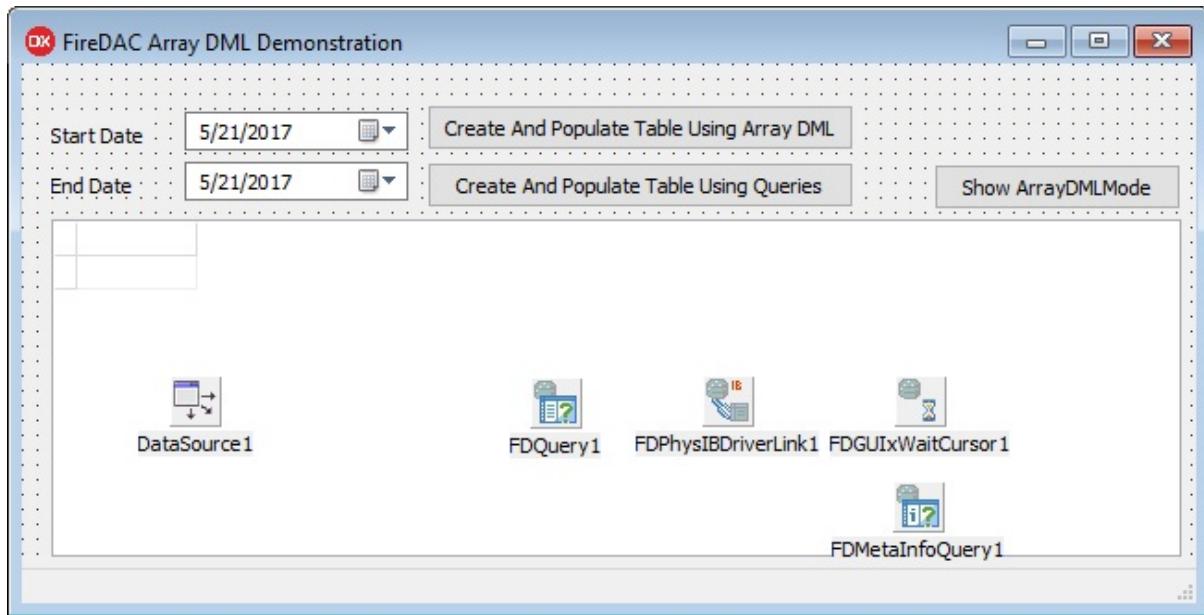
The second advantage is that Array DML provides a coherent framework for managing the manipulation of large amounts of data. The benefits of this framework include automatic record commitments, transaction rollbacks, and error tracking. These benefits can be realized even when the underlying database does not natively support batch command operations.

Most of the more popular servers for which FireDAC provides drivers support Array DML natively. These include InterBase (XE3 and higher), Firebird (2.1 and higher), Microsoft SQL Server, Oracle, IBM DB2, Informix, MySQL, Sybase SQL Anywhere, PostgreSQL (8.1 and higher), and SQLite (3.7.11 and higher when Params.BindMode is set to pbByNumber). The other database engines supported by FireDAC also permit you to use Array DML techniques. However, for these servers, the operations are emulated by FireDAC, and do not necessarily provide a performance benefit. As mentioned earlier, however, the Array DML framework has additional benefits, making it useful even when not natively supported by the underlying database.

## Using Array DML

The benefits of Array DML are demonstrated in the FDArrayDML project. The

main form of this project is shown in Delphi's designer in the Figure 15-1.



Chapter 15: Array DML 427

### Figure 15-1: The FDArrayDML project main form

This project uses SQL queries to create a table that contains the date and that date's day of the week for every day between two dates, inclusively. By default, the project initializes the start date to the current date, and the end date to be ten years after the start date. As a result, by default this project will need to insert over 3600 records into the target table.

*Code: The FDArrayDML project is in the code download*

There are two methods in this project that prepare the database before the records can be inserted. One of these is the OnCreate event handler for the main form. Two TDateTimePicker components are initialized from within this method, which are used to produce a range of dates. This date range is used to define the records that will be inserted into a target database table.

```
procedure TForm1.FormCreate(Sender: TObject);
```

```
begin
```

```
  DateTimePicker1.Date := DateOf(now);
```

```
  DateTimePicker2.Date := DateOf(incyear(now, 10));
```

```
end;
```

The second method prepares the table into which those date records will be inserted. The table is dropped if it already exists. Next, a new instance of this

428 Delphi in Depth: FireDAC

table is created. This routine, which is named `CreateTable`, is executed before

each and every insertion process. This method is shown in the following code segment:

```
procedure TForm1.CreateTable(TableName: string);
begin
  //Dropping the table in a transaction that is committed
  //ensures that the table is gone before proceeding
  SharedDMVcl.FDConnection.StartTransaction;
  try
    // The second parameter of ExecSQL instructs FireDAC to ignore
    // errors raised if the table to be dropped does not exist.
    FDQuery1.Connection.ExecSQL('DROP TABLE ' + TableName, True);
  finally
    SharedDMVcl.FDConnection.Commit;
  end;
  // Creating the table in a transaction that is committed
  // ensures that the table exists before proceeding
  SharedDMVcl.FDConnection.StartTransaction;
  try
    SharedDMVcl.FDConnection.ExecSQL('CREATE TABLE ' + TableName +
    '(' +
    ' DateValue date NOT NULL Primary Key,' +
    ' DayName varchar(16) ');
  finally
    SharedDMVcl.FDConnection.Commit;
  end;
  end;
```

One of the buttons on the main form initiates the process of Array DML. This button is labeled Create And Populate Table Using Array DML, and its OnClick event handler is shown here:

```
procedure TForm1.UsingArrayDMLButtonClick(Sender: TObject);
```

```

const
  TableName = 'Dates';
begin
  if DateTimePicker1.Date >= DateTimePicker2.Date then
    raise Exception.Create('Starting date cannot follow ' +
      'ending date');
  CreateTable(TableName);
  PopulateTableArrayDML(TableName);
end;

```

## Chapter 15: Array DML 429

The method that performs the record insertion using Array DML is called `PopulateTableArrayDML`, and the code associated with that method is shown in the following code segment:

```

procedure TForm1.PopulateTableArrayDML(TableName: string);
var
  CurrentDate: TDateTime;
  TotalDays: Integer;
  i: Integer;
  StopWatch: TStopWatch;
begin
  TotalDays := DateUtils.DaysBetween(DateTimePicker1.Date,
    DateTimePicker2.Date);
  CurrentDate := DateTimePicker1.Date;
  //Define the parameterized query
  FDQuery1.SQL.Text := 'INSERT INTO ' + TableName +
    ' (DateValue, DayName) ' +
    'VALUES( :date, :name )';
  //Configure query params for performance. Optional
  FDQuery1.Params[0].DataType := ftDate;
  FDQuery1.Params[1].DataType := ftString;

```

```
FDQuery1.Params[1].Size := 16;  
StopWatch := TStopWatch.StartNew;  
//Set the array size  
FDQuery1.Params.ArraySize := TotalDays;  
//Insert the parameters  
for i := 0 to TotalDays - 1 do  
begin  
  FDQuery1.Params[0].AsDates[i] := DateOf(FromDate);  
  FDQuery1.Params[1].AsStrings[i] :=  
    FormatDateTime('dddd', FromDate);  
  FromDate := IncDay(FromDate);  
end;  
SharedDMVcl.FDConnection.StartTransaction;  
try  
  //Execute the query  
  FDQuery1.Execute(FDQuery1.Params.ArraySize);  
  SharedDMVcl.FDConnection.Commit;  
finally  
  SharedDMVcl.FDConnection.Rollback;  
end;  
430 Delphi in Depth: FireDAC  
//Stop timer  
StopWatch.Stop;  
//Let's take a look at the data  
FDQuery1.Open('SELECT * FROM ' + TableName);  
DataSource1.DataSet := FDQuery1;  
StatusBar1.SimpleText := 'Array DML Milliseconds : ' +  
  IntToStr(StopWatch.ElapsedMilliseconds);  
end;
```

As you can see in this code segment, once the basic parameters of execution are calculated, a parameterized INSERT statement defines how data will be inserted into the underlying table.

Next, the query parameters themselves are configured. In this case, this is an optional step as FireDAC will determine the parameter types based on the values assigned to the associated array. However, it is often helpful if you specifically identify the types of parameters you are defining, and, when one or more parameters are strings, you can optimize the operation by specifying the size of string parameters.

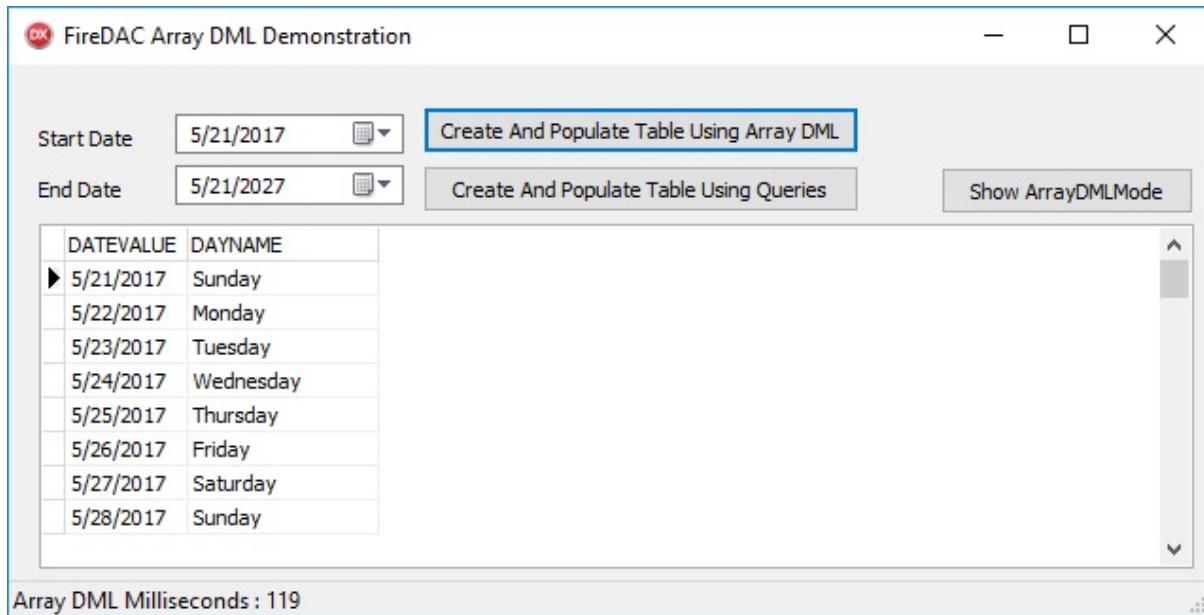
Before any parameter values can be assigned, it is necessary to configure the size of the Params array. You do this by assigning an integer value to the FDQuery's Params.ArraySize property. At a minimum, this value must be as large as or larger than the number of records you need to insert.

Finally, Array DML is triggered by calling the FDQuery's Execute method, to which you pass an integer indicating the size of the array that you have created.

Before each time the query is executed, data found in the Params array is bound to query's parameters.

The Execute method has a second, default parameter, which is 0 by default. This value represents the offset, which defines the array element from which to start the execution. With the default value of 0, the query executes beginning with the parameter values found in the first element of the array.

Figure 15-2 shows the results of a sample run of this project using the Array DML technique. A TStopWatch was used to gauge performance, and in this instance, more than 3500 records were inserted in about 119 milliseconds.



## Chapter 15: Array DML 431

**Figure 15-2: Using Array DML permitted over 3,500 records to be inserted in about 119 milliseconds**

The second button that appears in this project is labeled Create And Populate Table Using Queries. The event handler for this button is shown in the following code segment. As you can see, the only difference from the Array DML version of the button is that the PopulateTableUsingQueries method, instead of

PopulateTableArrayDML, is invoked:

```

procedure TForm1.UsingQueriesButtonClick(Sender: TObject);
const
  TableName = 'Dates';
begin
  if DateTimePicker1.Date >= DateTimePicker2.Date then
    raise Exception.Create('Starting date cannot ' +
      'follow ending date');
  CreateTable(TableName);
  PopulateTableUsingQueries(TableName);
end;

```

PopulateTableUsingQueries does not employ Array DML. Instead, it uses the tried and true method of creating a parameterized query and manually binding

the new data values to these parameters for each execution of the query.

PopulateTableUsingQueries is shown in the following code segment:

432 Delphi in Depth: FireDAC

```
procedure TForm1.PopulateTableUsingQueries(TableName: string);
```

```
var
```

```
  CurrentDate: TDateTime;
```

```
  TotalDays: Integer;
```

```
  i: Integer;
```

```
  StopWatch: TStopWatch;
```

```
begin
```

```
  TotalDays := DateUtils.DaysBetween(DateTimePicker1.Date,  
  DateTimePicker2.Date);
```

```
  CurrentDate := DateTimePicker1.Date;
```

```
  //Define the parameterized query
```

```
  FDQuery1.SQL.Text := 'INSERT INTO ' + TableName +  
  ' (DateValue, DayName) ' +  
  'VALUES( :date, :name )';
```

```
  FDQuery1.Params[0].DataType := ftDate;
```

```
  FDQuery1.Params[1].DataType := ftString;
```

```
  FDQuery1.Params[1].Size := 16;
```

```
  SharedDMVcl.FDConnection.StartTransaction;
```

```
try
```

```
  StopWatch := TStopWatch.StartNew;
```

```
  for i := 0 to TotalDays - 1 do
```

```
    begin
```

```
      FDQuery1.Params[0].AsDate := DateOf(CurrentDate);
```

```
      FDQuery1.Params[1].AsString :=
```

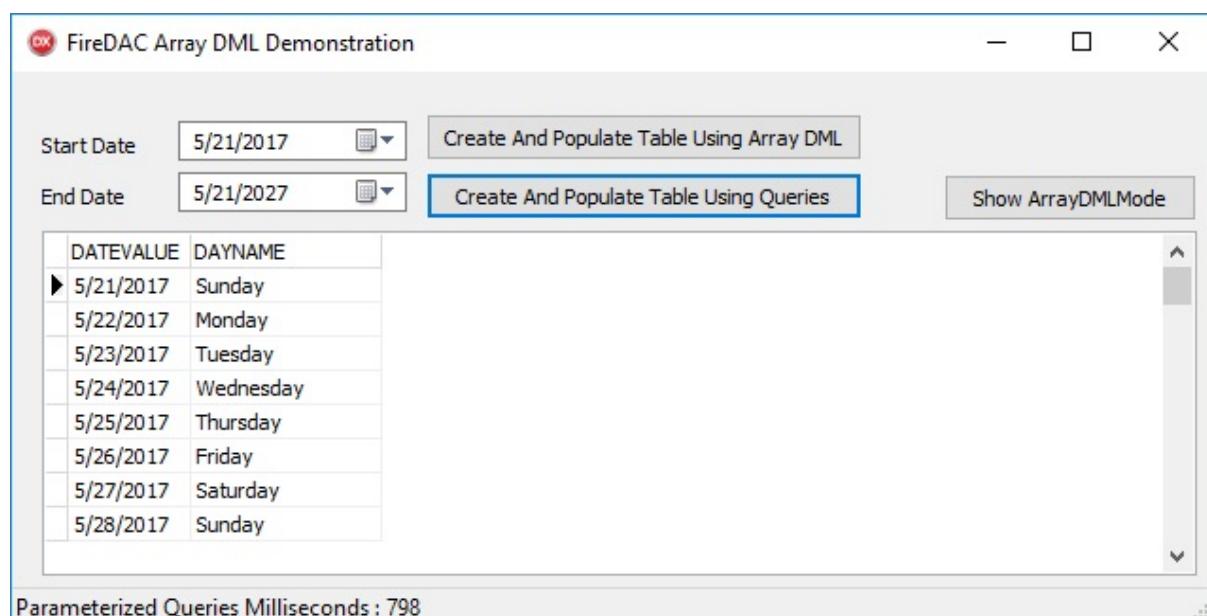
```
      FormatDateTime('dddd', CurrentDate);
```

```
      CurrentDate := IncDay(CurrentDate);
```

```

FDQuery1.ExecSQL;
end;
SharedDMVcl.FDConnection.Commit;
finally
SharedDMVcl.FDConnection.Rollback;
end;
StopWatch.Stop;
//Let's take a look at the data
FDQuery1.Open('SELECT * FROM ' + TableName);
DataSource1.DataSet := FDQuery1;
StatusBar1.SimpleText := 'Parameterized Queries Milliseconds : ' +
IntToStr(StopWatch.ElapsedMilliseconds);
end;

```



## Chapter 15: Array DML 433

Like the `PopulateTableArrayDML` method, this method uses a `TStopWatch` to measure performance. Figure 15-3 shows the results of a test run performed using traditional parameterized queries.

### Figure 15-3: More than 3,500 records were inserted using a parameterized query in about 798 milliseconds

If you compare the two preceding figures, you will notice a very dramatic

difference in performance between these two methods for inserting records.

When Array DML is used, records are inserted in about a tenth of a second. By comparison, when parameterized queries are used, it takes about three quarters of a second to perform the same insertion. In other words, Array DML is more than four times faster than prepared parameterized queries. Of course, this project uses InterBase, which natively supports batch command operations.

To be honest, I really tried to make the parameterized version as fast as possible.

I tried explicitly preparing the parameterized query in advance of execution, but that made things slower, so I removed the call to prepare. No matter what I did, the results that you see in Figure 15-3 were the best I could manage on my test machine (which was running InterBase locally. If I used a remote server I would expect the advantage of Array DML over traditional parameterized queries to be significantly greater). Nonetheless, Array DML was consistently faster, and by very significant amounts of time.

#### 434 Delphi in Depth: FireDAC

So how does Array DML do when used with a database that does not natively support batch command operations? Well, one database about which I have written a number of books, the Advantage Database Server (ADS) is one such database. I modified this project to execute against the Advantage Database Server (ADS). In those tests, Array DML showed a small and almost negligible performance benefit (530 milliseconds versus 560 milliseconds). Nonetheless, over repeated executions the Array DML method showed a slight, but consistent performance improvement over the parameterized query version.

Speaking of performance, it's worth noting that ADS is based on ISAM (Indexed Sequential Access Method) technology, which is generally faster than SQL-based servers when it comes to creating and modifying records. This

explains why the parameterized query results with ADS were faster than with InterBase. But a side note to this discussion is that it is worthwhile testing various scenarios if you are concerned about improving performance.

#### **Array DML Mode and Errors**

How FireDAC handles errors that occur during an Array DML operation is determined by the readonly `ArrayDMLMode` property, a property of the

ConnectionMetaDataIntf property of the FDConnection to which the FDQuery

is connected. This property is determined by the type and version of database server to which you are connected. There are three different ArrayDMLModes.

These are aeUpToFirstError, aeOnErrorUndoAll, and aeCollectAllErrors.

ArrayDMLMode affects how FireDAC reacts to errors encountered when the Execute method is called. This mode also determines what you can do within an OnExecuteError event handler. OnExecuteError has the following signature:

```
TFDExecuteErrorEvent = procedure (ASender: TObject; ATimes,
```

```
AOffset: LongInt; AError: EFDBEngineException;
```

```
var AAction: TFDErrorAction) of object;
```

When called, ASender is the FireDAC dataset for which Execute was called, ATimes is the size of the Array DML array (and may not match the Times parameter that you passed in your call to the Execute method), and AOffset is the index of the parameters array whose query was executing when the error was encountered.

AError is the exception that was raised, and in the case of aeCollectAllErrors, AError provides access to an array of exceptions. Finally, AAction permits you

## Chapter 15: Array DML 435

to control how Array DML proceeds. If you set AAction to eaFail, the Array DML operation will be aborted. Set AAction to eaSkip to skip this item and continue execution with AOffSet + 1. If you determine that the error was called by a bad parameter in the parameters array, you can edit the parameters at this offset in the parameters array and set AAction to eaRetry.

When ArrayDMLMode is aeUpToFirstError, FireDAC will stop executing queries upon encountering the first error. At this point, the RowsAffected property of the FireDAC dataset will indicate how many rows were successfully applied, and the AError parameter of the OnExecuteError event handler will hold the exception or exceptions associated with the one failed record. In addition, the AError.Errors[0].RowIndex property will hold the index value of the parameters array of the query that generated the first error.

An ArrayDMLMode of aeOnErrorUndoAll will cause FireDAC to stop

processing records and rollback any changes that had been applied. Once rolled back, FireDAC will restart the process using a mode similar to aeUpToFirstError. If errors are encountered in this mode, handle them as you would errors encountered during an aeUpToFirstError mode.

In the final mode, aeCollectAllErrors, FireDAC attempts to apply all records, continuing even when errors are found. At the conclusion of execution, all applied records are committed, the FireDAC dataset RowsAffected property returns the number of rows successfully applied, and you can use the AError.Errors[i].RowIndex property to determine which elements of the parameters array encountered an error. For example, the following code will display, one at a time, the offsets in the parameters array that cause errors:

```
procedure TForm1.FDQuery1ExecuteError(ASender: TObject; ATimes,
```

```
AOffset: Integer; AError: EFDBEngineException; var AAction:
```

```
TFDErrorAction);
```

```
var
```

```
i: Integer;
```

```
begin
```

```
for i := 0 to AError.ErrorCount - 1 do
```

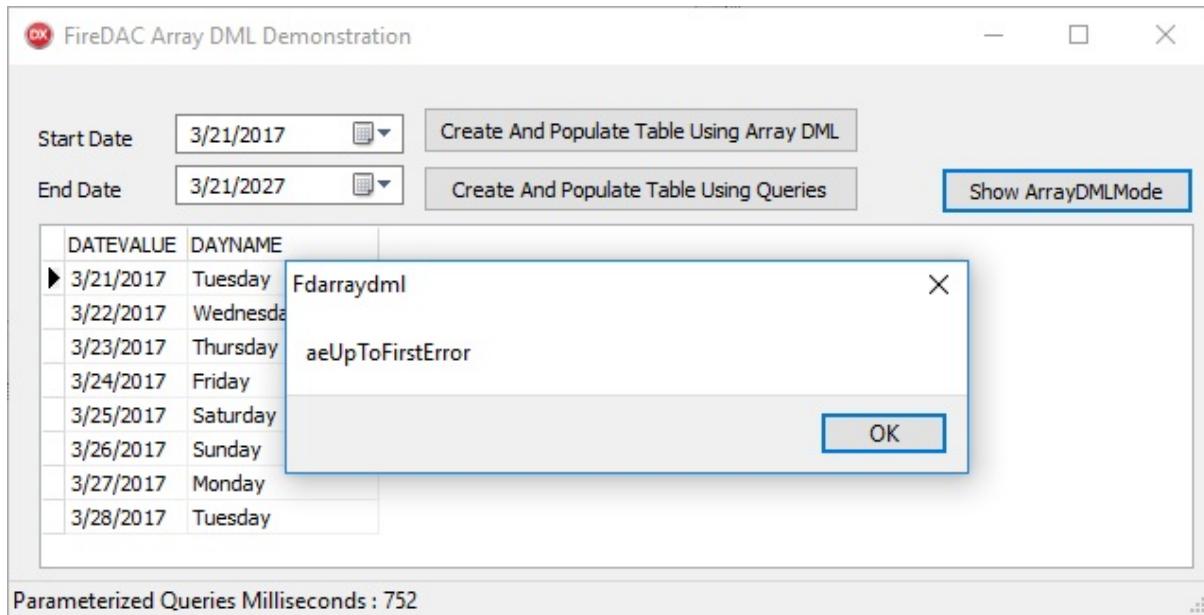
```
ShowMessage(IntToStr(AError.Errors[i].RowIndex));
```

```
end;
```

IBM DB2, Informix, and Microsoft SQL Server support the ArrayDMLMode of

aeCollectAllErrors. Oracle, Firebird (2.1 and higher), and PostgreSQL (8.1 and higher) support aeOnErrorUndoAll. The remainder support aeUpToFirstError

(with the exception of SQLite version 3.11.7 and higher which emulates batch



## 436 Delphi in Depth: FireDAC

command processing in aeUpToFirstError mode when Params.BindMode is set

to pbByName, but supports native execution for INSERT commands in aeOnErrorUndoAll mode when Params.BindMode is set to bpByNumber).

If you want to confirm the ArrayDMLMode of the database to which you are connected, you can read the ArrayExecMode property of the

ConnectionMetaDataIntf property of your connection.

ConnectionMetaDataIntf is a public property, so this is something that you are going to have to do at runtime. For example, the following event handler, which is associated with the OnClick event handler of the button labeled Show ArrayDMLMode, uses RTTI

(runtime type information) to display the human readable form of the ArrayDMLMode property, as shown in Figure 15-4:

```
procedure TForm1.btnShowArrayDMLModeClick(Sender: TObject);
```

```
begin
```

```
  ShowMessage(TypeInfo.GetTypeName(TypeInfo(TFDPhysArrayExecMode)),  
  Ord(
```

```
    SharedDMVcl.FDConnection.ConnectionMetaDataIntf.ArrayExecMode  
  )));
```

```
end;
```

## **Figure 15-4: The ArrayDMLMode value of the FDArrayDML project**

Chapter 15: Array DML 437

### **Handling Array DML Limits**

While most of FireDAC's supported databases provide native support for Array DML, most do impose limits on the amount of data/records that you can

manipulate in a single Execute call, and these limits are typically associated with available memory. If FireDAC can determine these limits, it will automatically divide the array into supportable groups. Alternatively, you can configuring the ResourceOptions.ArrayDMLSize property. When you do this, FireDAC will transparently divide a single call to Execute into multiple calls, thereby avoiding the limits imposed by individual servers.

In the following next chapter, you will find a detailed discussion of cached updates.

Chapter 16: Using Cached Updates 439

# **Chapter 16**

## **Using Cached Updates**

When enabled, cached updates hold changes made to the records of FireDAC datasets in cache. These changes can then be examined, altered, discarded, or applied to the underlying database. By comparison, when cached updates are not enabled, changes made to records in a FireDAC dataset are written back to the underlying database on a record-by-record basis.

Cached updates are a feature of all FireDAC datasets. Specifically, cached updates can be employed by FDQuery, FDStoredProc, and FDMemTable components. While cached updates can also be used by FDTable components, doing so is somewhat limited compared to the use of FDQueries. As a result, the FDQuery component is the preferred component for working with cached updates.

Cached updates permit you to perform a variety of operations that would otherwise be difficult or impossible. For example, when using cached updates, it is possible to apply changes to many records, and even many underlying tables, within a transaction, ensuring that those changes are applied in an all-or-none fashion. Cached updates also permit you to programmatically examine the one or more records that a user has changed, ensuring that those changes are meaningful and consistent, before attempting to save the data.

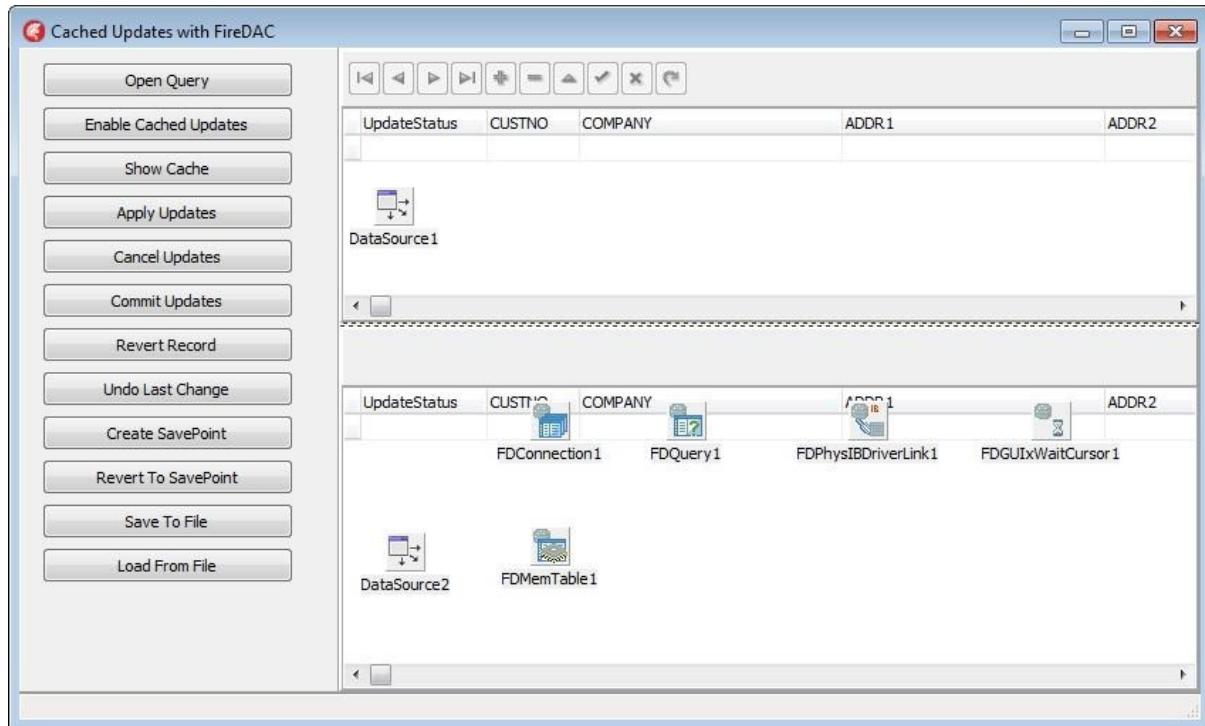
Cached updates also permit you to edit data in unconventional ways. For example, you might permit your users to edit data acquired from one or more tables. However, after the user has made their changes, you might write that data to a completely different set of tables, or even to a different database.

In addition, cached updates let you get creative when the edits contain errors that prevent the data from being written to the underlying database. For instance, if a user makes a change that violates a referential integrity rule in the database, your code can hold those changes in memory while the user makes corrections.

You can then attempt to re-apply those changes.

There is one more really big feature of cached updates. The changes made to

records can be cached over more than a single session. For example, a user can query a database and start making changes in the morning. The user may then



## 440 Delphi in Depth: FireDAC

close the application, and re-open it later in the day (or on some other day for that matter) to make more changes. At some arbitrary time in the future, the user could review all of his or her changes, and then ask to have those changes applied to the underlying database. Of course, this requires that you persist the data between editing sessions, which means either saving the data to a file or streaming it to some storage location. That process is discussed in *Chapter 11, Persisting Data*.

Most of the features of cached updates are demonstrated in the FDBasicCache project. The main form of this project is shown in Figure 16-1.

**Figure 16-1: The FDBasicCache project**

*Code: You can find the FDBasicCache project in the code download. See Appendix A for details.*

Let's start our examination of cached updates by looking at the basics.

Chapter 16: Using Cached Updates 441

### Cached Updates: The Basics

FireDAC supports two modes of cached updates: the *centralized model* and the *decentralized model*. In the centralized model, a single cache is

maintained for one or more FireDAC datasets. By comparison, in the decentralized model, each FireDAC dataset maintains its own cache.

To use the centralized model, you must add an FDSchemaAdapter to your project. Each FireDAC dataset that will participate in the caching will have its SchemaAdapter property set to this schema adapter. From that point on, many operations involving the cache, including applying it, canceling updates, and determining how many changes it contains, are performed through the schema adapter.

By comparison, if you do not use a schema adapter, you will manipulate the cache of each FireDAC dataset independently. Specifically, you must initiate cached updates and apply the resulting updates to the cache of each FireDAC dataset in separate operations.

The centralized model has the advantage that you can apply updates to two or more tables within a single transaction, and those updates will be applied in the same order in which they were logged in the cache. The centralized model is also very useful when two or more of the tables you are caching represent master-detail relationships.

Other than the use of a schema adapter required by the centralized model, most of the tasks that you perform with cached updates are the same in both models, other than those slight variations noted in the following discussions.

Consequently, in order to keep things simple, the FDBasicCache project makes use of the decentralized model. Later in this chapter, I will discuss the features specific to the centralized model of cached updates.

### **Entering and Exiting Cached Updates**

You initiate cached updates by setting the CachedUpdates property of a FireDAC dataset to True. Correspondingly, you terminate cached updates by setting the CachedUpdates property to False. When you are in the cached updates mode, the cache must be empty before you can exit cached updates.

This can be done either by applying the updates found in the cache (after which the cache must be cleared), or by canceling all updates. Also, you cannot enter or exit the cached updates mode of an FDTable that is active.

442 Delphi in Depth: FireDAC

*Note: If a FireDAC dataset is in cached updates mode, and has changes in cache, closing that dataset without applying the updates results in the loss of those cached changes.*

Entering and exiting the cached updates mode is demonstrated in the following code which is associated with the button whose initial caption reads Enable Cached Updates:

```
procedure TForm1.EnableCachedUpdatesBtnClick(Sender: TObject);
begin
  if not FDQuery1.CachedUpdates then
    begin
      EnableCachedUpdatesBtn.Caption := 'Disable Cached Updates';
      FDQuery1.CachedUpdates := True;
    end
  else //we are already in a cached updates mode
    begin
      if FDQuery1.UpdatesPending then
        begin
          ShowMessage('You must apply, cancel, or commit changes ' +
                     'to the cache before exiting the cached updates mode');
          exit;
        end;
      FDQuery1.CachedUpdates := False;
      EnableCachedUpdatesBtn.Caption := 'Enable Cached Updates';
    end;
  if FDQuery1.Active then
    FDQuery1.Refresh;
end;
```

### Detecting the Cache State

When you are in the cached updates mode, there are several properties that may be relevant to your code. For example, you may want to know if there are

changes in the cache. If there are, you may want to know how many changes are pending.

You determine whether or not there are changes in the cache by reading the

UpdatesPending property. The number of changes can be discovered by reading the ChangeCount property.

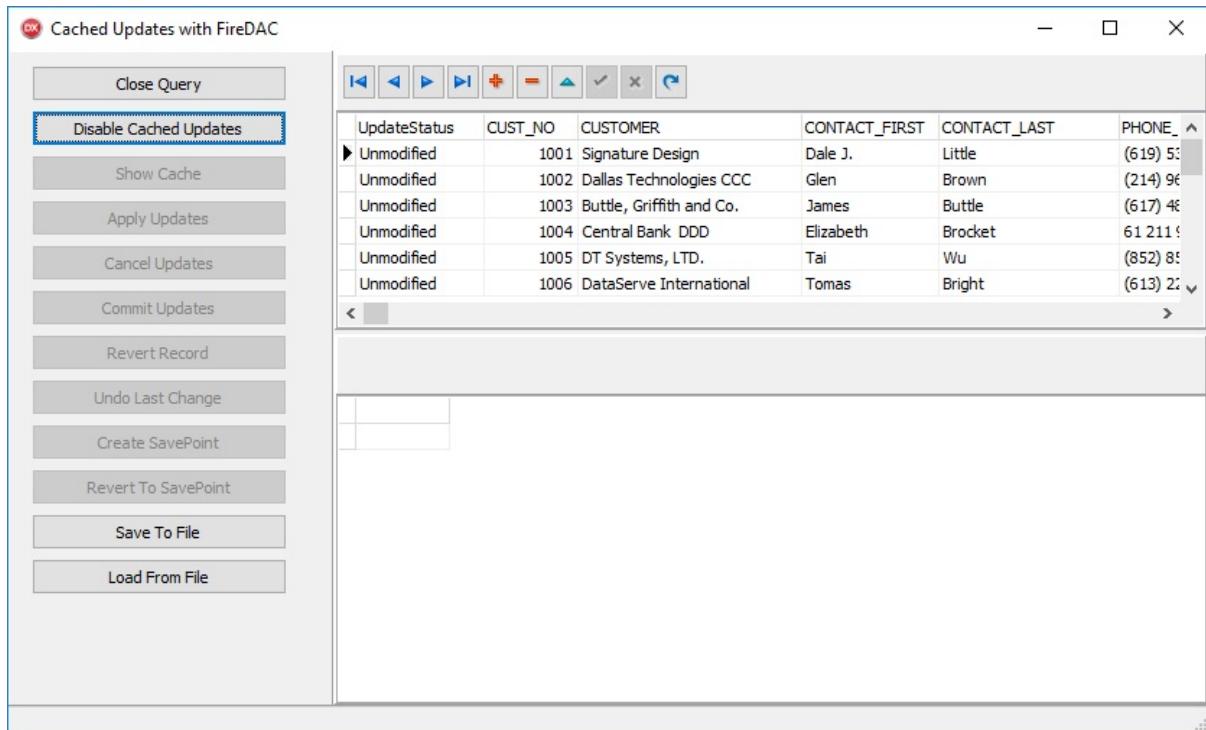
## Chapter 16: Using Cached Updates 443

The use of UpdatesPending can be seen in the OnDataChange event handler of the FDQuery's DataSource, shown in the following code segment, along with

the UpdateButtons method it invokes. The UpdateButtons method toggles the Enabled properties of the various buttons on this project's form:

```
procedure TForm1.DataSource1DataChange(Sender: TObject;  
Field: TField);  
begin  
  UpdateButtons(FDQuery1.UpdatesPending);  
  // additional code here  
end;  
procedure TForm1.UpdateButtons(const Value: Boolean);  
begin  
  ApplyUpdatesBtn.Enabled := Value;  
  ShowCacheBtn.Enabled := Value;  
  CancelUpdatesBtn.Enabled := Value;  
  CommitUpdatesBtn.Enabled := Value;  
  RevertRecordBtn.Enabled := Value;  
  UndoLastChangeBtn.Enabled := Value;  
  CreateSavePointBtn.Enabled := Value;  
end;
```

Figure 16-2 shows how the main form looks at runtime, after placing the FDQuery into the cached updates state, but before any changes have been posted.



## 444 Delphi in Depth: FireDAC

**Figure 16-2: An FDQuery is in cached updates mode, but no changes have been posted**

### Record Status and Change Filters

Another important piece of information you may need to access is related to the state of the records in cache. For example, you may want to know if a particular record has been changed or not. This is determined by reading the record's `UpdateStatus` property, which returns the status of the current record.

`UpdateStatus` is a `TUpdateStatus` type property. Here is the declaration of the `TUpdateStatus` enumeration from the `Data.DB` unit:

```
TUpdateStatus = (usUnmodified, usModified,  
usInserted, usDeleted);
```

There is a calculated field in `FDQuery1`, and this field is used to display the status of individual records on the form. The following is the `OnCalcFields` event handler for this dataset, which uses `UpdateStatus` to determine each record's status:

```
procedure TForm1.FDQuery1CalcFields(DataSet: TDataSet);  
begin
```

```
case DataSet.UpdateStatus of
  usUnmodified:
    DataSet.Fields[DataSet.FieldCount -1].AsString := 
      'Unmodified';
  usModified:
    DataSet.Fields[DataSet.FieldCount -1].AsString := 
      'Modified';
  usInserted:
    DataSet.Fields[DataSet.FieldCount -1].AsString := 
      'Inserted';
  usDeleted:
    DataSet.Fields[DataSet.FieldCount -1].AsString := 
      'Deleted';
end;
end;
```

If you are not already familiar with cached updates, you may have found something in the preceding code puzzling. Specifically, how it is possible to display a calculated field for a deleted record? I mean, it has been deleted, right, so it no longer exists. Well, that's one of the cool things about cached updates.

The record is marked for deletion, and will be deleted from the database if the updates are successfully applied, but the record still exists so long as it is in cache.

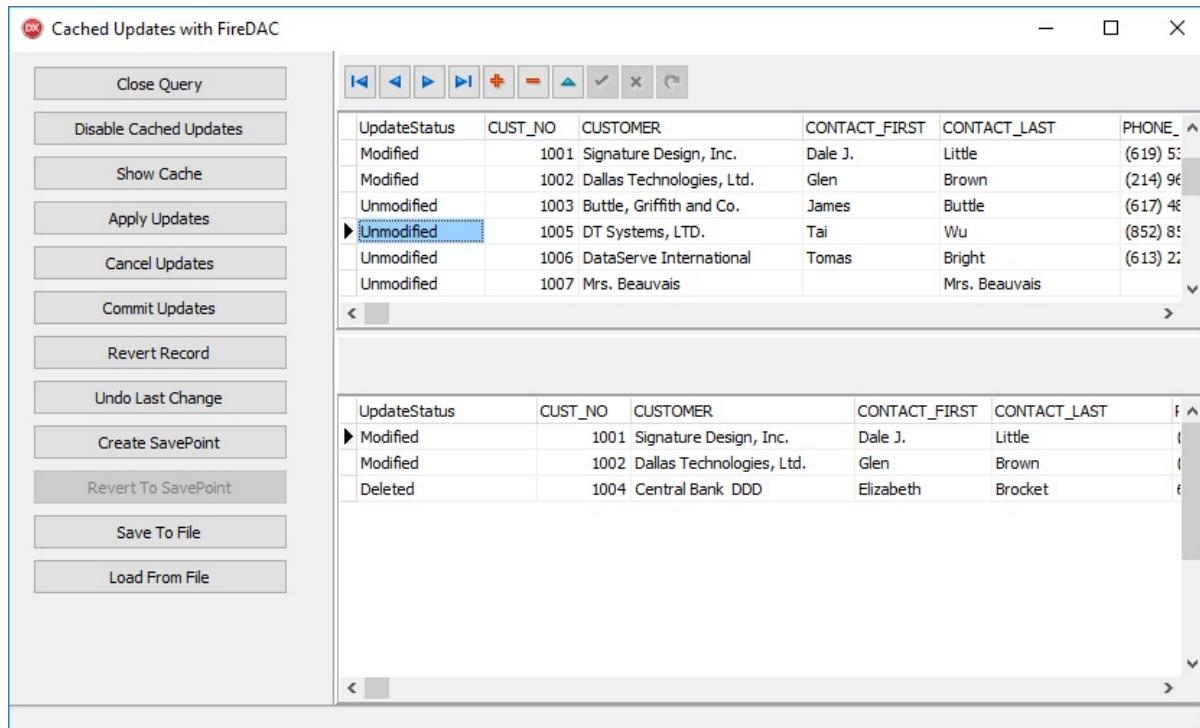
Do you want to see which records were deleted before cached updates are applied? It is possible by changing the FilterChanges property of the FireDAC dataset. This property is of the TFDUpdateRecordTypes type. The following is the declaration of TFDUpdateRecordTypes:

```
TFDUpdateRecordTypes = set of (rtModified, rtInserted,
  rtDeleted, rtUnmodified, rtHasErrors);
```

The default value of this property is the following set:

```
[ rtModified, rtUnmodified, rtInserted ]
```

When this property contains the rtDeleted flag, those records that have been deleted will be present in the dataset, along with other records associated with any other flags in the FilterChanges property, and will be visible if the dataset is being displayed using a data-aware control. The following code uses the FilterChanges property of an FDMemTable to display only those records that have been modified, inserted, or deleted:



## 446 Delphi in Depth: FireDAC

```
procedure TForm1.ShowCacheBtnClick(Sender: TObject);
begin
  if FDQuery1.Active then
    begin
      if FDMemTable1.Active then
        FDMemTable1.Close;
      FDMemTable1.CloneCursor(FDQuery1, True);
      FDMemTable1.FilterChanges :=
        [rtModified, rtInserted, rtDeleted];
    end;
  end;
```

Figure 16-3 shows the calculated field displaying the change type, as well as a

filtered FDMemTable based on FDQuery1 that displays modified, inserted, and deleted records.

**Figure 16-3: The FilterChanges property permits you to see your changes, including deleted records**

Chapter 16: Using Cached Updates 447

### What Has Changed?

Knowing that there are cached changes is important, but in most cases, of equal importance is knowing *what* has changed. Fortunately, a feature of Fields permits you to ask that question. Specifically, there are four properties of Fields that are useful for this purpose. They are OldValue, CurValue, NewValue, and Value, and they are all variant types. And, with the exception of the Value property, these properties only make sense when your dataset is in the cached updates mode.

When a record has been modified, OldValue contains the original value that was loaded, and CurValue has the current value. If a record was inserted, the

NewValue property of fields in the newly created record will contain the values that were inserted. The Value property reflects the current contents of the field, whether it was modified, inserted, deleted, or unmodified.

An example of using the OldValue and CurValue properties to detect field-level changes is shown in the OnDataChange event handler of the DataSource that

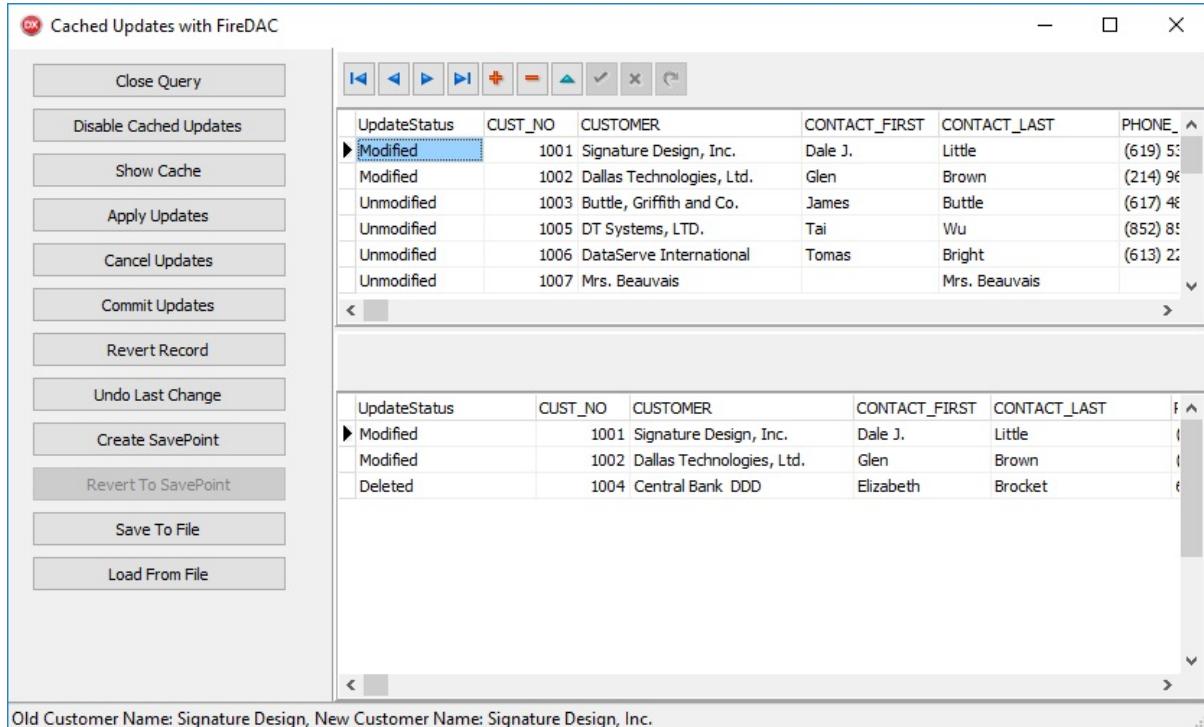
points to the FDQuery. The following is the complete contents of this event handler:

```
procedure TForm1.DataSource1DataChange(Sender: TObject;  
Field: TField);  
begin  
  UpdateButtons(FDQuery1.UpdatesPending);  
  if (FDQuery1.UpdateStatus = usModified) then  
    StatusBar1.SimpleText := 'Old Customer Name: ' +  
    FDQuery1.FieldByName('Customer').OldValue +  
    ', New Customer Name: ' +  
    FDQuery1.FieldByName('Customer').CurValue  
else
```

```
StatusBar1.SimpleText := ";
```

```
end;
```

Figure 16-4 depicts the FDBasicCache project with some changes, and a modified record selected in the FDQuery. The original and current values of the Customer field are displayed in the status bar.



## 448 Delphi in Depth: FireDAC

**Figure 16-4: OldValue, NewValue, and CurValue Field properties permit you to detect what changes have been posted**

### Persisting the Cache

All FireDAC datasets support persistence. Specifically, you can call `SaveToFile` or `SaveToStream` to write the data held by the dataset to a file or stream, respectively. If the dataset you are persisting includes cached data, the cache is persisted as well, by default. For a detailed discussion of FireDAC dataset persistence, see *Chapter 11, Persisting Data*.

You restore a previously persisted dataset by calling `LoadFromFile` or `LoadFromStream`. If the persisted data included cache data, the cache is also restored to its original state, by default. This is a very powerful feature that is used in applications that employ what is called a *briefcase model*.

The following code demonstrates persistence in FireDAC datasets. These two event handlers are associated with the buttons labeled `Save To File` and `Load`

From File, respectively:

```
procedure TForm1.SaveToFileBtnClick(Sender: TObject);
begin
  FDQuery1.SaveToFile(ExtractFilePath(ParamStr(0)) + 'file.xml',
  TFDStorageFormat.sfXML);
Chapter 16: Using Cached Updates 449
end;

procedure TForm1.LoadFromFileBtnClick(Sender: TObject);
begin
  if IOUtils.TFile.Exists(ExtractFilePath(ParamStr(0)) +
  'file.xml') then
    begin
      //disconnect the query from the DataSource
      FDQuery1.DisableControls;
      try
        FDQuery1.LoadFromFile(ExtractFilePath(ParamStr(0)) +
        'file.xml', TFDStorageFormat.sfXML);
      finally
        FDQuery1.EnableControls;
      end;
    end;
  end;
```

It is worth noting that once EnableControls is called for the FDQuery, the DataSource's OnDataChange event handler triggers. If the previously saved data included cached records, the call to UpdateButtons found in the OnDataChange event handler causes the buttons associated with manipulating the cache to be enabled once more.

### **Managing the Cache**

One of the most important characteristics of cached data is that you can manipulate it programmatically. For example, you can choose to remove some or all of the changes from the cache. It is even possible to clear the

cache without losing the user's changes to the data. The following sections describe how to programmatically manipulate the cache.

## **Canceling the Last Change**

You cancel the most recent change to the cache by calling `UndoLastChange`. If you need to, you can call `UndoLastChange` repeatedly to systematically unwind the changes in the cache.

Using `UndoLastChange` is shown in the following event handler:

```
procedure TForm1.UndoLastChangeBtnClick(Sender: TObject);  
begin  
  FDQuery1.UndoLastChange(True);  
  450 Delphi in Depth: FireDAC  
end;
```

As you can see, the `UndoLastChange` method takes a single Boolean parameter.

When `True` is passed, and undoing the last change changes the position of the current record, the current record will remain the current record, and the cursor will shift position within the dataset. If `False` is passed, the current record may appear to fly away if the restored value requires the record to appear in a different location within the dataset.

## **Canceling a Specific Change**

While `UndoLastChange` implements a LIFO (last in, first out) type operation, it is possible to undo the change to any record in cache by calling `RevertRecord`.

For example, you can use `FilterChanges` to display only deleted records, and then restore any given deleted record by making it the current record and then calling `RevertRecord`.

The use of `RevertRecord` is shown in the following event handler. So long as the current record has been changed in some way, this event handler will restore it to its original state (an inserted record will be removed):

```
procedure TForm1.RevertRecordBtnClick(Sender: TObject);  
begin  
  if FDQuery1.UpdateStatus <> usUnmodified then  
    FDQuery1.RevertRecord;
```

**end;**

## **Canceling All Updates**

To cancel all changes to the cache, restoring the dataset to its originally loaded state (or to its state when ApplyChanges was last called), call CancelUpdates.

The following event handler demonstrates the use of CancelUpdates. (Recall that this button is only enabled so long as there is at least one change in the cache.):

```
procedure TForm1.CancelUpdatesBtnClick(Sender: TObject);  
begin  
  FDQuery1.CancelUpdates;  
end;
```

Chapter 16: Using Cached Updates 451

## **Clearing the Cache Without Canceling Changes**

Calling CommitUpdates empties the cache and marks all records as unchanged.

CommitUpdates is often called after a call to ApplyChanges has caused all changes in the cache to be successfully applied to the underlying database. However, calling CommitUpdates to flush the cache without calling ApplyUpdates can also be a useful technique.

*Note: Calling CommitUpdates is not necessary when the AutoCommitUpdates property of the FireDAC dataset's UpdateOptions property is set to True.*

*AutoCommitUpdates is available only in Delphi 10 Seattle and later.*

Here is an example of where you might want to use CommitUpdates without first calling ApplyUpdates. Imagine that your FDQuery holds the result of a multi-table join. After permitting the user to edit these records, the user clicks a button to update the underlying data. Since the query included a join, and only you, as the database designer, really know how to write the changes to the database, you take matters into your own hands. Specifically, you filter, in turn, on inserted, deleted, and updated records, executing custom SQL statements to write the corresponding changes to the appropriate tables.

Having now

completed your manual updates, you call CommitUpdates to flush the cache

(since all of the updates have been applied). I have a name for this technique. I call it the *brute force method*.

Here is another example. Imagine that you use an FDMemTable to load data from a file, permit the user to edit that data, and when the user is done editing, you write the data back to the file. While the user is editing data, you might enable cached updates so that the user can edit the cache before writing the data back to the file. When the user is ready to save those changes back to the file, you call CommitUpdates to make the user's edits permanent before writing to the file. Calling CommitUpdates flushes the cache, and the resulting file is smaller, since it contains no cached changes (Delta).

The following event handler demonstrates an example of flushing the cache without first calling ApplyUpdates:

```
procedure TForm1.CommitUpdatesBtnClick(Sender: TObject);  
begin  
  FDQuery1.CommitUpdates;  
end;
```

452 Delphi in Depth: FireDAC

## Using Save Points

A save point is an integer property that represents a position in the change cache log. You can store the value of a FireDAC dataset's SavePoint property to store a position in the change cache to which you might want to return. You return to that position by setting the dataset's SavePoint property to the previously saved integer.

Here is how this might work. After deleting four records, the user might record a save point (by clicking a button you provide) so that they could easily restore the cache to its current state.

The user can now continue editing the data. If they subsequently decide that they want to cancel all of the changes that they made since saving the save point, they can click another button you provide to restore the cache to the state it was in when the save point was saved. In other words, clicking the restore button would return the cache to the state where only four record deletions were recorded.

Save points are only valid so long as the integrity of the cache prior to the save point is intact. For example, if the user deleted four records, created a save point, made some additional edits, including restoring one of the deleted

records, it would not be possible to restore to the save point.

The following event handlers demonstrate creating and restoring a save point:

```
procedure TForm1.CreateSavePointBtnClick(Sender: TObject);
```

```
begin
```

```
  FSavePoint := FDQuery1.SavePoint;
```

```
  RevertToSavePointBtn.Enabled := True;
```

```
end;
```

```
procedure TForm1.RevertToSavePointBtnClick(Sender: TObject);
```

```
begin
```

```
  DataSource1.DataSet := nil;
```

```
  FDQuery1.SavePoint := FSavePoint;
```

```
  DataSource1.DataSet := FDQuery1;
```

```
  FSavePoint := 0;
```

```
  RevertToSavePointBtn.Enabled := False;
```

```
end;
```

Chapter 16: Using Cached Updates 453

## Applying Updates

One of the primary reasons for caching updates is to perform a batch application of those edits to an underlying database. While that might sound obvious, it is worth pointing out that cached updates don't always result in the application of changes to the data. Sometimes the changes are discarded because the user

realized that they were working on the wrong set of records. In other circumstances, your use of cache updates might be to observe a user's editing behavior without any intention of saving those edits. Granted, this latter scenario may be rare, but it's not out of the question.

Having made those qualifications, it is true that in most cases the end result of a cached updates session is to attempt to write the cached changes to a database.

And there are two general approaches to doing this: The brute force method and calling the ApplyUpdates method.

### Apply Updates Using Brute Force

I mentioned the brute force method in passing earlier in this chapter when introducing the CommitUpdates method. The brute force method involves systematically filtering the cache on the changes it contains, and manually writing the changes to the underlying database.

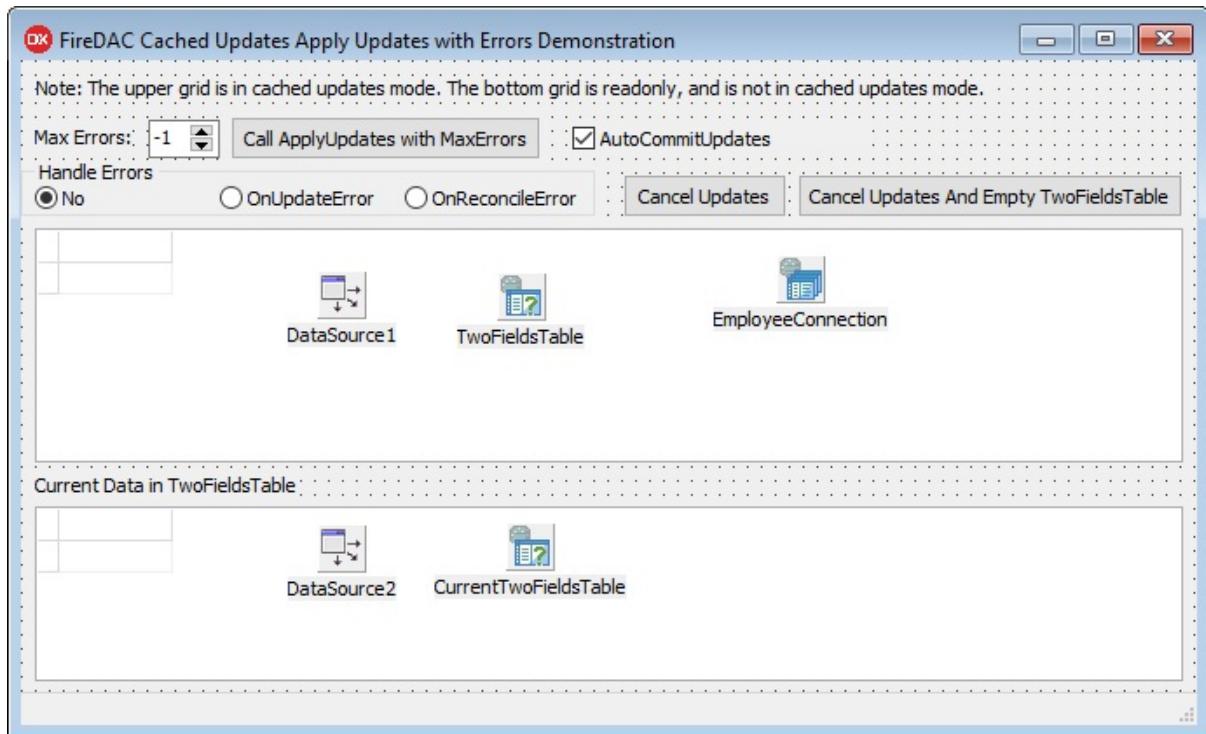
For example, you might first filter the cache on inserted records, and then execute one SQL INSERT statement for each record in the filtered set. Next, you could filter on deleted records, issuing one SQL DELETE statement for each of those records. Finally, you would filter on modified records. After doing so, you would use the OldValue and CurValue properties of the underlying

TFields to determine which fields were changed, and then construct the appropriate SQL UPDATE statements to apply those changes. When you are done updating the data, you would call CommitUpdates to flush the cache and mark all of the records in memory as unchanged (or call CancelUpdates and then refresh the datasets).

The brute force method requires a significant amount of code, and you are responsible for handling errors if they arise. As a result, the brute force method is typically employed when the use of the ApplyUpdates method is impractical or impossible.

### **Calling the ApplyUpdates Method**

When you call ApplyUpdates, FireDAC examines the contents of the cache and attempts to apply each of the changes, one at a time. When using ApplyUpdates there are two modes: Automatic updates, where you permit FireDAC to generate



## 454 Delphi in Depth: FireDAC

and execute the queries that will apply the updates, and the manual mode, where you write an `OnUpdateRecord` event handler, where your code applies the

updates, but which is called by FireDAC, once for each update in cache that needs to be applied.

I am going to begin this section with a discussion of the process of calling `ApplyUpdates`, and I will cover the use of the `OnUpdateRecord` event handler later in this section.

Since handling errors during the call to `ApplyUpdates` is more complicated than when a single record is being updated, I have created a project named `FDCachedUpdatesErrors`. This project can help you to better understand the process of calling `ApplyUpdates`, and what happens when there are errors in the cache. The main form for this project is shown in Figure 16-5.

**Figure 16-5: The main form of the `FDCachedUpdatesErrors`**

*Code: The `FDCachedUpdatesErrors` project is included in the code download.*

Because this project needs to support both successful posts as well as failures, I have created a very simple table in the `employee.gdb` database. This table is

named TwoFieldsTable, and it contains two fields, both of type VarChar(30), and both required. While FireDAC will normally enforce a required fields constraint before accepting posts to cache, I have disabled that operation, by setting the UpdateOptions.CheckRequired property of the FDQuery that employs the cached updates mode in this project to False.

I have a second query in this project which does not use cached updates. This query is used to display the current contents of the TwoFieldsTable table in the lower DBGrid in on the form shown in Figure 16-5.

The creation of the TwoFieldsTable table (when necessary), and the basic configuration of the TwoFieldsTable and CurrentTwoFieldsTable FDQueries are shown in the following code segment, which is executed each time the project is run:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  FDQuery: TFDQuery;
const
  TabName = ‘TWOFIELDSTABLE’;
begin
  EmployeeConnection.Open();
  FDQuery := TFDQuery.Create( nil );
  try
    FDQuery.Connection := EmployeeConnection;
    FDQuery.SQL.Text := ‘SELECT RDB$RELATION_NAME ‘ +
      ‘ FROM RDB$RELATIONS ‘ +
      ‘ WHERE RDB$RELATION_NAME = ”’ +
      ‘
      TabName + ””;’;
    FDQuery.Open();
    if FDQuery.RecordCount = 0 then
      begin
```

```
EmployeeConnection.StartTransaction;

try
  FDQuery.ExecSQL('CREATE TABLE "'+ TabName + '" (' +
    ' "FIRST" VARCHAR(30) NOT NULL, ' +
    ' "LAST" VARCHAR(30) NOT NULL );');

  EmployeeConnection.Commit;

except
  EmployeeConnection.Rollback;
  raise;
end;
end;

FDQuery.Close;

456 Delphi in Depth: FireDAC

finally
  FDQuery.Free;
end;

TwoFieldsTable.SQL.Text := 'SELECT * FROM ' + TabName;
{$IF CompilerVersion >= 30.0} // Delphi 10 Seattle or later
TwoFieldsTable.UpdateOptions.AutoCommitUpdates := True;
{$ENDIF}

TwoFieldsTable.UpdateOptions.CheckRequired := False;
TwoFieldsTable.UpdateOptions.CheckReadOnly := False;
TwoFieldsTable.UpdateOptions.CheckUpdatable := False;
TwoFieldsTable.CachedUpdates := True;
TwoFieldsTable.Open;
{$IF CompilerVersion < 30}
  cbxAutoCommitUpdates.Enabled := False;
{$ELSE} // Delphi 10 Seattle or later
  TwoFieldsTable.UpdateOptions.AutoCommitUpdates :=
```

```
cbxAutoCommitUpdates.Checked;  
{$ENDIF}  
CurrentTwoFieldsTable.Open('SELECT * FROM ' + TabName );  
end;
```

In addition to the button that you press to apply updates to the TwoFieldsTable FDQuery, there are two additional buttons that permit you to reset the form to its initial state, which I found useful while testing various forms of the ApplyUpdates method. The first button, labeled Cancel Updates, clears the

current contents of the cache by calling the CancelUpdates method. The second button, labeled Cancel Updates And Empty TwoFieldsTable, both clears the

cache and deletes all of the current records from the TwoFieldsTable table in employee.gdb. The event handlers for these two buttons are shown here:

```
procedure TForm1.btnCancelUpdatesClick(Sender: TObject);  
begin  
TwoFieldsTable.FilterChanges :=  
[ rtUnmodified, rtModified, rtInserted,  
rtDeleted, rtHasErrors ];  
TwoFieldsTable.CancelUpdates;  
TwoFieldsTable.Refresh;  
CurrentTwoFieldsTable.Refresh;  
StatusBar1.SimpleText := 'The cache was canceled';  
end;  
procedure TForm1.btnCancelUpdatesAndEmptyTwoFieldsTableClick(  
Sender: TObject);  
Chapter 16: Using Cached Updates 457  
begin  
TwoFieldsTable.CancelUpdates;  
EmployeeConnection.StartTransaction;  
try
```

```
EmployeeConnection.ExecSQL('DELETE FROM TWOFIELDSTABLE');
EmployeeConnection.Commit;
except
EmployeeConnection.Rollback;
raise;
end;
TwoFieldsTable.FilterChanges := [ rtUnmodified, rtModified,
rtInserted, rtDeleted, rtHasErrors ];
TwoFieldsTable.Refresh;
CurrentTwoFieldsTable.Refresh;
StatusBar1.SimpleText := 'The cache was canceled and ' +
'TwoFieldsTable was emptied';
end;
```

Let's now take a look at how FireDAC applies updates automatically.

## **THE APPLYUPDATES METHOD IN AUTOMATIC MODE**

When you are not employing an OnUpdateRecord event handler, calling ApplyUpdates will cause FireDAC to examine the cache. If updates are pending, and assuming that FireDAC can determine which table your changes need to be applied to, it will generate one query for each change in cache, and will invoke that query.

What happens to the records in cache depends on three factors. The first is whether or not errors occur. When an error occurs, it will necessarily remain in cache. The second factor is the value of the AMaxErrors parameter of the ApplyUpdates method. The final factor is the value of the

UpdateOptions.AutoCommitUpdates property of the dataset for which you are

calling ApplyUpdates. This property, which was first introduced in Delphi 10 Seattle, interacts with the AMaxErrors property.

I'm getting a little ahead of myself, since I haven't said much about ApplyUpdates yet, but I consider the AutoCommitUpdates property to be very important. Prior to the release of Delphi 10 Seattle, a call to

ApplyUpdates needed to be followed by a call to CommitUpdates in order to remove records from the cache (which was necessary before you could turn cached updates off).

The problem was that if some records were applied, and some failed, you were left with all records in cache, including those that were applied. Furthermore,

## 458 Delphi in Depth: FireDAC

the records that were actually applied were still marked as modified (or inserted or deleted) in cache. This made dealing with errors very difficult.

When AutoCommitUpdates is set to True (the default is False, so you must set AutoCommitUpdates to True), records that are successfully applied are removed from the cache (assuming that the call to ApplyUpdates has not been aborted due to excessive errors. I'll come back to this issue). This behavior, which is similar to how the ClientDataSet operates when its ApplyUpdates method is

called, is highly desirable. It is so desirable that I strongly recommend that if you are using Delphi XE8 or earlier, you upgrade your copy of Delphi so that you can make use of FireDAC's cached updates capabilities.

In the following discussion, I am going to assume that you are using Delphi 10

Seattle or later, and have set AutoCommitUpdates to True. The easiest way to do this is to change this property to True in your FDConnection, or better yet, in your FDManger. If you do not do this, or you cannot upgrade to Delphi 10 Seattle or later, you should use the FDCCachedUpdatesErrors project, or an adaptation of it, in order to completely understand how to successfully manage the cache when errors are present during a call to ApplyUpdates.

Ok, now that we've got that out of the way, let's talk about ApplyUpdates. This method has the following signature:

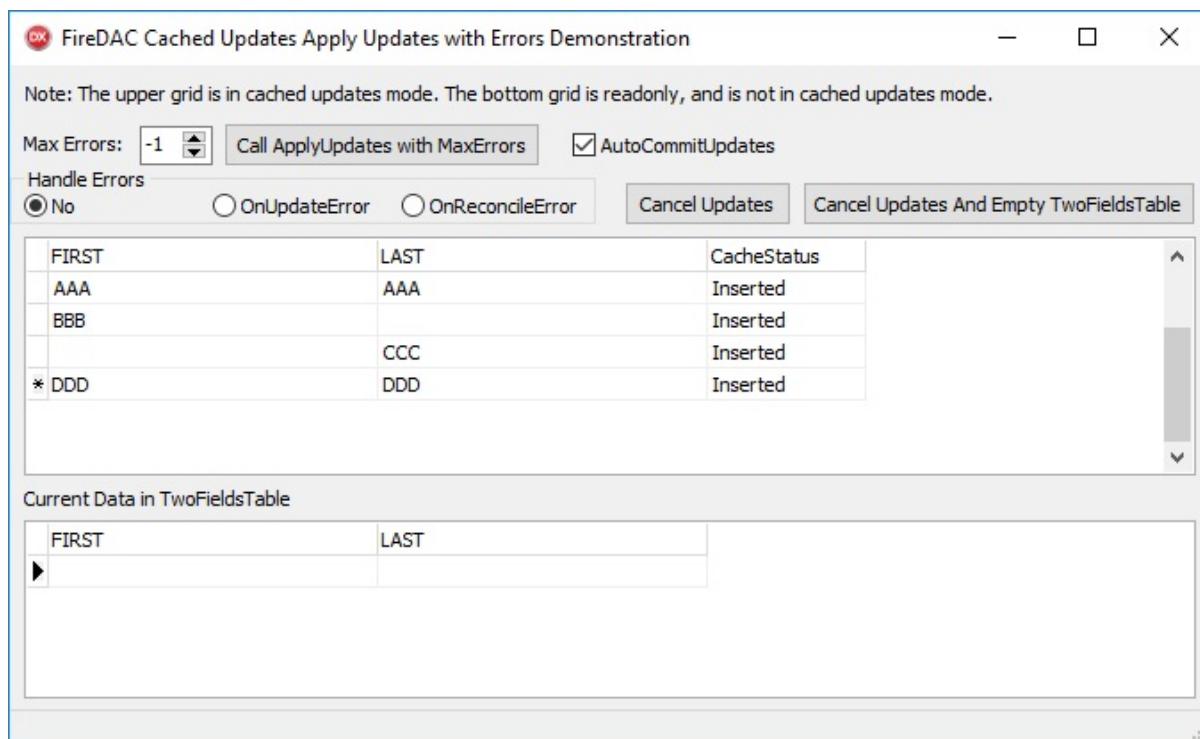
**function** ApplyUpdates(AMaxErrors: Integer = -1): Integer;

You call ApplyUpdates either on an individual FireDAC dataset (the decentralized model) or on an FDSCHEMAApapter (the centralized model). When you call ApplyUpdates, you either pass a single Integer parameter whose formal parameter name is AMaxErrors, or pass no parameters, in which case

AMaxErrors defaults to -1. ApplyUpdates returns an Integer result, which

represents the number of errors encountered during the attempt to apply the updates. Note that this number of errors may not necessarily reflect the total number of errors that exist in cache, since in some cases, as I will describe shortly, FireDAC will abort the update process before attempting to update all of the records in cache.

AMaxErrors signals your tolerance for update errors. When you pass -1 in AMaxErrors, you are telling FireDAC to attempt to update all changes in the cache no matter how many errors it encounters along the way. All records that are applied are removed from cache and their UpdateStatus is changed to usUnmodified.



## Chapter 16: Using Cached Updates 459

*Note: When I say all applied updates are removed from cache, I again am referring to Delphi 10 Seattle and later with the*

*UpdateOptions.AutoCommitUpdates property set to True. Since this assumption is made for the remainder of the discussions in this chapter, I am not going to mention it again.*

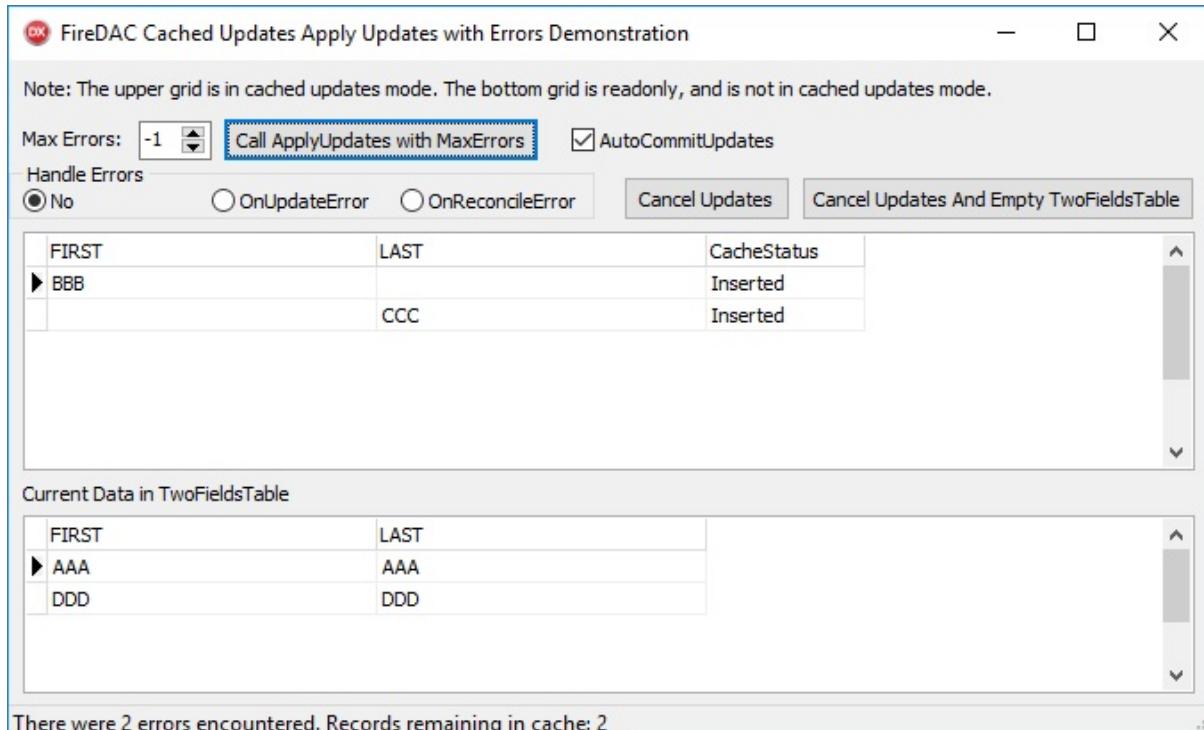
Imagine that the TwoFieldsTable table of the employee.gdb database is currently empty, and I have entered four records into the

FDCachedUpdatesErrors project, as shown in Figure 16-6. Two of these records are valid (both fields have values), and two will be rejected due to null values in at least one field.

**Figure 16-6: Two records are valid and two will fail to post**

Here is the OnClick event handler associated with the button labeled Call ApplyUpdates with MaxErrors:

```
procedure TForm1.btnAddUpdatesClick(Sender: TObject);  
var  
  NumErrors: Integer;
```



## 460 Delphi in Depth: FireDAC

```
begin  
if TwoFieldsTable.State in dsEditMode then  
  TwoFieldsTable.Post;  
  
  NumErrors := TwoFieldsTable.ApplyUpdates(  
    StrToInt(edtMaxErrors.Text));  
  
  TwoFieldsTable.FilterChanges := [ rtModified, rtInserted,  
    rtDeleted, rtHasErrors ];  
  
  CurrentTwoFieldsTable.Refresh;  
  
  StatusBar1.SimpleText := 'There were ' + NumErrors.ToString + '  
  ' errors encountered. Records remaining in cache: ' +
```

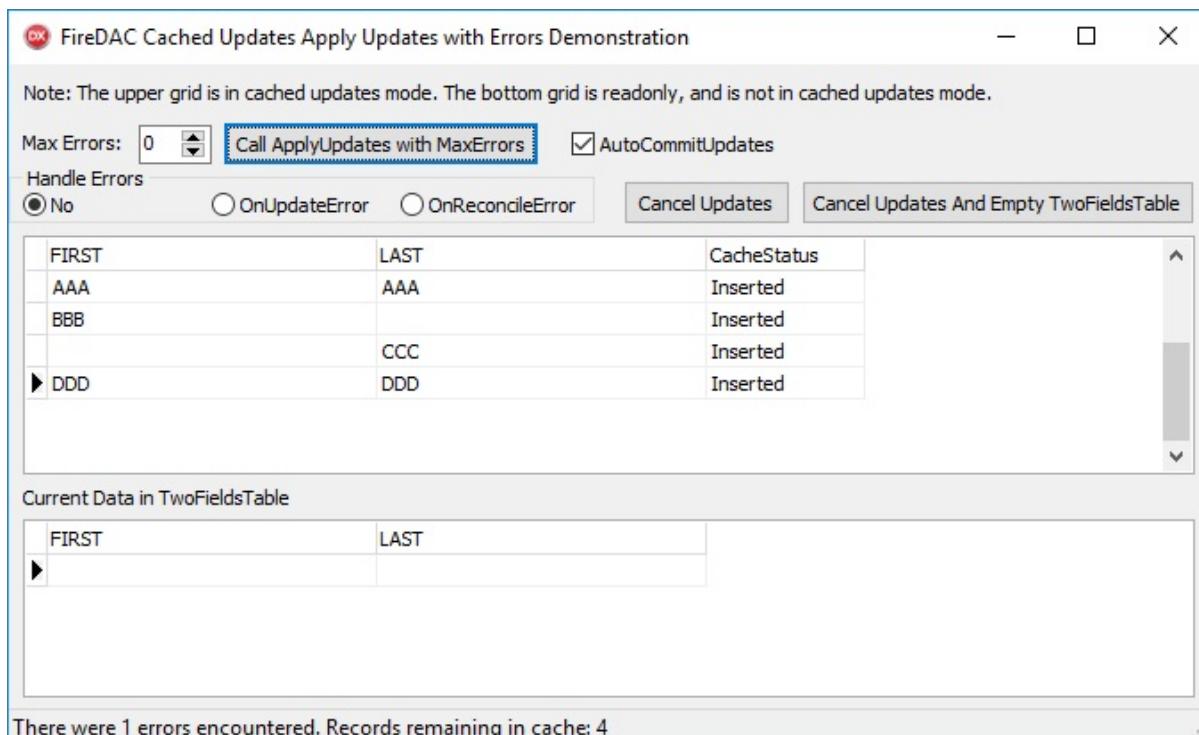
```
TwoFieldsTable.ChangeCount.ToString;
```

```
end;
```

When you click the button labeled Call ApplyUpdates with MaxErrors, and you have set MaxErrors to -1, all successfully applied records are removed from cache, and the unsuccessfully applied records remain, as shown in Figure 16-7.

### **Figure 16-7: Two records have been committed and removed from cache, and two records remain in cache**

When you pass a value of 0 to AMaxErrors, you are signaling no tolerance for errors. Under this condition, if even one error is encountered, FireDAC



### Chapter 16: Using Cached Updates 461

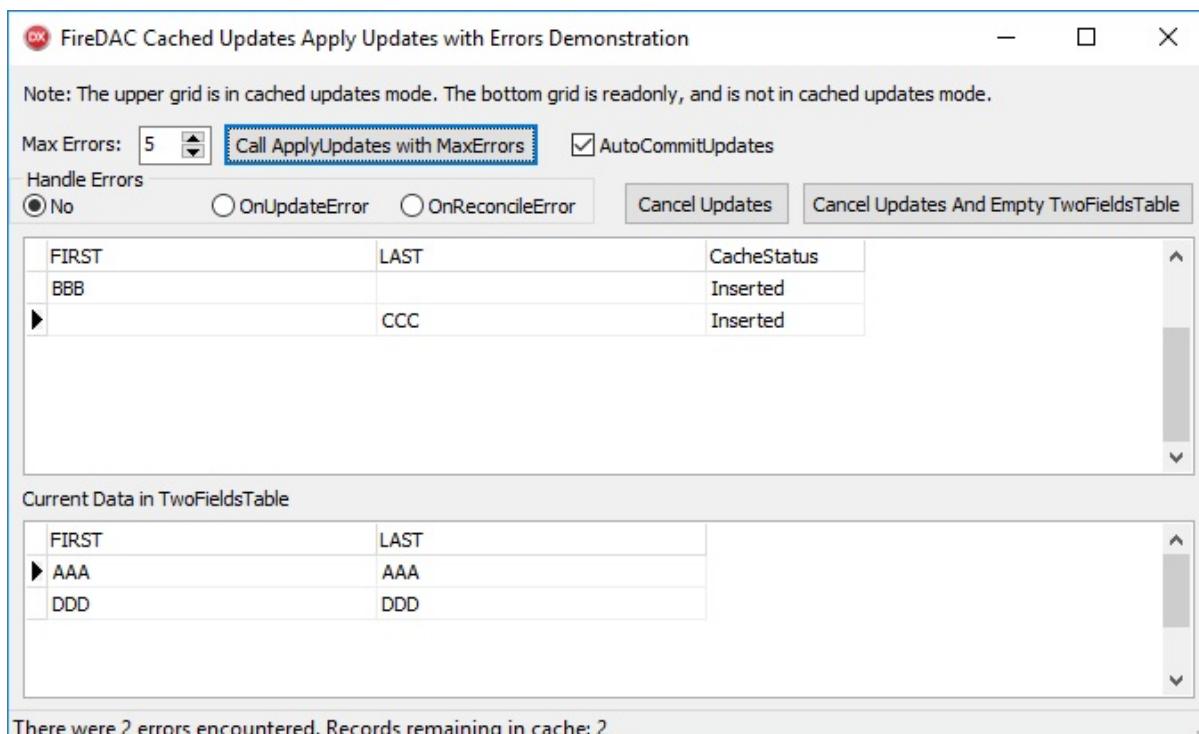
terminates the update process immediately, the cache is restored to its pre-ApplyUpdates state (all inserts, deletes, and modifications remain intact), and no updates are applied to the underlying database. Assuming that

FDCachedUpdatesErrors looked like it did in Figure 16-6, with the exception that MaxErrors was set to 0, clicking the apply updates button would produce the result shown in Figure 16-8.

### **Figure 16-8: With MaxErrors of 0, even one failure will result in no updates, one error (the first record to fail), and the cache will remain unchanged**

Passing a positive integer in the AMaxErrors parameter instructs FireDAC to accept some errors, up to the maximum number identified in AMaxErrors. For example, calling ApplyUpdates with an AMaxErrors of 5 instructs FireDAC to continue to apply updates, so long as no more than five errors are encountered.

If five errors or less are encountered, the ApplyUpdates call will complete. In that case, the successfully applied records are removed from cache (their UpdateStatus is changed to usUnmodified), and the remaining records in cache are all associated with errors. If more than five errors are encountered, the update process terminates upon the sixth error, and the cache will be restored to



## 462 Delphi in Depth: FireDAC

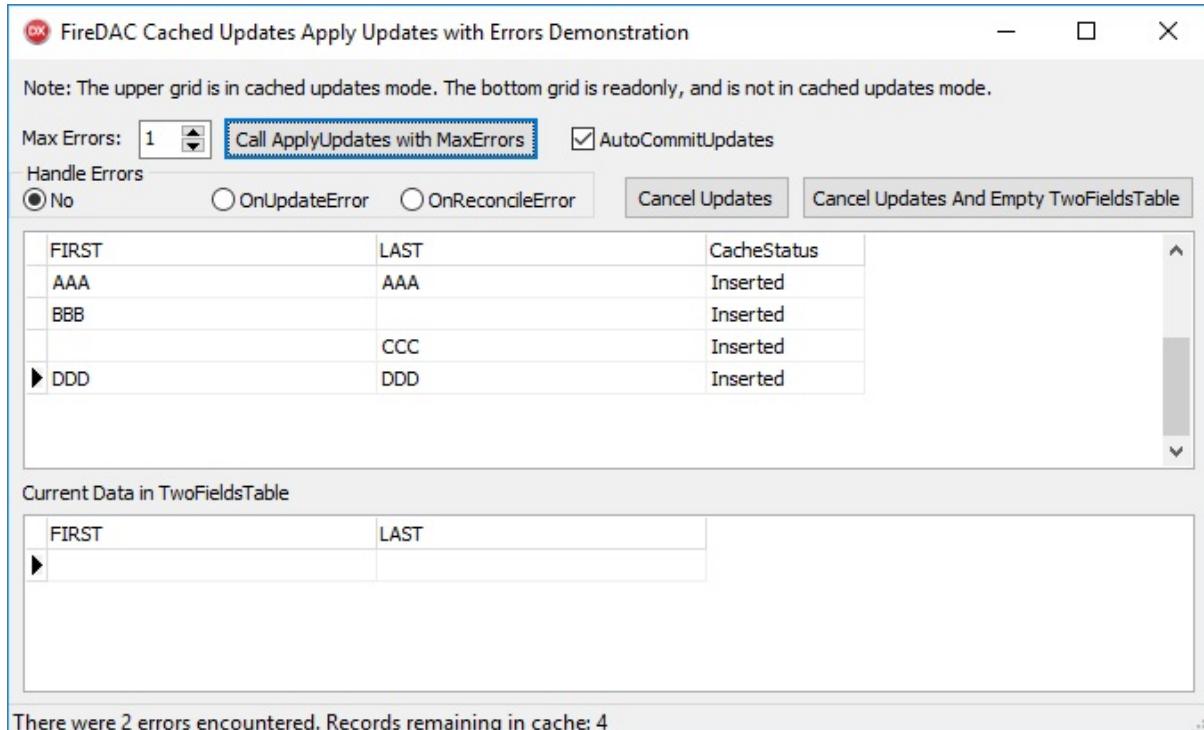
its pre-ApplyUpdates state, and no updates will have been written to the underlying database.

The successful application of records when no more than MaxErrors are encountered is demonstrated in Figure 16-9. Here we set MaxErrors to 5. Since there are only two errors, the successful records are applied and failed updates remain in cache.

**Figure 16-9: Since MaxErrors was not exceeded, the successful records were applied and removed from cache. The failed records remain in cache** What happens when AMaxErrors is exceeded is demonstrated in

Figure 16-10.

Here MaxErrors is set to 1. When the second error occurs, the update process is aborted and the cache remains intact.



## Chapter 16: Using Cached Updates 463

**Figure 16-10: MaxErrors was exceeded, MaxErrors plus one error were reported, and the cache remains unchanged**

### APPLYUPDATES AND ONUPDATERECORD

In the automatic mode, FireDAC creates and executes the SQL that performs the corresponding update to the underlying table. In the examples I've covered so far in this section, those have all been SQL INSERT queries.

Sometimes you need to take charge of the SQL that is executed against the underlying database. For example, the query that you have placed in cache may include one or more joins, and in that case, FireDAC will only apply changes to the primary table, or may not be able to generate the SQL at all. In other cases, you may actually want to write to one or more tables not even included in the original query. For example, you might want to update the queried table as well as a table used to maintain an audit trail.

In some cases, you need to perform these customized queries using the brute force method. However, when possible, it is easiest to add a custom

OnUpdateRecord event handler. When you do this, a call to ApplyUpdates

causes FireDAC to examine the cache, and then, instead of generating and executing its own SQL, it invokes your OnUpdateRecord event handler, once for each record in cache.

## 464 Delphi in Depth: FireDAC

The OnUpdateRecord event handler is of TFDUpdateRecordEvent type. The definition of TFDUpdateRecordEvent is shown here:

```
TFDUpdateRecordEvent = procedure (ASender: TDataSet;  
ARequest: TFDUpdateRequest; var AAction: TFDErrorAction;  
AOOptions: TFDUpdateRowOptions) of object;
```

ASender is a reference to the FireDAC dataset for which you called

ApplyUpdates, or in the case of the centralized model, it is the FireDAC dataset whose record needs to be applied. Importantly, the current record of this dataset is the record in a cached state, and its OldValue, NewValue, CurValue, and Value properties are available to your code (though depending on the type of update that needs to be applied, some of these properties may be null).

You use the current record of ASender to help you perform your update.

However, you do not try to change this record, and you do not move off of this current record. Instead, you use the values of this current record to perform whatever task you intend to perform in your OnUpdateRecord event handler,

and that task is typically to construct and execute an SQL query.

The ARequest parameter informs you about what kind of update you need to perform. This parameter is of a TFDUpdateRequest type, and it is a range of TFDActionRequest types. The definitions of TFDActionRequest and TFDUpdateRequest are shown here:

```
TFDActionRequest = (arNone, arFromRow, arSelect, arInsert,  
arUpdate, arDelete, arLock, arUnlock,  
arFetchRow, arUpdateHBlocks, arDeleteAll,  
arFetchGenerators);
```

```
TFDUpdateRequest = arInsert .. arFetchGenerators;
```

In most cases, you only need to concern yourself with arInsert, arUpdate, and

arDelete requests.

The AAction parameter permits you to inform FireDAC about your action, which it might use to abort the update process. AAction of a TFDErrorAction type, whose declaration is shown here:

```
TFDErrorAction = (eaFail, eaSkip, eaRetry, eaApplied,  
eaDefault, eaExitSuccess, eaExitFailure);
```

A description of the TFDErrorAction values are listed in Table 16-1.

## Chapter 16: Using Cached Updates 465

### Action

### Description

eaFail

Marks the update as failed and returns an error.

eaSkip

Leaves the record unchanged in the cache.

eaRetry

Retries the current operation. Typically not applicable to cached updates.

eaApplied

Signals to FireDAC that you applied the update. FireDAC will continue applying any remaining updates.

eaDefault

Requests that FireDAC perform its default process to this record. In other words, FireDAC should update the record.

eaExitSuccess Signals success, but aborts the ApplyUpdates process.

eaExitFailure

Signals failure and aborts the ApplyUpdates process.

### Table 16-1: The TFDErrorAction descriptions

The final parameter, AOptions, is a set of TFDUpdateRowOption flags. This is rarely used, readonly information that you can generally ignore when applying updates. The TFDUpdateRowOptions type, as well as the

TFDUpdateRowOptions enumeration, is shown here:

```
TFDUpdateRowOption = (uoCancelUnlock, uoImmediateUpd,  
uoDeferredLock, uoOneMomLock,  
uoNoSrvRecord, uoDeferredGenGet);
```

```
TFDUpdateRowOptions = set of TFDUpdateRowOption;
```

Writing OnUpdateRecord event handlers can be tricky, as you are taking full responsibility for performing the updates. This can be seen in the following event handler, which can be found in the FDCacheUpdatesErrors project:

```
procedure TForm1.TwoFieldsTableUpdateRecord(ASender: TDataSet;
```

```
AResponse: TFDUpdateResponse; var AAction: TFDErrorAction;
```

```
AOptions: TFDUpdateRowOptions);
```

```
var
```

```
FDQuery: TFDQuery;
```

```
466 Delphi in Depth: FireDAC
```

```
begin
```

```
AAction := eaFail;
```

```
FDQuery := TFDQuery.Create( nil );
```

```
try
```

```
FDQuery.Connection := EmployeeConnection;
```

```
case AResponse of
```

```
arInsert:
```

```
begin
```

```
FDQuery.SQL.Text := 'INSERT INTO TwoFieldsTable ' +  
' ("FIRST", "LAST")' +  
' VALUES ( :f, :l );';
```

```
FDQuery.Params[0].AsString :=
```

```
ASender.FieldByName('FIRST').CurValue;
```

```
FDQuery.Params[1].AsString :=
```

```
ASender.FieldByName('LAST').CurValue;
```

```
FDQuery.ExecSQL;
```

```
AAction := eaApplied;  
end;  
arUpdate:  
begin  
FDQuery.SQL.Text := ‘UPDATE TwoFieldsTable ‘ +  
‘ SET “FIRST” = :cf, ‘ +  
‘  
“LAST” = :cl ‘ +  
‘ WHERE “FIRST” = :of ‘ +  
‘  
AND “LAST” = :ol;’;  
FDQuery.Params[0].AsString :=  
ASender.FieldByName(‘FIRST’).CurValue;  
FDQuery.Params[1].AsString :=  
ASender.FieldByName(‘LAST’).CurValue;  
FDQuery.Params[2].AsString :=  
ASender.FieldByName(‘FIRST’).OldValue;  
FDQuery.Params[3].AsString :=  
ASender.FieldByName(‘LAST’).OldValue;  
FDQuery.ExecSQL;  
AAction := eaApplied;  
end;  
arDelete:  
begin  
FDQuery.SQL.Text := ‘DELETE FROM TwoFieldsTable ‘ +  
‘ WHERE “FIRST” = :f ‘ +  
‘ AND “LAST” = :l;’;  
FDQuery.Params[0].AsString :=  
ASender.FieldByName(‘FIRST’).OldValue;
```

```
FDQuery.Params[1].AsString :=  
Chapter 16: Using Cached Updates 467  
ASender.FieldByName('LAST').OldValue;  
FDQuery.ExecSQL;  
AAction := eaApplied;  
end;  
else  
begin  
AAction := eaDefault;  
end;  
end;  
finally  
FDQuery.Free;  
end;  
end;
```

There is, however, one approach that can significantly reduce the complexity of an OnUpdateRecord event handler, which is to employ an FDUpdateSQL component to perform your update. An FDUpdateSQL component can perform

the parameter binding for you, as well as assist in generating the parameterized INSERT, UPDATE, and DELETE queries at design time. For more information

on the FDUpdateSQL component, see *Chapter 5, More Data Access*.

### **ApplyUpdates and Errors**

In many cases, FireDAC does a good job of preventing errors from being posted into cache. For example, if I had not set the CheckRequired property of the TwoFieldsTable FDQuery's UpdateOptions property to False, I would not be

permitted to insert a record unless I had provided a value for both fields. As a result, the errors that I forced in my preceding examples would never have been produced.

Nonetheless, errors are always a possibility, and you need to have a strategy

for handling them.

There are two basic approaches that you can take to handle errors that are generated during the ApplyUpdates process. The first is to handle the errors upon the return of the call to the ApplyUpdates method, and the second is to handle the errors using one of two available event handlers.

## **MANAGING ERRORS FOLLOWING THE CALL TO APPLYUPDATES**

As I described earlier, the ApplyUpdates method returns an Integer value indicating how many records in cache could not be applied to the underlying database. And, these records remain in cache following your call to

468 Delphi in Depth: FireDAC

ApplyUpdates. Your options are to fix or discard any errors that were encountered.

To fix the errors, you can permit the user to make the fix, or you can do it programmatically. In the case where you are using cached updates and there is no user interface, programmatically fixing the errors is your only option.

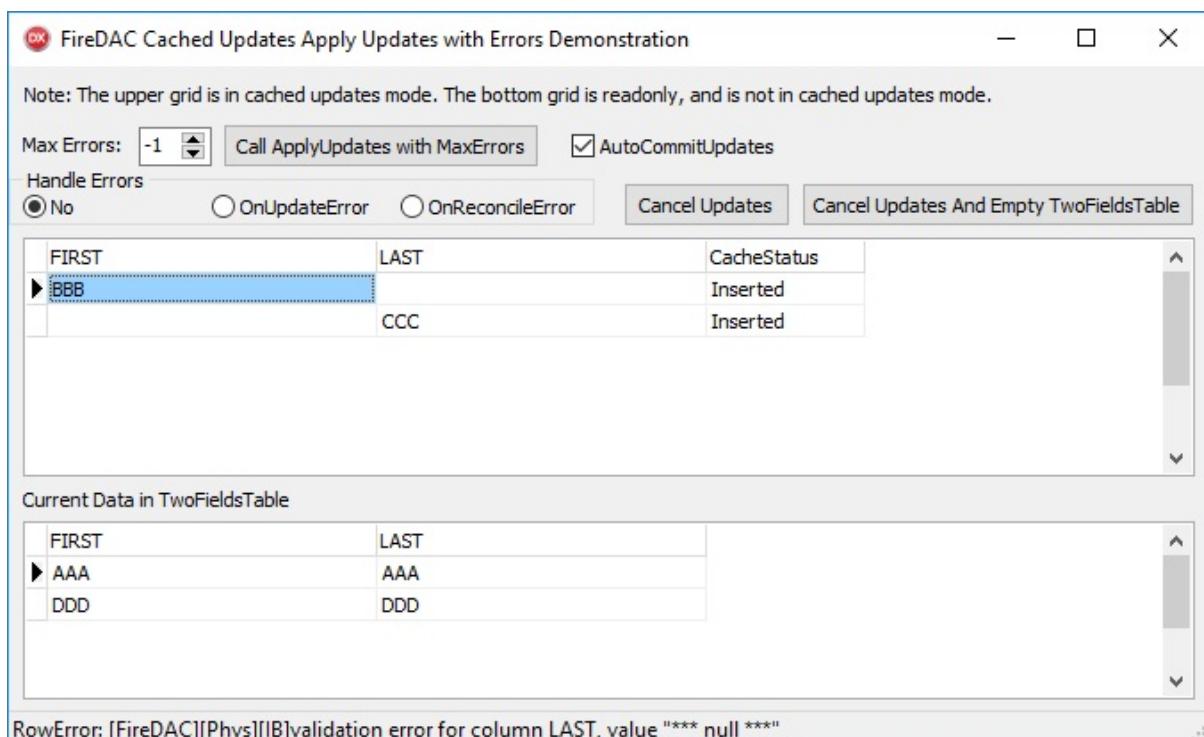
For those errors that you cannot fix, or do not want to fix, you can discard those errors. This can be done by reverting the record to its original state (inserted records will be deleted) using the RevertRecord method, or by canceling the cached updates entirely using the CancelUpdates method. Recall, however, that it is completely possible that ApplyUpdates gets aborted when AMaxErrors is exceeded. This means that some records that are perfectly fine may remain in the cache, in which case, canceling all updates using CancelUpdates will discard those records as well.

Fortunately, there is an easy way to determine which records in cache caused a problem, and to be able to actually examine the exception that was raised when the update failed. When a FireDAC dataset is in the cached updates mode, each record in that dataset has an object attached to it, and this object can be referenced using the RowError property of the dataset. This property returns the object associated with the current record, and that object will either be nil, or will be the exception that was raised when FireDAC attempted to post that change.

One use of the RowError property is demonstrated in the OnDataChange event handler of the data source that points to the TwoFieldsTable FDQuery. This event handler will display any errors that were encountered in the available status bar as you navigate through the cache that appears in the

upper DBGrid of the FDCachedUpdatesErrors project, as shown in Figure 16-11:

```
procedure TForm1.DataSource1DataChange(Sender: TObject; Field: TField);
begin
  if TwoFieldsTable.RowError <> nil then
    StatusBar1.SimpleText := 'RowError: ' +
    TwoFieldsTable.RowError.Message
  else
    StatusBar1.SimpleText := '';
  end;
```



Chapter 16: Using Cached Updates 469

**Figure 16-11: The value of the RowError exception is shown for the first record in cache following a failed attempt to insert the record**

## MANAGING ERRORS USING EVENT HANDLERS

Instead of waiting until the `ApplyUpdates` method returns, you can inspect the error, and potentially fix it, using event handlers. One of these event handlers, `OnUpdateError`, triggers when an error occurs during the call to `ApplyUpdates`.

This is true whether you are relying on FireDAC to apply the updates, or you set AAction to eaFail from within an OnUpdateRecord event handler. If you are

successful in fixing all errors that trigger the OnUpdateError event handler, all records in cache will be applied, and the cache will be empty.

The second event handler that you can use is the OnReconcileError event handler. This event handler will trigger once for each record that generated an error, but does so at the conclusion of the ApplyUpdates call. Unlike OnUpdateError, records that generate an OnReconcileError event handler cannot be applied to the underlying database from that event handler. However, you can fix them from the OnReconcileError event handler. If you are

successful, you will then have to issue a second call to ApplyUpdates to apply the changes you have fixed. Let me now consider each of these event handlers in more detail.

#### 470 Delphi in Depth: FireDAC

OnUpdateError is executed immediately after an attempt to apply an update fails. This event is of the type TFDUpdateErrorEvent, and its definition is shown here:

```
TFDUpdateErrorEvent = procedure (ASender: TDataSet;  
AException: EFDEception; ARow: TFDDatSRow;  
ARequest: TFDUpdateRequest;  
var AAction: TFDErrorAction) of object;
```

The first parameter, ASender, is the FireDAC dataset whose update was being applied. This parameter is similar to the OnUpdateRecord ASender parameter, in that the current record of this dataset is associated with the record whose update has failed.

AException, the second parameter, is the exception that was raised as a result of the failure.

ARow is a TFDDatSRow instance. It represents the current row of ASender, but it has a different set of methods.

ARequest is a TFDUpdateRequest reference, and it identifies the type of update that FireDAC (or your OnUpdateRecord event handler) was trying to apply

when the update failed. TFDUpdateRequest was described in some detail in the section, *ApplyUpdates and OnUpdateRecord*, earlier in this chapter.

Finally, AAction is a TFDErrorAction value passed by reference. By default, its value will be eaDefault, which will equate to a value of eaFail if you do not change it. If you determine that you can fix the error, you can read the OldValue and NewValue values of the fields of the current record, and then assign data to NewValue if you can fix the data. When you do fix the data, you should assign AAction the value of eaRetry, and that will cause FireDAC to either try again, or to invoke your OnUpdateRecord event handler again.

Unfortunately, there is a bug that may prevent you from using OnUpdateRecord successfully. Specifically, when UpdateOptions.UpdateChangedFields is set to True, and you correct an error from OnUpdateError and set AAction to eaRetry, OnUpdateRecord will attempt to update only the fields that you fixed, and that may not include fields that were edited during cached updates. This will either cause another error to be generated, or will post an incomplete record to the underlying table.

To work around this error, you can employ an FDUpdateSQL component and define the insert, update, or delete queries yourself, including the error

Chapter 16: Using Cached Updates 471

correction. Alternatively, you fix the problem by simply setting the UpdateOptions.UpdateChangedFields property to False.

I used this second approach to make OnUpdateError work in the

FDCachedUpdatesErrors. To begin with, the main form of this project includes a radio button group which permits you to select to either use OnUpdateError, OnReconcileError, or neither. Here is the event handler associated with this radio button group:

```
procedure TForm1.rgpHandleErrorsClick(Sender: TObject);
begin
case rgpHandleErrors.ItemIndex of
  0:
    begin
      TwoFieldsTable.OnUpdateError := nil;
      TwoFieldsTable.OnReconcileError := nil;
```

```

TwoFieldsTable.UpdateOptions.UpdateChangedFields := True;
end;
1:
begin
TwoFieldsTable.OnUpdateError := TwoFieldsTableUpdateError;
TwoFieldsTable.UpdateOptions.UpdateChangedFields := False;
TwoFieldsTable.OnReconcileError := nil;
end;
2:
begin
TwoFieldsTable.OnUpdateError := nil;
TwoFieldsTable.OnReconcileError :=
TwoFieldsTableReconcileError;
TwoFieldsTable.UpdateOptions.UpdateChangedFields := True;
end;
end;
end;

```

As you can see, when you select to use OnUpdateError, an event handler is assigned to the OnUpdateError property and

UpdateOptions.UpdateChangedFields is set to False. The following is the code on OnUpdateError event handler:

```

procedure TForm1.TwoFieldsTableUpdateError(ASender: TDataSet;
AException: EFDEception; ARow: TFDDatSRow;
ARequest: TFDUpdateRequest; var AAction: TFDErrorAction);
begin
472 Delphi in Depth: FireDAC
if (not ASender.Fields[0].IsNull) and
(not ASender.Fields[1].IsNull) then
AAction := eaDefault
else

```

```

if ASender.Fields[0].IsNull and ASender.Fields[1].IsNull then AAction := eaFail

else

if ASender.Fields[0].IsNull then

begin

ASender.Edit;

ASender.Fields[0].Value := ASender.Fields[1].Value;

ASender.Post;

AAction := eaRetry;

end

else

if ASender.Fields[1].IsNull then

begin

ASender.Edit;

ASender.Fields[1].Value := ASender.Fields[0].Value;

ASender.Post;

AAction := eaRetry;

end

else

AAction := eaFail;

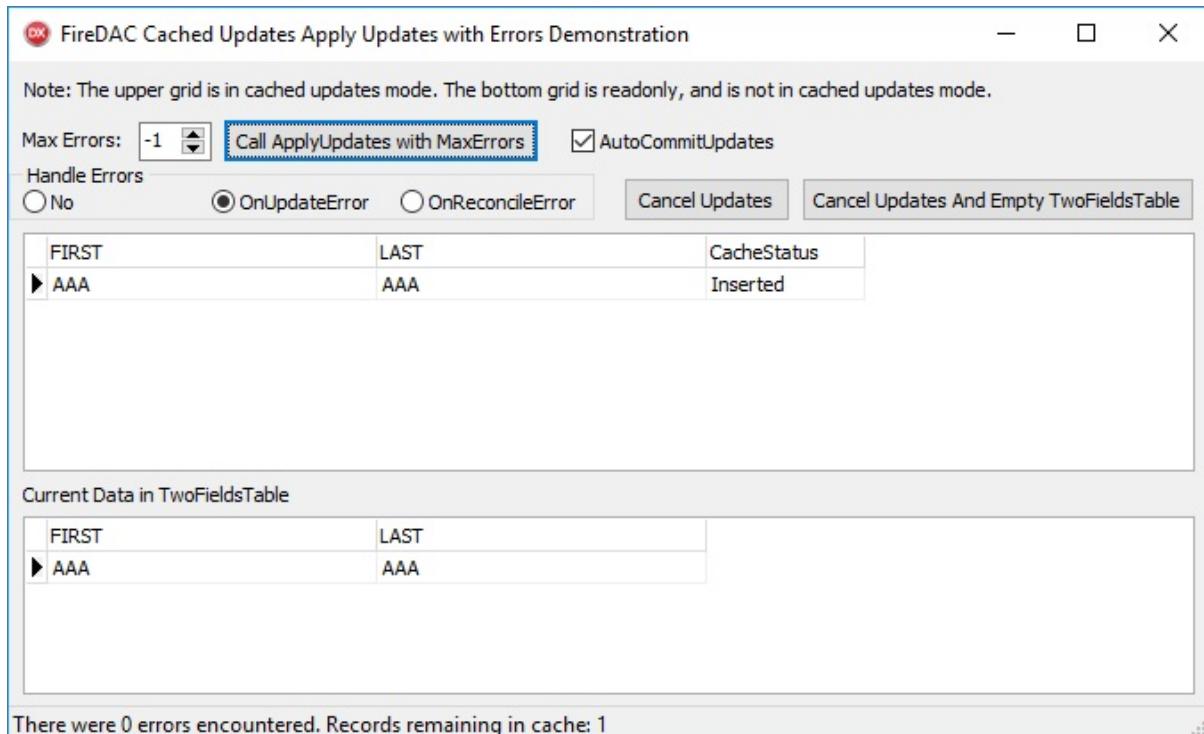
end;

```

To demonstrate the use of OnUpdateError, I started by clearing the cache and emptying the TwoFieldsTable table. I then entered a single record with AAA in the first field, after which I set the radio button group to OnUpdateError and clicked the button labeled Call ApplyUpdates with MaxErrors.

The result is shown in Figure 16-12. The record gets fixed and posted.

Unfortunately, it also remains in cache with an update status of usInserted.



## Chapter 16: Using Cached Updates 473

**Figure 16-12: OnUpdateError has fixed an error during the call to ApplyUpdates**

*Note: I reported this problem, where the record remains in cache with an update status of usInserted, using the Embarcadero Quality Portal. You may want to monitor the issue I created to see if and when the problem is resolved.*

*The issue number is RSP-17785.*

As mentioned earlier, OnReconcileError executes at the conclusion of the ApplyUpdates invocation, and if it corrects errors, ApplyUpdates must be invoked again. OnReconcileError is a TFDReconcileErrorEvent property. The declaration of TFDReconcileErrorEvent is shown here:

```
TFDReconcileErrorEvent = procedure(DataSet: TFDDataset;
E: EFDEexception; UpdateKind: TFDDatSRowState;
var Action: TFDDAptReconcileAction) of object;
```

DataSet is the FireDAC dataset from which the record is being applied. The current record is the one associated with the record in cache. As with the

474 Delphi in Depth: FireDAC

OnUpdateError and OnUpdateRecord event handlers, you should make your

changes to this current record, but you should not attempt to navigate off of it.

The second parameter is the exception associated with this update failure. And the third parameter is the type of update that was being applied.

Finally, Action is a TFDDAptReconcileAction parameter passed by value. You use this parameter to signal to FireDAC what it should do with the failed record.

Here is the declaration of TFDDAptReconcileAction enumerated type:

```
TFDDAptReconcileAction = (raSkip, raAbort, raMerge, raCorrect,  
raCancel, raRefresh);
```

Table 16-2 lists the purpose of the TFDDAptReconcileAction values.

#### **Value**

#### **Description**

**raAbort**

Aborts the call to Reconcile. No further update failures will be processed.

**raCancel**

Cancels the updates to the current record. Inserted records are deleted.

**raCorrect**

Clears the current record error state. In other words, it marks the record as correctly applied.

**raMerge**

Clears the current record error state, so the changes to the record become the initial state of this record. In other words, it merges changes to the dataset's cache.

**raRefresh** Reverts the record to its original state and re-reads the record from the underlying database. Should not be used from within cached updates.

**raSkip**

Do nothing. Leave the record in cache.

**Table 16-2: The TFDDAptReconcileAction values and their descriptions**

Here is the event handler that is called when the radio group box is set to use the OnReconcileEvent handler:

Chapter 16: Using Cached Updates 475

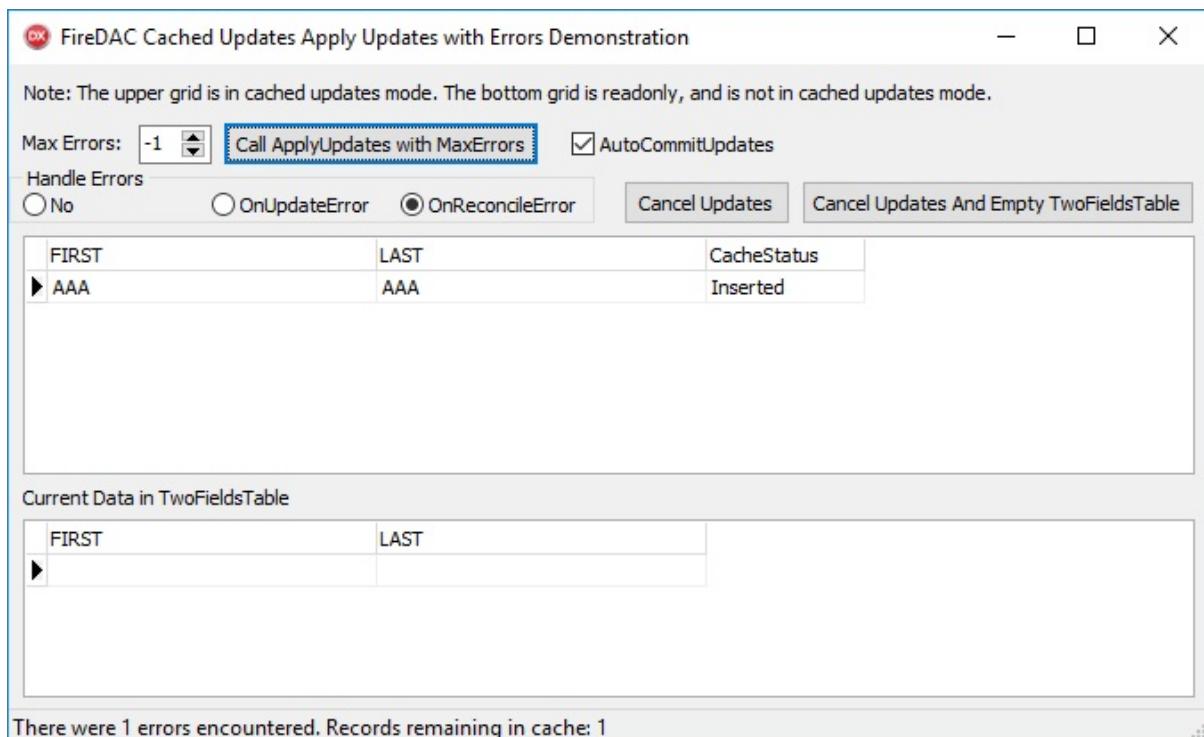
```
procedure TForm1.TwoFieldsTableReconcileError(  
  DataSet: TFDDataset;  
  E: EFDEception; UpdateKind: TFDDatSRowState;  
  var Action: TFDDAptReconcileAction);  
begin  
  if (not DataSet.Fields[0].IsNull) and  
    (not DataSet.Fields[1].IsNull) then  
    Action := raSkip  
  else  
    if DataSet.Fields[0].IsNull and DataSet.Fields[1].IsNull then Action :=  
      raSkip  
    else  
      if DataSet.Fields[0].IsNull then  
        begin  
          DataSet.Edit;  
          DataSet.Fields[0].Value := DataSet.Fields[1].Value;  
          DataSet.Post;  
          Action := raCorrect;  
        end  
      else  
        if DataSet.Fields[1].IsNull then  
          begin  
            DataSet.Edit;  
            DataSet.Fields[1].Value := DataSet.Fields[0].Value;  
            DataSet.Post;  
            Action := raCorrect;  
          end  
        end  
  end
```

```

end
else
  Action := raSkip;
end;

```

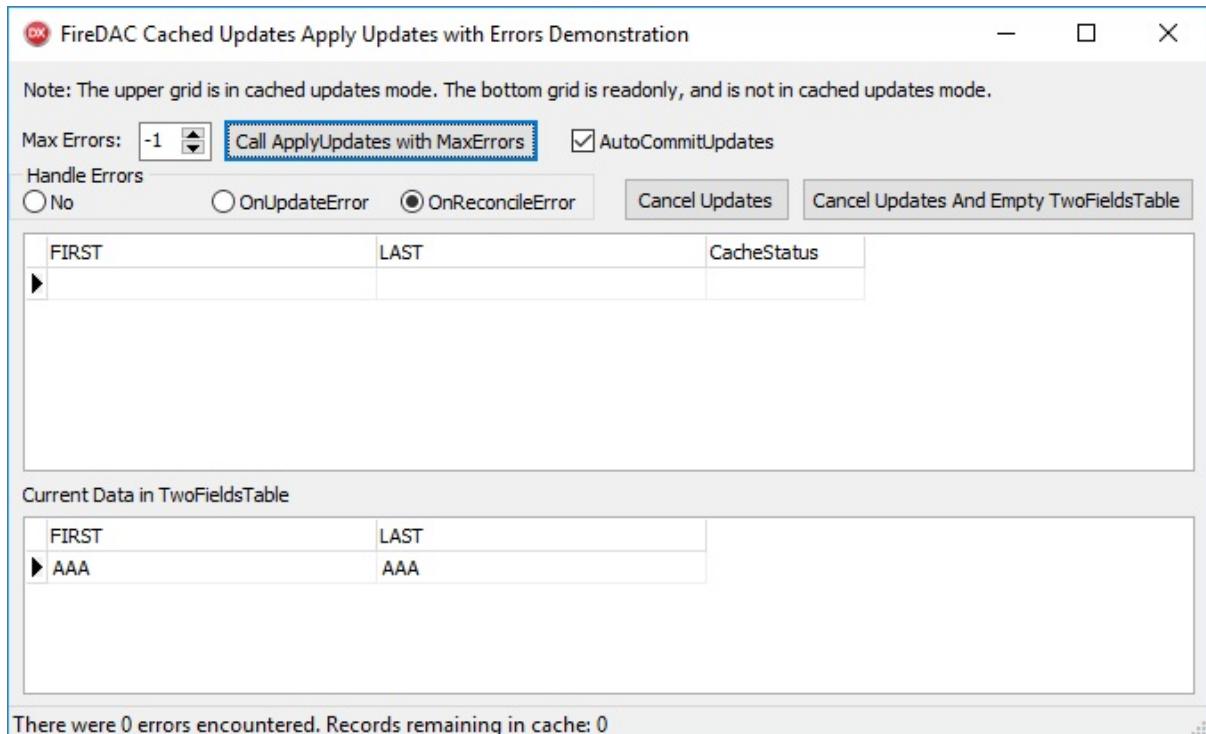
Using the FDCachedUpdatesError project, I cleared the cache and emptied TwoFieldsTable, after which I entered a single record with the value AAA in the first field. After setting the radio button group to OnReconcileError, I clicked the button labeled Call ApplyUpdates with MaxErrors. The update failed, and the updated record, with AAA in both fields, appears as an inserted record in the top grid, as shown in Figure 16-13.



476 Delphi in Depth: FireDAC

**Figure 16-13: The OnReconcileError event handler has fixed the record, but ApplyUpdates still needs to be called again**

I then clicked the button labeled Call ApplyUpdates with MaxErrors again, and this time, the corrected record was inserted into the underlying table and removed from the cache, as shown in Figure 16-14.



## Chapter 16: Using Cached Updates 477

**Figure 16-14: ApplyUpdates has been called again, and the update was applied and removed from the cache**

### Understanding Centralized Cached Updates

Up to this point, we have discussed cached updates from a decentralized model perspective. Fortunately, almost everything that I've discussed so far also applies to the centralized model of cached updates.

In the centralized model, one or more FireDAC datasets are associated with an FDSchemaAdapter by way of their SchemaAdapter property. The primary difference between the centralized model and the decentralized model is that certain operations are performed through the methods and properties of the schema adapter instead of through the individual datasets. For example, you initiate the update process by calling the ApplyUpdates method of the schema adapter. Similarly, you cancel updates, undo the last change, save and restore the save point, and commit updates through the schema adapter.

There are some operations that you can perform either through the schema adapter or the individual datasets, depending on your needs. For example, you can test the UpdatesPending property of the schema adapter to determine whether or not there are changes across any of the associated datasets. Testing the UpdatesPending property of a particular dataset will let you know if that

## 478 Delphi in Depth: FireDAC

dataset has changes. Similarly, ChangeCount can be used both on the schema adapter and individual datasets, where the former counts all changes, and the latter tells you only about the changes in the one dataset.

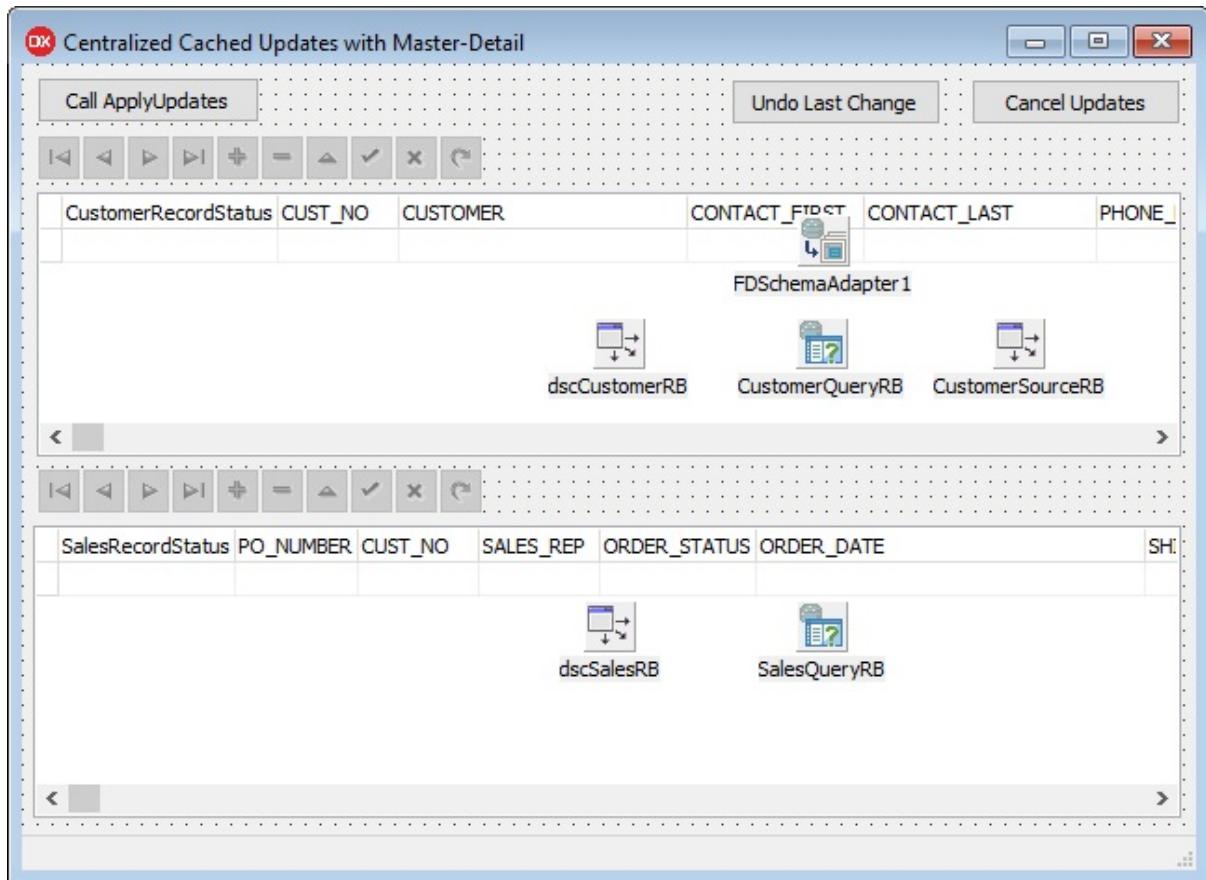
The centralized cached updates model is useful any time you want to manage the cache of one or more tables through a single touch point, the schema adapter. However, there are two scenarios where the centralized cached updates model is particularly useful. The first is when you want to ensure that the updates are applied in the same order in which the individual records participating in the cached updates session were edited, inserted, and deleted. In the decentralized model, the order in which the updates are applied is not guaranteed.

The second scenario, and one that is somewhat related to the first, is when two or more datasets participating in the cached updates session are related, either in a one-to-one association, or in a master-detail relationship. This second scenario is related to the first in that it is necessary that master table records get inserted before detail records, and that detail records get deleted before their associated master record gets deleted. The centralized model also supports cascading

deletes, in that, when properly configured, the deletion of a master table record will automatically result in the deletion of associated detail records, thereby preventing the accidental orphaning of the detail records.

The use of the centralized model of cached updates is demonstrated in the FDCachedMasterDetail project whose main form is shown in Delphi's designer in Figure 16-15. This project is based on the range-based example provided in the FDMasterDetail project, which was discussed in *Chapter 9, Filtering Data*.

Importantly, in order for the centralized model of cached updates to support master-detail relationships, it is necessary that the master-detail relationship be defined using the dynamic range-based technique. For more information about the dynamic range-based technique for defining master-detail relationships, please refer to *Chapter 9*.



## Chapter 16: Using Cached Updates 479

**Figure 16-15: The FDCachedMasterDetail project uses the centralized cached updates model**

*Code: The FDCachedMasterDetail project is included in the code download.*

The three buttons along the top of this form are used to apply updates, undo the last change, and to cancel all updates in cache. These buttons, which are initially disabled, are enabled once there are changes in cache. This is done from the `OnDataChange` event handler that is used by both of the data sources associated with the grids on this form. This event handler also displays information about any changes found in cache in the status bar. The assignment of change

information to the status bar demonstrates using the `ChangeCount` property of both the schema adapter and the associated datasets. This event handler, and the `ToggleButtons` custom method that it calls, are shown here:

```
procedure TMainForm.OnDataChange(Sender: TObject; Field: TField);
begin
```

```

ToggleButtons( FDSchemaAdapter1.UpdatesPending );
if FDSchemaAdapter1.UpdatesPending then
  StatusBar1.SimpleText := 
    'There are a total of ' +
    FDSchemaAdapter1.ChangeCount.ToString +
    ' updates in cache. The Customer table has ' +
    customerQueryRB.ChangeCount.ToString +
    ' changes, and the Sales table has ' +
    SalesQueryRB.ChangeCount.ToString
else
  StatusBar1.SimpleText := 'There are no changes in cache';
end;

procedure TMainForm.ToggleButtons(Enable: Boolean);
begin
  btnApplyUpdates.Enabled := Enable;
  btnUndoLastChange.Enabled := Enable;
  btnCancelUpdates.Enabled := Enable;
end;

```

In the decentralized model of cached updates, the operations initiated by these three buttons would be performed by calling the appropriate methods on the individual FireDAC datasets. In the centralized model, however, these operations must be performed at the schema adapter level. The following code contains the event handlers associated with these three buttons:

```

procedure TMainForm.btnApplyUpdatesClick(Sender: TObject);
var
  NumErrors: Integer;
  NumChanges: Integer;
begin
  // Ensure that all edits have been posted
  if CustomerQueryRB.State in dsEditModes then

```

```
CustomerQueryRB.Post;
if SalesQueryRB.State in dsEditModes then
  SalesQueryRB.Post;
  SharedDMVcl.FDConnection.StartTransaction;
  try
    NumChanges := FDSchemaAdapter1.ChangeCount;
    NumErrors := FDSchemaAdapter1.ApplyUpdates( -1 );
    SharedDMVcl.FDConnection.Commit;
    if NumErrors > 0 then
      StatusBar1.SimpleText := 'Not all changes could be posted'
      Chapter 16: Using Cached Updates 481
    else
      StatusBar1.SimpleText := NumChanges.ToString +
        ' were posted';
    except
      SharedDMVcl.FDConnection.Rollback;
    end;
    OnDataChange( nil, nil );
  end;
procedure TMainForm.btnUndoLastChangeClick(Sender: TObject);
begin
  FDSchemaAdapter1.UndoLastChange;
  OnDataChange( nil, nil );
end;
procedure TMainForm.btnCancelUpdatesClick(Sender: TObject);
begin
  FDSchemaAdapter1.CancelUpdates;
  OnDataChange( nil, nil );
end;
```

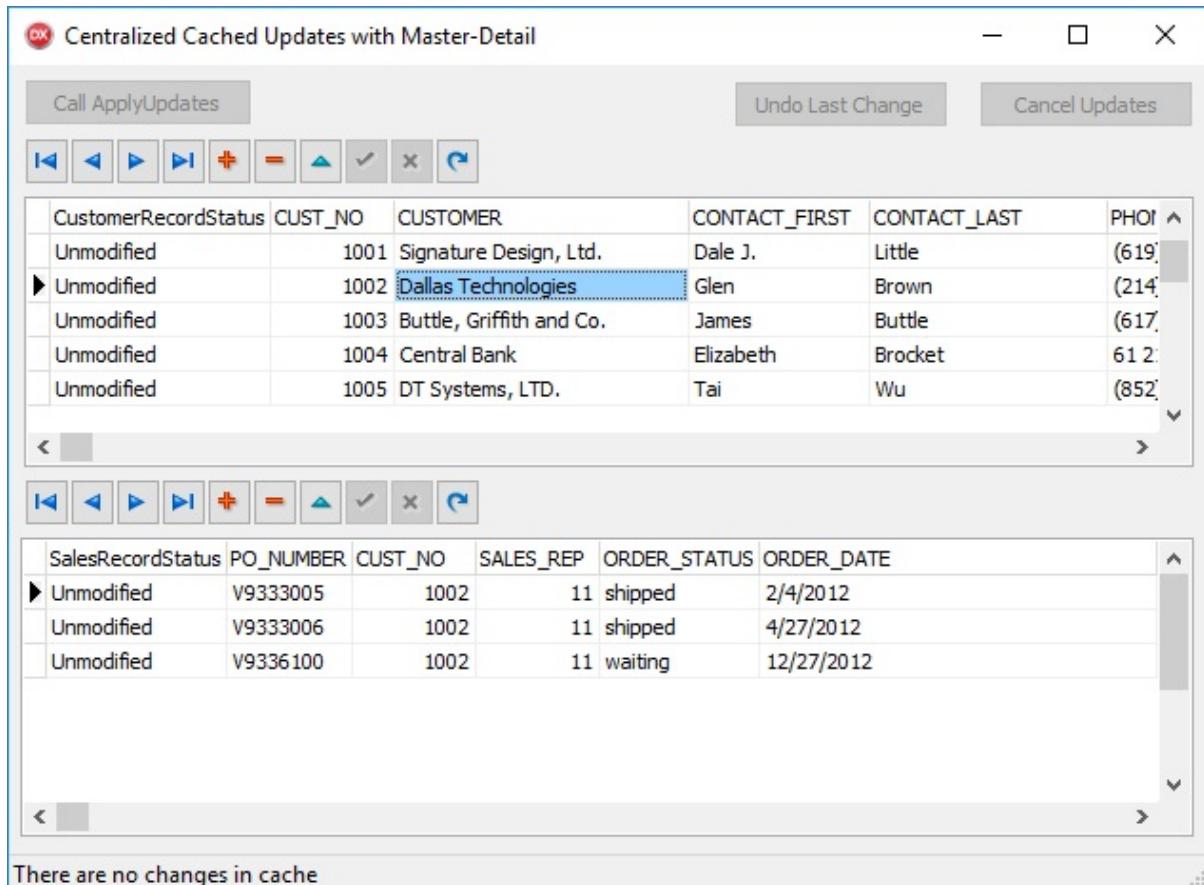
## **Managing Master-Detail Tables in the Centralized Cached Updates Model**

The centralized model of cached updates does a great job of permitting you to use cached updates with two or more tables related in a master-detail relationship. In fact, one of the projects that I am currently working on contains one data module where more than 30 tables, many with master-detail relationships, are being managed by centralized cached updates. There are, however, some configurations that you will normally need to apply in order for these related edits to be successfully applied to the underlying database when `ApplyUpdates` is called on the schema adapter. As I already mentioned, the

master-detail relationship needs to be defined using the dynamic range-based master-detail technique.

The range-based dynamic master-detail relationship ensures that only the detail records for the current master table record are available in the detail table. And in most cases, if a master table record is deleted, you will want any detail records associated with that master record to be deleted as well. In order to achieve this, you must set the `FetchOptions` properties `DetailDelete` and

`DetailServerDelete` to `True` (their default values are `False`).



## 482 Delphi in Depth: FireDAC

FetchOptions.DetailDelete provides a client-side deletion of the detail records associated with a master table record deletion.

FetchOptions.DetailServerDelete ensures that these detail records are also deleted from the underlying database when the ApplyUpdates method of the associated schema adapter is called. You need to set these properties to True on both the master FireDAC dataset as well as the detail dataset.

The CustomerQueryRB and SalesQueryRB components on the FDCachedMasterDetail project both have their DetailDelete and DetailServerDelete FetchOptions properties set to True. In addition, the UpdateOptions.AutoCommitUpdates property of the FD Schema Adapter is set to

True (very important!). Figure 16-16 shows the main form with the cursor on the second record of the Customer table, with three detail records being displayed in the Sales table.

**Figure 16-16: There are three sales records for customer Dallas Technologies**

Centralized Cached Updates with Master-Detail

CustomerRecordStatus	CUST_NO	CUSTOMER	CONTACT_FIRST	CONTACT_LAST	PHOT
Unmodified	1001	Signature Design, Ltd.	Dale J.	Little	(619)
► Unmodified	1003	Buttle, Griffith and Co.	James	Buttle	(617)
Unmodified	1004	Central Bank	Elizabeth	Brocket	612
Unmodified	1005	DT Systems, LTD.	Tai	Wu	(852)
Unmodified	1006	DataServe International	Tomas	Bright	(613)

SalesRecordStatus	PO_NUMBER	CUST_NO	SALES_REP	ORDER_STATUS	ORDER_DATE
► Unmodified	V9346200	1003	11	waiting	12/31/2012
Unmodified	V9345200	1003	11	shipped	11/11/2012
Unmodified	V9345139	1003	127	shipped	9/9/2012

There are a total of 4 updates in cache. The Customer table has 1 changes, and the Sales table has 3.

## Chapter 16: Using Cached Updates 483

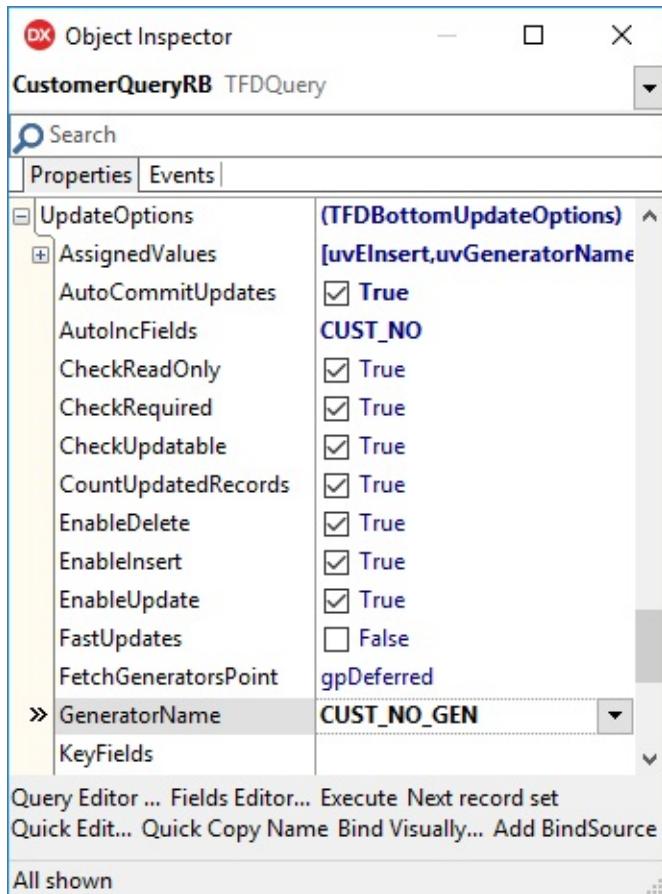
In Figure 16-17, the record for customer Dallas Technologies has been deleted.

The OnDataChange method has now updated the enabled property of the top three buttons, and has displayed information about the number of changes in cache in the status bar.

### **Figure 16-17: Deletion of the master record has cascaded, resulting in the deletion of the three detail records**

The insertion of detail records may also require some configuration. In the case of the Customer/Sales relationship, it is necessary for the CUST\_NO field in Customer to match the CUST\_NO field in Sales. This is somewhat complicated by the fact that the CUST\_NO field of the Customer table is an auto-increment field. And in the InterBase database, this is performed by a database entity called a *generator*. Other databases use other mechanisms. For example, auto-increment fields are supported in Microsoft SQL Server through *identity* fields.

In the case of InterBase, two UpdateOptions properties needed to be configured for the CustomerQueryRB FDQuery in order for it to properly generate the new



## 484 Delphi in Depth: FireDAC

CUST\_NO field value using the generator. These properties are GeneratorName and AutoIncFields, and they are shown in Figure 16-18, where GeneratorName is set to CUST\_NO\_GEN, and AutoIncFields is set to CUST\_NO.

**Figure 16-18: The CUST\_NO generator for CUST\_NO auto-increment field for the Customer table has been configured**

Figure 16-19 shows how a newly inserted record is given a temporary key field value, which is -1 in this case. This value is used to associate the newly created customer table record with the detail sales record.

Centralized Cached Updates with Master-Detail

CUST_NO	CUSTOMER	CONTACT_FIRST	CONTACT_LAST	PHONE_NO	AD
1012	3D-Pad Corp.	Michelle	Roche	1 43 60 61	22
1013	Lorenzi Export, Ltd.	Andreas	Lorenzi	02 404 6284	Via
1014	Dyno Consulting	Greta	Hessels	02 500 5940	Rui
1015	GeoTech Inc.	K.M.	Neppelebroek	(070) 44 91 18	P.C
-1	New World Industries	Robert	Floyd	(408) 324-1111	

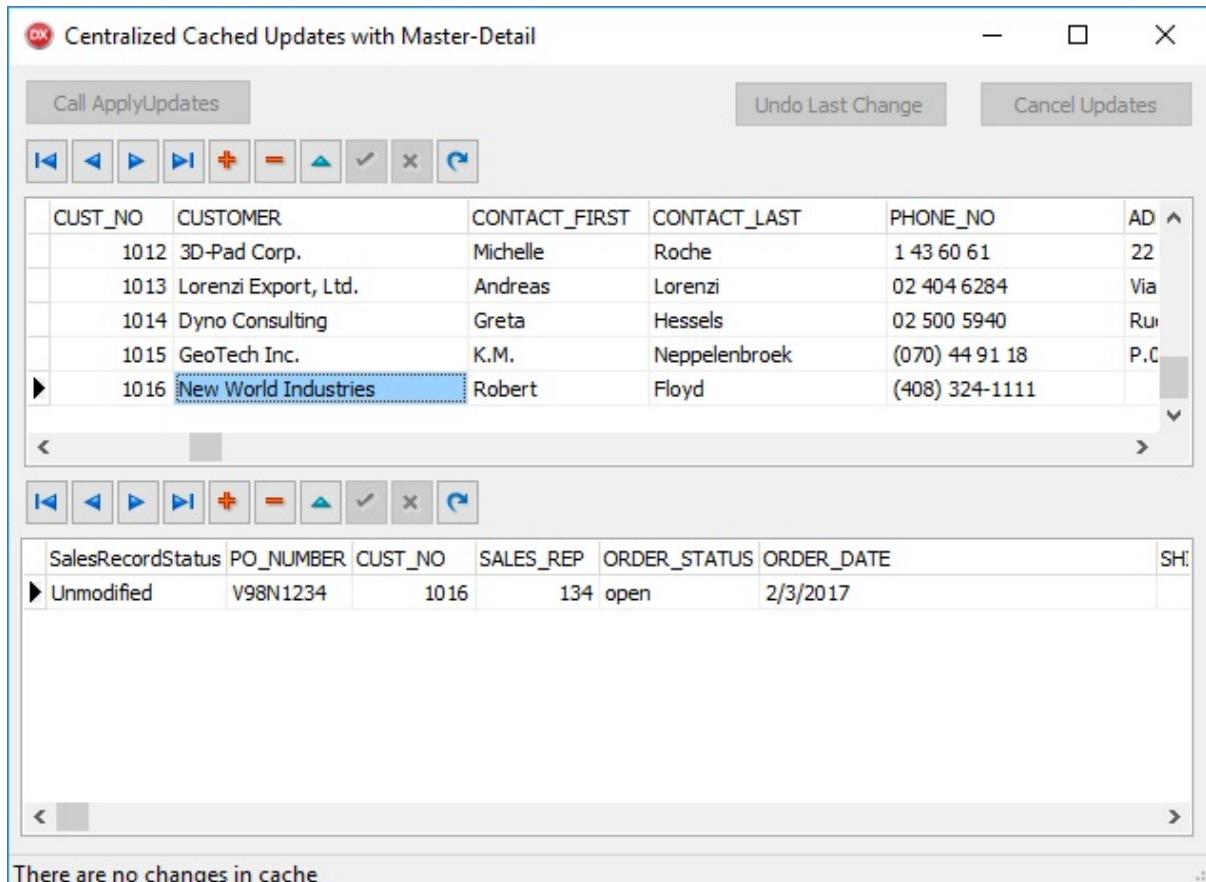
SalesRecordStatus	PO_NUMBER	CUST_NO	SALES_REP	ORDER_STATUS	ORDER_DATE
Inserted	V98N1234	-1	134	open	2/3/2017

There are a total of 2 updates in cache. The Customer table has 1 changes, and the Sales table has 1.

## Chapter 16: Using Cached Updates 485

**Figure 16-19: FireDAC has created a temporary key field value, -1, for the CUST\_NO field**

Once the values in cache have been posted, the temporary value is replaced by the value created by the generator. As you can see in Figure 16-20, this value is 1016.



## 486 Delphi in Depth: FireDAC

**Figure 16-20: The value created by the generator replaced the temporary key**

*Note: As I have mentioned elsewhere in this book, the employee.gdb database includes a number of constraints that must be met in order to post updates or insert new records, and the Sales table is a good example of this. In order to insert the record for PO number V98N1234, I had to supply valid values in the PO\_NUMBER, SALES REP, ORDER\_STATUS, ORDER\_DATE, DATE\_NEEDED, PAID, QTY\_ORDERED, TOTAL\_VALUE, DISCOUNT, and ITEM\_TYPE fields.*

In the next chapter, I show how to use Local SQL.

Chapter 17: Understanding Local SQL 487

# Chapter 17

## Understanding Local

### SQL

I've saved one of the more interesting features for last, and that is local SQL. Local SQL permits you to execute SQL statements against any dataset. For example, you can perform a query against an FDTable to gather simple aggregate statistics like SUM and AVG from the data it contains. Similarly, you can query an FDQuery and perform a left outer join to an FDStoredProc component (in which case, the stored procedure must return a result set).

Importantly, this ability to query datasets is not limited to FireDAC datasets. As a result, there is nothing to prevent you from performing a query that performs a join between an IBQuery, an SQLDataSet, and a ClientDataSet.

FireDAC performs this SQL slight-of-hand by converting SQL statements into TDataSet calls. For example, an SQL INSERT query is implemented by converting the SQL into calls to TDataSet.Post and TDataSet.Append.

Similarly, WHERE clauses are implemented through TDataSet.SetRange and TDataSet.Filter. This is the reason that Local SQL supports any TDataSet implementation, including those provided by third-party providers, such as Direct Oracle Access and UniDAC.

It's a very clever solution, and one that enables a whole range of interesting data-related operations that would otherwise be difficult or impossible to implement. For example, even if you are already using another data-access framework, such as dbExpress, or even a third-party data access framework, you can use FireDAC to perform SELECT queries against those datasets, and then use FireDAC's advanced features on the returned result set.

The specific dialog of SQL supported by Local SQL is based on SQLite, an open source and cross-platform SQL engine. In fact, in order to use Local SQL, you must have a connection that employs the FireDAC SQLite driver.

488 Delphi in Depth: FireDAC

*Note: If you are using the original release of Delphi XE5 without updates, you might be missing two files that are critical for working with SQLite. If you*

*encounter a compiler error indicating that one of these files is missing when building an application using the FireDAC SQLite driver, install the latest update for Delphi XE5.*

## **Implementing Local SQL**

Local SQL is fairly easy to use, but it does have a number of specific requirements. These are:

- ▀ You must have a FireDAC connection component that is configured to use the FireDAC SQLite driver.
- ▀ You must use an FDLocalSQL component, and this component must be configured to point to the FDConnection created in the preceding step.
- ▀ The FDLocalSQL needs to refer to any dataset you want to query in its DataSets property. There are several ways to do this.
  - ▀ You can then add one or more FDQuery components (or FDCommand components), which must be configured to use the FDConnection created for SQLite. The SQL statements defined for these components (the queries and commands) can include references to any of the datasets referred to by the FDLocalSQL component's DataSets property (or are assigned from within the FDLocalSQL's OnGetDataSet event handler).
  - ▀ Before any of the queries referred to in the preceding step can be executed, the FDLocalSQL component must be made active by setting its Active property to True.

My experience with local SQL is similar to my experience with aggregates. Specifically, using local SQL requires that you perform a series of steps, some of which must be performed in a particular order in order to ensure success. As a result, like I did when I discussed Aggregate fields (*Chapter 10, Creating and Using Virtual Fields*) I am going to demonstrate local SQL by walking you through a series of steps.

### **The Initial Setup**

Use the following steps to create the basic building blocks of the application:

1. Begin by selecting File | New | VCL Forms Application.



Chapter 17: Understanding Local SQL 489

2. Place on the newly created form a DBNavigator, a DBGrid, and a DataSource. Set the DataSource property of the DBNavigator and the DBGrid to DataSource1. Your new form should look something like that shown in Figure 17-1.

**Figure 17-1: The initial design for the FDLocalSQL project main form**

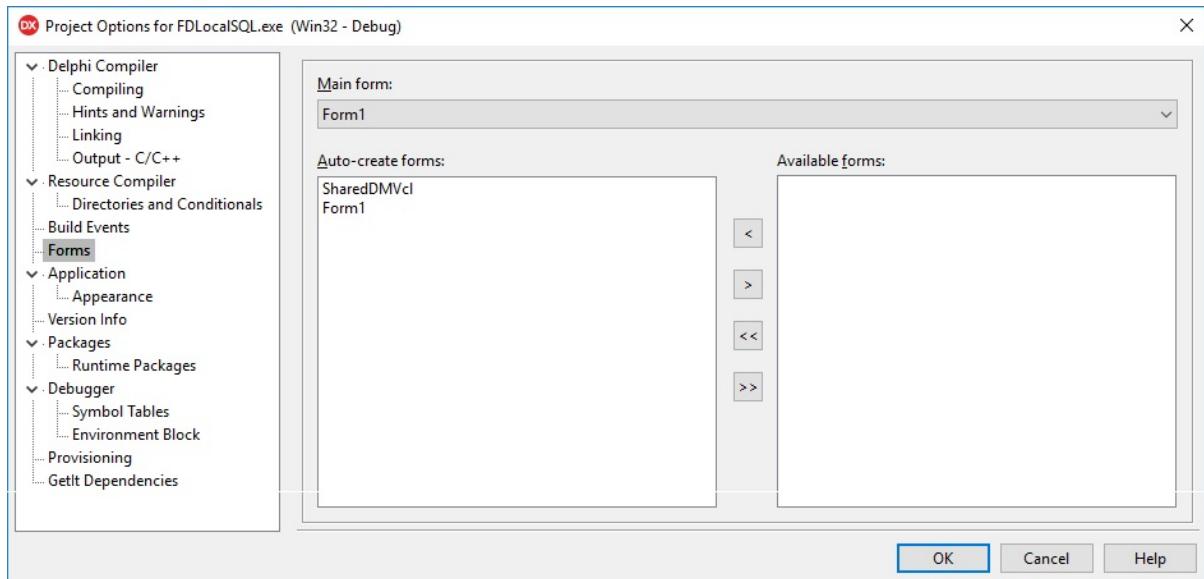
3. Next, select Project | Add to Project. Using the displayed browser, navigate to the Shared Files directory from the code download. If you unzipped to code download files onto your C drive, you will find this folder in the following location:

c:\FireDAC\Shared Files

Select the files SharedDMVclU.pas and DataPaths.pas and then click Open to add them to your project.

4. From your main form, select File | Use Units (or press Alt-F11) to open the Use Units dialog box. Select both SharedDMVclU and DataPaths and click OK.

5. You now need to ensure that the SharedDMVcl data module is created before your main form. To do this, select Project | Options (or press Ctrl-Shift-F11) to display the Project Options dialog box. Select the Forms tab from the panel on the left side of the Project Options dialog box, and



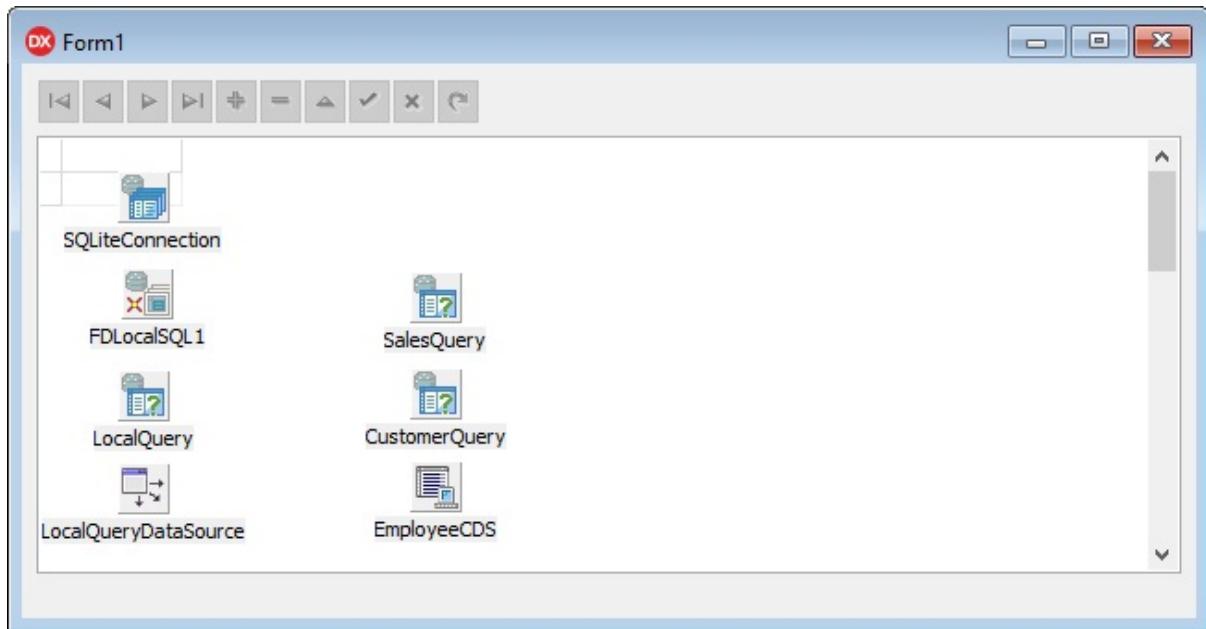
## 490 Delphi in Depth: FireDAC

then move the SharedDMVcl data module to a position above Form1, as shown in Figure 17-2. Click OK to continue.

**Figure 17-2: The SharedDMVcl data module has been positioned above Form1 in the Auto-create forms list**

6. Return to the main form. From the Tool Palette, place one FDConnection, three FDQueries, one FDLocalSQL, and one ClientDataSet on the form. If you are using an older version of FireDAC, you might also have to add an FDPhysSQLiteDriverLink component to the form as well.

7. To simplify many of the steps later in this segment, give meaningful names to most of your components using Table 17-1 as your guide.



Chapter 17: Understanding Local SQL 491

### Original Name

### New Name

FDConnection1

SQLiteConnection

FDQuery1

LocalQuery

DataSource1

LocalQueryDataSource

FDQuery2

SalesQuery

FDQuery3

CustomerQuery

ClientDataSet1

EmployeeCDS

**Table 17-1: New names for the components on the main form**

When you are done your form should look something like the one shown in Figure 17-3.

**Figure 17-3: Data access components have been placed on the form and**

**provided with meaningful names**

492 Delphi in Depth: FireDAC

## **Configuring the SQLite Connection**

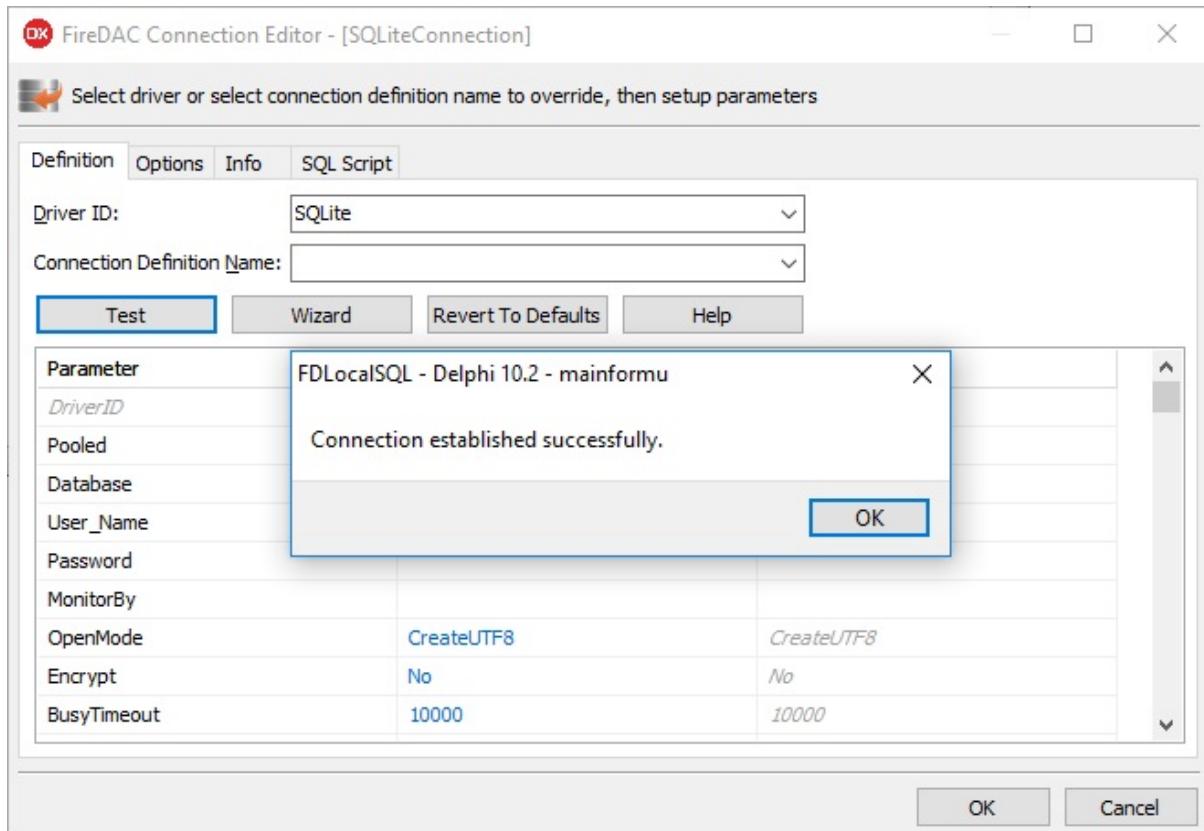
At this point there are two FDConnections in this project. One of them is on the SharedDMVcl data module and the other appears on the main form. The FDConnection on the data module will connect automatically to the employee.gdb database when the data module is initialized, so we do not have to worry about that connection.

In order to use LocalSQL, we will need an additional connection that will provide access to the SQLite engine. Use the following steps to configure and test this connection.

1. From the main form, double-click SQLiteConnection to display the FireDAC Connection Editor.
2. Set Driver ID to SQLite. FireDAC will respond by loading the parameters for the SQLite driver. You do not have to do any further configuration. However, you should test that everything is ok by clicking the Test button.

FireDAC will respond by challenging you for a username and password. Leave these fields blank and simply click OK. FireDAC should respond by displaying the confirmation shown in Figure 17-4. Click OK to close the confirmation dialog box, and then click OK to save the connection configuration.

If you fail to establish a connection to the SQLite engine, close the confirmation dialog box and then use the Info page of the FireDAC Connection Editor to determine what failed, after which, you should fix the problem and test the connection again.



## Chapter 17: Understanding Local SQL 493

**Figure 17-4: FireDAC has confirmed that it can access the SQLite engine**  
3. There is one final step. Since no login is required to use the SQLiteConnection, set its LoginPrompt property to False.

### Configuring the Datasets

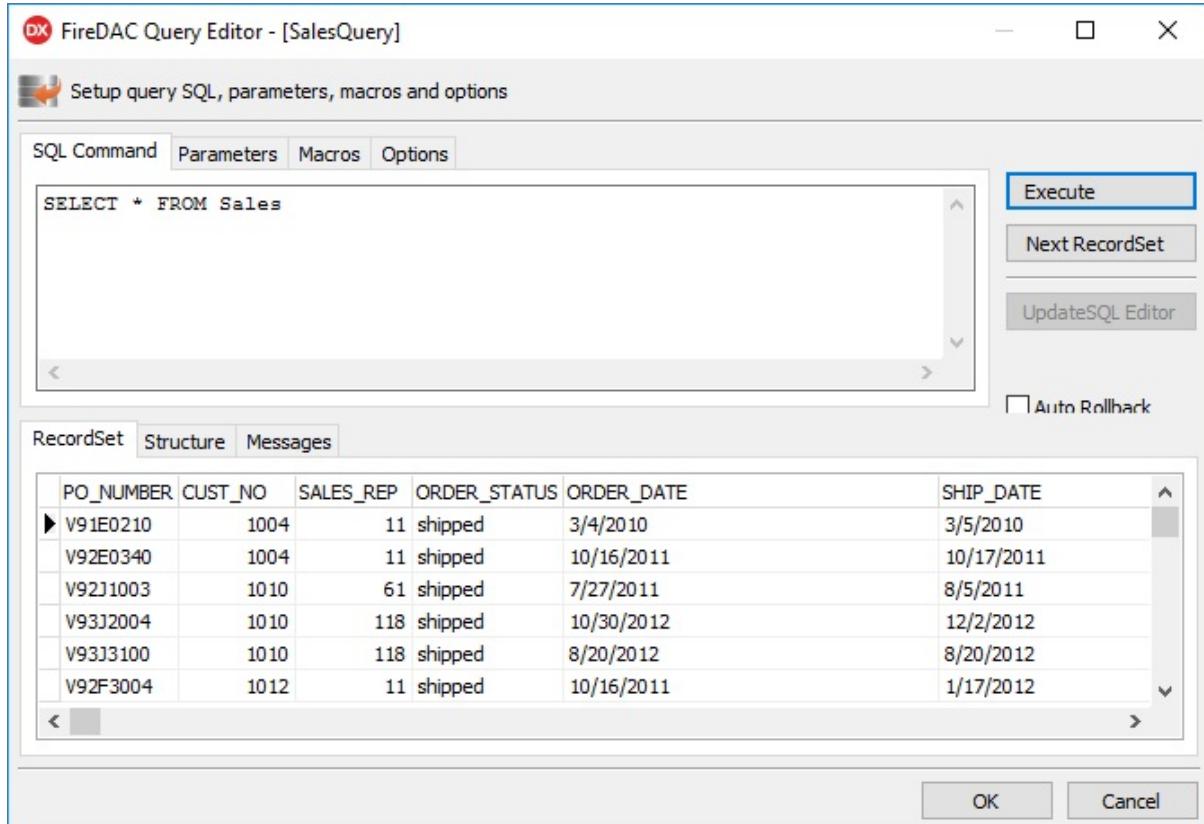
The individual datasets, the ones that we will query using Local SQL, are configured in the same fashion as you have done throughout this book — there is nothing uncommon about this configuration. The following steps walk you through the configuration of the two FDQueries named SalesQuery and

CustomerQuery, and the ClientDataSet named EmployeeCDS:

1. Select SalesQuery. Assuming that you placed it onto the main form after the FDConnection component had been added, this FDQuery will already have its Connection property set to SQLiteConnection. Select the Connection property and use the supplied dropdown menu to change this property to SharedDMVcl.FDConnection.

2. Double-click SalesQuery to open the FireDAC Query Editor. Enter the

following query in the upper pane of the SQL Command tab of the FireDAC Query Editor:



## 494 Delphi in Depth: FireDAC

SELECT \* FROM Sales

3. Click Execute to test this query. Your screen should look something like that shown in Figure 17-5. Click OK to close the FireDAC Query Editor and save the query in SalesQuery.

**Figure 17-5: The SELECT \* FROM Sales query returns all records and fields from the Sales table of employee.gdb**

4. Select CustomerQuery. Set its Connection property to SharedDMVcl.FDConnection. Next, set its SQL property to the following SELECT statement:

SELECT \* FROM Customer

Chapter 17: Understanding Local SQL 495

5. Finally, select EmployeeCDS and set its FileName property to the employee.xml file. This file can be found in the following directory in

Delphi 10.2 Tokyo:

c:\users\public\documents\embarcadero\studio\19.0\  
samples\data\customer.xml

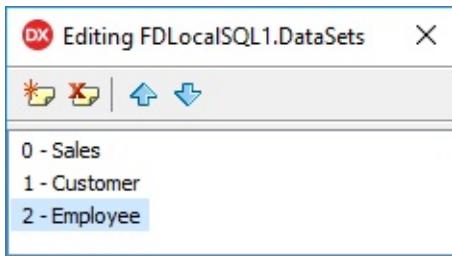
In other versions of Delphi, this directory will be similar, in most cases differing only by the version number. (See the DataPaths.pas unit for examples of valid paths based on the version of Delphi you are using.)

6. Set the Active property of SalesQuery, CustomerQuery, and EmployeeCDS to True.

## **Configuring FDLocalSQL**

All the prerequisite requirements for using Local SQL are now set. The remaining steps are necessary to configure the FDLocalSQL component, as well as the FDQuery that will actually execute the SQL against your datasets.

1. Select FDLocalSQL1. Its Connection property should already be assigned to SQLiteConnection. If not, set Connection to SQLiteConnection now.
2. Next, select the DataSets property of the FDLocalSQL and click the ellipsis button to display the DataSets collection editor. Click the Add New button on the DataSets collection editor three times to add three new datasets to the collection.
3. Select the first dataset in the collection editor and use the Object Inspector to set the DataSet property to SalesQuery and its Name property to Sales. Next, select the next dataset in the collection and set its DataSet property to CustomerQuery and its name property to Customer. Finally, select the last dataset and set its DataSet property to EmployeeCDS and its Name property to Employee. Your DataSets collection editor should now look something like that shown in the following illustration.



## 496 Delphi in Depth: FireDAC

4. Close the DataSets collection editor. The last step we need to perform on the FDLocalSQL component is to set its Active property to True. This is essential. If you do not take this step, any queries against the SQLite connection will fail.

5. We are finally ready to create our local query. Double-click the FDQuery named LocalQuery to display its FireDAC Query Editor. The query that we need can make use of most of the syntax supported by the SQLite engine, with very few exceptions. For example, data definition language (DDL) SQL statements such as ALTER and DROP are not supported. However, most of the SQL statements that you are likely to use are supported, such as SELECT, INSERT, and DELETE.

6. Enter the following SQL statement into the FireDAC Query Editor.

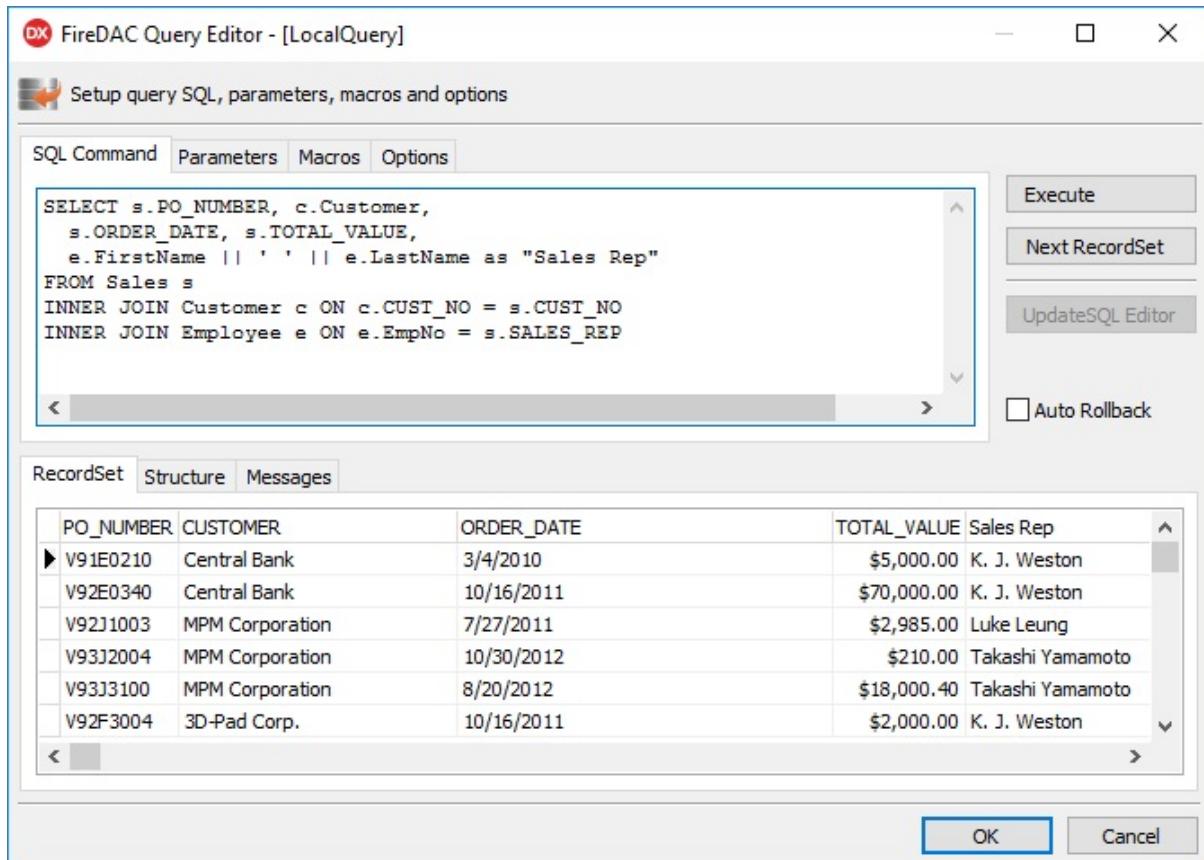
Notice that in this statement we used the names that appear in the FDLocalSQL's DataSets collection editor to refer to the individual datasets that we are querying:

```
SELECT s.PO_NUMBER, c.Customer,
s.ORDER_DATE, s.TOTAL_VALUE,
e.FirstName || ' ' || e.LastName as "Sales Rep"
FROM Sales s
```

```
INNER JOIN Customer c ON c.CUST_NO = s.CUST_NO
```

```
INNER JOIN Employee e ON e.EmpNo = s.SALES_REP
```

7. Test your query by clicking the Execute button. Your Query Editor should look something like that shown in Figure 17-6.



## Chapter 17: Understanding Local SQL 497

**Figure 17-6: Testing a Local SQL query in the FireDAC Query Editor**

8. Click OK to save your query.
9. Set the Active property of LocalQuery to True.
10. Finally, set the DataSet property of LocalQueryDataSource to LocalQuery. Your form should now look something like that shown in Figure 17-7. I moved the data access components from the position you saw them in in Figure 17-6 so that they would not cover up the text.

The screenshot shows a Delphi application window titled "Form1". Inside, a DBGrid control displays a dataset. The grid has columns: PO\_NUMBER, CUSTOMER, ORDER\_DATE, TOTAL\_VALUE, and Sales Rep. The data is as follows:

PO_NUMBER	CUSTOMER	ORDER_DATE	TOTAL_VALUE	Sales Rep
V91E0210	Central Bank	3/4/2010	\$5,000.00	K. J. Weston
V92E0340	Central Bank	10/16/2011	\$70,000.00	K. J. Weston
V92J1003	MPM Corporation	7/27/2011	\$2,985.00	Luke Leung
V93J2004	MPM Corporation	10/30/2012	\$210.00	Takashi Yamamo
V93J3100	MPM Corporation	8/20/2012	\$18,000.40	Takashi Yamamo
V92F3004	3D-Pad Corp.	10/16/2011	\$2,000.00	K. J. Weston
V93F3088	3D-Pad Corp.	8/27/2012	\$10,000.00	Jacques Glon
V93F2030	3D-Pad Corp.	12/12/2012	\$450,000.49	Jacques Glon
V93F2051	3D-Pad Corp.	12/18/2012	\$999.98	Jacques Glon
V93H0030	DT Systems, LTD.	12/12/2012	\$5,980.00	Takashi Yamamo
V94H0079	DT Systems, LTD.	2/13/2013	\$9,000.00	Luke Leung

Icons next to some dates indicate they are linked to specific components: SQLiteConnection, FDLocalSQL1, SalesQuery, CustomerQuery, and EmployeeCDS.

## 498 Delphi in Depth: FireDAC

**Figure 17-7: A DBGrid displays the result set created by a local query that joins two FDQueries with a ClientDataSet**

In the preceding steps, we added the datasets against which we were going to execute SQL statements to the FDLocalSQL's DataSets property. There are two alternative ways of assigning datasets to the FDLocalSQL.DataSets property.

Both of these alternatives actually result in the dataset being added to the DataSets property, but do so indirectly.

The first alternative is available only when your datasets are FireDAC datasets.

In those cases, you can use their LocalSQL property to assign them to the FDLocalSQL.DataSets property. In these cases, the name of the FireDAC dataset component can be used in the SQL query. For example, if instead of adding the SalesQuery and CustomerQuery FDQueries to the DataSets collection of the LocalSQL, if we had set the LocalSQL property of these two FDQueries to FDLocalSQL1, we could have modified the SQL associated with

the LocalQuery component to look like the following:

```
SELECT s.PO_NUMBER, c.Customer,
s.ORDER_DATE, s.TOTAL_VALUE,
e.FirstName || ' ' || e.LastName as "Sales Rep"
```

```
FROM SalesQuery s  
INNER JOIN CustomerQuery c ON c.CUST_NO = s.CUST_NO  
INNER JOIN Employee e ON e.EmpNo = s.SALES_REP
```

## Chapter 17: Understanding Local SQL 499

The second alternative way to add datasets components to an FDLocalSQL's DataSets property is to add code to the FDLocalSQL's OnGetDataSet event handler. When this event handler is assigned, the FDLocalSQL component calls this method for each table reference in the SQL it is trying to process. The name of the table in the query is passed in the AName parameter, and you respond by assigning the associated dataset to the ADataSet parameter. An example of an OnGetDataSet event handler is shown later in this chapter.

### Activating Local SQL Queries at Runtime

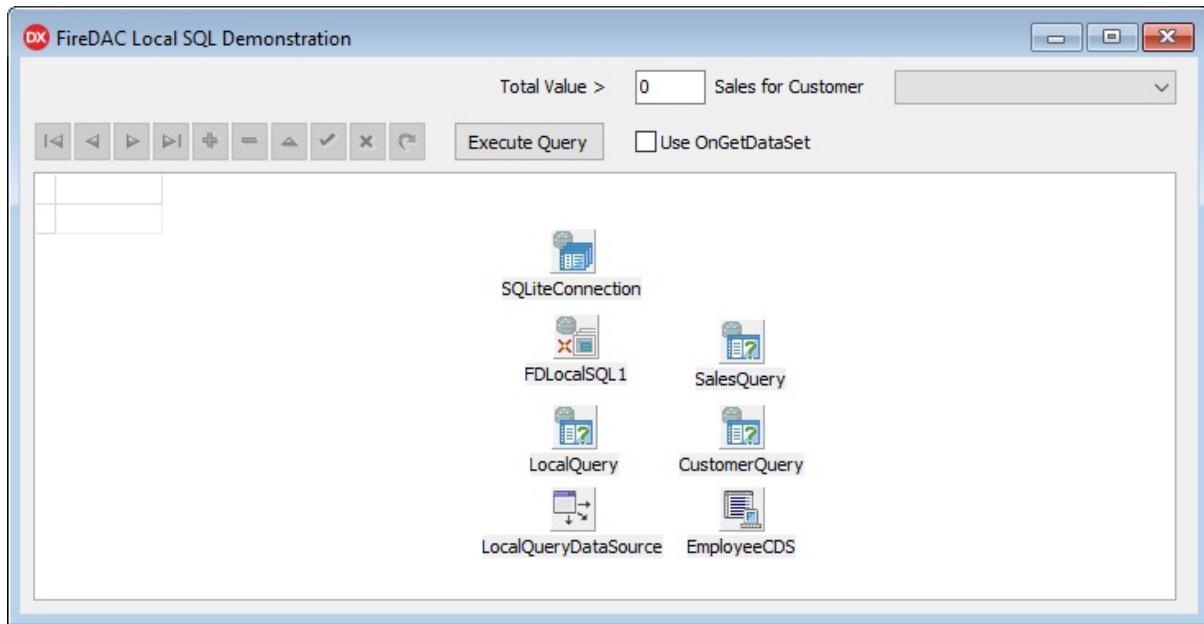
One of the major uses of Local SQL is to create ad hoc queries at runtime.

Doing so is not really different than what you do at design time, just so long as you activate the various components in the correct order. Specifically, the FDLocalSQL component must be active before you attempt to execute the local query.

Another consideration is that you are likely to request input from the user, and use that data in your queries to create flexibility. Since local queries have nearly all of the power provided for by SQLite, including the ability to delete records, it is important that you treat the data provided by the end user in the same fashion as you would in queries you execute against your underlying database.

Specifically, you should use parameters when you want to include data obtained from the user to prevent SQL injection.

The FDLocalSQL project, which is based on the project that you have created here in this chapter, includes a couple of controls that permits the user to execute queries that produce filtering on the result set returned by the LocalQuery FDQuery. The main form of this project is shown in Figure 17-8.



## 500 Delphi in Depth: FireDAC

**Figure 17-8: The main form of the FDLocalSQL project**

*Code: The FDLocalSQL project can be found in the code download. For more information, see Appendix A.*

Initially, all of the data access components are not active. Activation of SalesQuery, CustomerQuery, and EmployeeCDS occurs when the form is first created. This is also when the combobox is loaded with customer names, and the FDLocalSQL component is made active.

Because this project wants to expose some flexibility in the local query, the LocalQuery component's query includes both a parameter as well as a macro.

This can be seen in the following code, which is the form's OnCreate event handler:

```

procedure TForm1.FormCreate(Sender: TObject);
begin
  SalesQuery.Open();
  CustomerQuery.Open();
  EmployeeCDS.LoadFromFile( DataPaths.SamplePath +
    '\employee.xml' );
  cbxCustomerList.Clear;
  cbxCustomerList.Items.Add('All');
  while not CustomerQuery.Eof do
  
```

```
begin
  cbxCustomerList.Items.Add(
    Chapter 17: Understanding Local SQL 501
    CustomerQuery.FieldName( 'CUSTOMER' ).AsString );
  CustomerQuery.Next;
end;
  cbxCustomerList.ItemIndex := 0;
  SQLiteConnection.Connected := True;
  FDLocalSQL1.Active := True;
  // This query is parameterized, and uses macro substitution
  LocalQuery.SQL.Text := 'SELECT s.PO_NUMBER, c.Customer, ' +
  ' 
  s.ORDER_DATE, s.TOTAL_VALUE, ' +
  ' 
  e.FirstName || " " || ' +
  ' 
  e.LastName as "Sales Rep" '+
  ' FROM SalesQuery s '+
  ' INNER JOIN CustomerQuery c '+
  ' 
  ON c.CUST_NO = s.CUST_NO '+
  ' INNER JOIN EmployeeCDS e '+
  ' 
  ON e.EmpNo = s.SALES REP '+
  ' WHERE s.TOTAL_VALUE >= :v '+
  ' 
  &MoreWhere';
  cbxUseOnGetDataSetClick( nil );
end;
```

The last line of the preceding code makes a call to the OnChange event handler of the checkbox named cbxUseOnGetDataSet. This event handler, shown here,

toggles between the code using the DataSets property of the FDLocalSQL component to identify the datasets named in the query, and the OnGetDataSet event handler:

```
procedure TForm1.cbxUseOnGetDataSetClick(Sender: TObject);
begin
  if cbxUseOnGetDataSet.Checked then
    begin
      FDLocalSQL1.DataSets.Clear;
      FDLocalSQL1.OnGetDataSet := FDLocalSQL1GetDataSet;
    end
  else
    begin
      FDLocalSQL1.OnGetDataSet := nil;
      FDLocalSQL1.DataSets.Add.DataSet := SalesQuery;
      FDLocalSQL1.DataSets.Add.DataSet := CustomerQuery;
      FDLocalSQL1.DataSets.Add.DataSet := EmployeeCDS;
    end;
  end;
```

## 502 Delphi in Depth: FireDAC

The following is the OnGetDataSet event handler:

```
procedure TForm1.FDLocalSQL1GetDataSet(ASender: TObject;
  const ASchemaName, AName: string; var ADataSet: TDataSet;
  var AOwned: Boolean);
begin
  if AName = ‘SalesQuery’ then
    ADataSet := SalesQuery
  else if AName = ‘CustomerQuery’ then
```

```
ADataSet := CustomerQuery  
else if AName = ‘EmployeeCDS’ then  
ADataSet := EmployeeCDS;  
end;
```

*Note: If you are running this project in the debugger, and select the checkbox to use OnGetDataSet, you will get a series of exceptions when you click the Execute Query button. Specifically, there will be one exception for each call to OnGetDataSet. This is an internal exception, and does not reflect an actual error. At runtime those exceptions will be suppressed internally, as well as when you run the project without debugging.*

There is only one more event handler in this project, and that is the one associated with the button labeled Execute Query. From within this event handler, a value is assigned to the macro depending on what the user selected in the combobox, and the parameter is also assigned a value. Macro substitution was used here because the macro is replaced with either an empty string or a complex SQL segment. This replacement cannot be performed using a simple

parameter. (For information on macro substitution, please see *Chapter 14, The SQL Command Preprocessor*.) This event handler is shown here:

```
procedure TForm1.btnExitQueryClick(Sender: TObject);  
begin  
LocalQuery.Close;  
if cbxCustomerList.ItemIndex = 0 then  
LocalQuery.MacroByName( ‘MoreWhere’ ).AsRaw := ”  
else  
LocalQuery.MacroByName( ‘MoreWhere’ ).AsRaw :=  
‘AND c.Customer = ” + cbxCustomerList.Text + ”;’;  
LocalQuery.ParamByName( ‘v’ ).AsString := edtTotalOver.Text;  
LocalQuery.Open();
```

The screenshot shows a Delphi application window titled "FireDAC Local SQL Demonstration". At the top, there are two input fields: "Total Value >" with the value "0" and a dropdown menu "Sales for Customer" set to "All". Below these are several control buttons and an "Execute Query" button, which is highlighted with a blue border. To the right of the Execute Query button is a checkbox labeled "Use OnGetDataSet". The main area contains a grid table with the following columns: PO\_NUMBER, CUSTOMER, ORDER\_DATE, TOTAL\_VALUE, and Sales Rep. The data rows are as follows:

PO_NUMBER	CUSTOMER	ORDER_DATE	TOTAL_VALUE	Sales Rep
V9324320	Signature Design	8/16/2012	\$0.00	Michael Yanowski
V9346200	Buttle, Griffith and Co.	12/31/2012	\$0.00	K. J. Weston
V93C0120	DataServe International	3/22/2012	\$47.50	Claudia Sutherland
V93B1002	Dyno Consulting	9/20/2012	\$100.02	Jacques Glon
V93J2004	MPM Corporation	10/30/2012	\$210.00	Takashi Yamamoto
V93F0020	Max	10/10/2012	\$490.69	Luke Leung
V93F2051	3D-Pad Corp.	12/18/2012	\$999.98	Jacques Glon
V93N5822	GeoTech Inc.	12/18/2012	\$1,500.00	Jacques Glon
V94S6400	Dynamic Intelligence Corp	1/6/2013	\$1,980.72	Pierre Osborne
V92F3004	3D-Pad Corp.	10/16/2011	\$2,000.00	K. J. Weston
V93I4700	Lorenzi Export, Ltd.	10/27/2012	\$2,693.00	Roberto Ferrari
V92J1003	MPM Corporation	7/27/2011	\$2,985.00	Luke Leung

## Chapter 17: Understanding Local SQL 503

**end;**

Figure 17-9 shows how it appears when it is run and the Execute Query button is pressed without restricting records.

### **Figure 17-9: The Execute Query button is pressed to run the query without restricting records**

After having selected to display only records associated with 3D-Pad Corp. in the combobox, and records with a total value of over \$1000, pressing the Execute Query button produces the result set shown in Figure 17-10.

The screenshot shows the same Delphi application window as Figure 17-9, but with different filter settings. The "Total Value >" field now contains "1000", and the "Sales for Customer" dropdown is set to "3D-Pad Corp.". The "Execute Query" button is again highlighted with a blue border. The resulting grid table shows only three records that meet both criteria:

PO_NUMBER	CUSTOMER	ORDER_DATE	TOTAL_VALUE	Sales Rep
V92F3004	3D-Pad Corp.	10/16/2011	\$2,000.00	K. J. Weston
V93F3088	3D-Pad Corp.	8/27/2012	\$10,000.00	Jacques Glon
V93F2030	3D-Pad Corp.	12/12/2012	\$450,000.49	Jacques Glon

## **Figure 17-10: An updated query has been executed**

There are other ways to produce the effects that we saw here. We could have used parameterized queries against a database that included all tables of interests, and we could have also used filtering to achieve some of these effects, but there are advantages to this Local SQL approach. First, once the data is loaded, we could have done all sorts of interesting things with the data without any further roundtrips to the server.

Second, we could have gotten very creative with the SQL, producing results from code at runtime that would otherwise be impossible. For example, our queries could have involved complex calculations using FireDAC scalar functions that would have otherwise needed to be anticipated in advance with calculated fields.

A third advantage is that Local SQL permits you to execute heterogeneous SQL

queries across two or more physical databases. Local SQL is the only way that you can do this.

### **Some Comments on FDLocalSQL**

I've worked with the FDLocalSQL component for some time, and have found it to be a bit challenging to work with from time to time. Specifically, there have been times when I have changed the dataset definitions that appear in the DataSets property, and this has resulted in both design-time and runtime errors.

#### **Chapter 17: Understanding Local SQL 505**

These errors usually indicate that the names that are assigned to the datasets in the DataSets property are invalid.

These errors are frustrating, and can be difficult to correct. In fact, I've had situations where I have had to remove the offending FDLocalSQL component and replace it with a new one, which I then have to configure over again.

However, this is the only way that I have managed to correct these errors in some cases. My recommendation is that if you run into issues like these, that you simply replace the offending component rather than fight it. You may likely save yourself some significant time.

Finally, I want to mention that this chapter has not been an exhausted survey of Local SQL. It is intended to get you started, but there are properties and methods that have not been discussed. If you find Local SQL to your liking,

you may want to refer to the online help to explore additional capabilities available to you.

Appendix A Code Download, Database Preparation, and Errata 507

## **Appendix A**

### **Code Download,**

### **Database Preparation,**

### **and Errata**

Many code samples have been shown throughout this book, and the source code for these projects is available online for you to download. Most of these examples make use of databases and files that ship with Delphi or InterBase, and you may need to make some modifications to Delphi or one of the sample files before you can successfully run these projects. This appendix discusses how to download and install the code samples, as well as how to enable data access.

The final section in this appendix describes where you can find updated information about this book, in case errors or issues are discovered after its publication.

### **Code Download**

The code samples associated with this book are found in a zip file that can be downloaded from the following URL:

<http://www.JensenDataSystems.com/firedacbook/firedacbookcode.zip>

After you download this file you should unzip it into a single directory. When you are done that directory will contain approximately 40 folders.

Many of the projects use a data module and unit from the Shared Files folder. This folder is always assumed to be in a directory parallel to the folder in which the project that uses these files resides. For example, the FDSaveAndLoad

project uses files in the Shared Files folder. The FDSaveAndLoad folder and the Shared Files folder should be in the same subdirectory. For example, if you

508 Delphi in Depth: FireDAC

unzipped the code samples zip file into a folder named FireDAC on your C drive, the FDSaveAndLoad folder will be located in the following directory:

c:\FireDAC\FDSaveAndLoad

and the Shared Files folder will be located here:

c:\FireDAC\Shared Files

Likewise, some of the projects in the code samples use a saved FDMemTable named BigMemTable.xml, and that file can be found in the BigMemTable folder. Just as it is with the Shared Files folder, the BigMemTable folder must be in a directory parallel to the folder in which the project that uses BigMemTable.xml is located.

*Note: The BigMemTable.xml file contains fictional data for use in testing FDMemTable performance, and is copyrighted by Jensen Data Systems, Inc.*

*For more information, consult the file named About BigMemTableXml.txt , which is located in the same folder as BigMemTable.xml.*

These projects are specifically designed for Delphi or RAD Studio XE6 and higher. While you can compile many of them in Delphi XE5 (the first version of FireDAC that employed the FD prefix for component names), there are a lot of projects that require units that were not available in Delphi XE5, and therefore will not compile. These units include FireDAC.Stan.ExprFuncs, FireDAC.Stan.StorageBin, FireDAC.Stan.StorageXML, and FireDAC.Stan.StorageJSON.

In addition, you will need to use the Professional version of Delphi, or higher. These versions of Delphi include the FireDAC InterBase driver, which is used by a large number of the sample projects.

Even though all of these projects will compile in Delphi XE6, my recommendation is that you use one of the latest versions of Delphi, certainly no earlier than Delphi 10 Seattle. A number of important features were introduced in Delphi 10 Seattle. For example, if you want to employ cached updates, I recommend that you use no version earlier than Delphi 10 Seattle, as this is the version in which UpdateOptions.AutoCommitUpdates was introduced. You can

find a detailed discussion of the importance of this property in *Chapter 16, Using Cached Updates*.

Appendix A Code Download, Database Preparation, and Errata 509

## **Database Preparation**

Many of the sample projects employ one of several InterBase databases that ship with Delphi. This will require that you have installed the InterBase server (the free InterBase Developer Edition is sufficient) before you can run these projects.

## **Installation**

InterBase must be installed and started before you can successfully use these projects. If you have only just installed RAD Studio and InterBase, and have not yet re-booted your computer, InterBase is probably not running. Before you can use InterBase, you will need to either re-boot your computer, or launch the Services applet from the Administrative Tools section of the Control Panel, and then start the InterBase service (or the InterBase Guardian service).

If these services are not configured to start automatically, you may want to consider changing them to start automatically. This can be done within the services applet by right-clicking the InterBase Guardian service and settings its Startup Type property to Automatic.

In some versions of Delphi, InterBase will need to be started manually each time you boot your computer. If you get an error the first time you try to run one of the sample projects after re-booting, open the Services applet and re-start the InterBase Guardian service. I've found this step necessary on occasion.

## **The DataPaths Unit**

Many of the sample projects in the code download rely on sample databases and files that ship with Delphi and RAD Studio. Unfortunately, where these files are installed on your computer depends on which version of Delphi you are using.

When I designed these sample projects, I specifically wanted to make it easy for you to accommodate different versions of Delphi, and I accomplished this by using a unit named DataPaths.pas. This unit, which is located in the Shared Files folder of the code download, defines three constants that are used by the sample projects to locate the needed data. In addition, this unit validates one of these paths, and also assembles a TStringList that defines the connection parameters for the most used database (employee.gdb).

The following is a partial listing of this unit (some of the conditional statements have been omitted for brevity).

```
unit DataPaths;
```

```
510 Delphi in Depth: FireDAC
```

**interface**

**uses** System.SysUtils, System.Classes;

**type**

EFireDACBookException = class( Exception );

**var**

ConnectionParams: TStrings;

**const**

{\$IF CompilerVersion > 31.0} //Delphi 10.2 Tokyo and above

IBEmpPath = 'C:\Users\Public\Documents\Embarcadero\Studio\19.0' +

'Samples\Data\employee.gdb';

IBDemoPath = 'C:\Users\Public\Documents\Embarcadero\Studio\19.0' +

'Samples\Data\dbdemos.gdb';

SamplePath =

'C:\Users\Public\Documents\Embarcadero\Studio\19.0\Samples\Data';

{\$ELSE}

IBEmpPath = 'C:\ProgramData\Embarcadero\InterBase\gds\_db' +

'examples\database\employee.gdb';

{\$ENDIF}

{\$IF CompilerVersion = 31.0} //Delphi 10.1 Berlin

IBDemoPath = 'C:\Users\Public\Documents\Embarcadero\Studio\18.0' +

'Samples\Data\dbdemos.gdb';

SamplePath =

'C:\Users\Public\Documents\Embarcadero\Studio\18.0\Samples\Data';

{\$ENDIF}

{\$IF CompilerVersion = 30.0} //Delphi 10 Seattle

IBDemoPath = 'C:\Users\Public\Documents\Embarcadero\Studio\17.0' +

'Samples\Data\dbdemos.gdb';

SamplePath =

'C:\Users\Public\Documents\Embarcadero\Studio\17.0\Samples\Data';

```
{$ENDIF}

// ... more definitions appear below here

var
BigMemTablePath: string;

procedure ValidatePath( Path: string );

implementation

uses System.IOUtils;

procedure ValidatePath( Path: string );

var
Found: Boolean;
Msg: string;

Appendix A Code Download, Database Preparation, and Errata 511

begin

if Path.EndsWith('.gdb') then
  Found := TFile.Exists( Path )
else
  Found := TDirectory.Exists( Path );
if not Found then
  begin
    Msg := Format('Invalid Path: ' +
      '%s was not found. Please read Appendix A ' +
      'in the book "Delphi in Depth: FireDAC" ' +
      'for information on locating the sample ' +
      'data directories, and then update ' +
      'the constants declared in the DataPaths ' +
      'unit, which is located in the SharedFiles ' +
      'directory of the code download', [ Path ] );
    raise EFireDACBookException.Create( Msg );
  end;
```

**end;**

**initialization**

```
ConnectionParams := TStringList.Create;  
DataPaths.ValidatePath( IBEMpPath );  
ConnectionParams.Add('Database=localhost:' +  
DataPaths.IBEMpPath);  
ConnectionParams.Add('User_Name=sysdba');  
ConnectionParams.Add('Password=masterkey');  
ConnectionParams.Add('Protocol=TCPIP');  
ConnectionParams.Add('DriverID=IB');  
BigMemTablePath := ExtractFilePath( ParamStr( 0 ) );  
BigMemTablePath := BigMemTablePath +  
'..\BigMemTable\BigMemTable.Xml';
```

**finalization**

```
if Assigned( ConnectionParams ) then  
ConnectionParams.Free;  
end.
```

The three constants defined in this unit are IBEMpPath, IBDemoPath, and SamplePath. IBEMpPath points to the employee.gdb database, which is used by a majority of the sample projects. SamplePath points to the directory in which are large number of sample files are found, including Paradox tables, InterBase database files, and ClientDataSet files. IBDemoPath points to the demos.gdb InterBase database in this same folder.

## 512 Delphi in Depth: FireDAC

If you cannot run some of these sample projects because your installation does not match the paths defined in DataPaths, locate the correct folders and use this information to update the constant definitions in the DataPaths unit.

### Using SharedDMVcl

While the DataPaths unit defines the locations for the code samples, it is the SharedDMVcl and SharedDMFmx units that define a FireDAC connection that

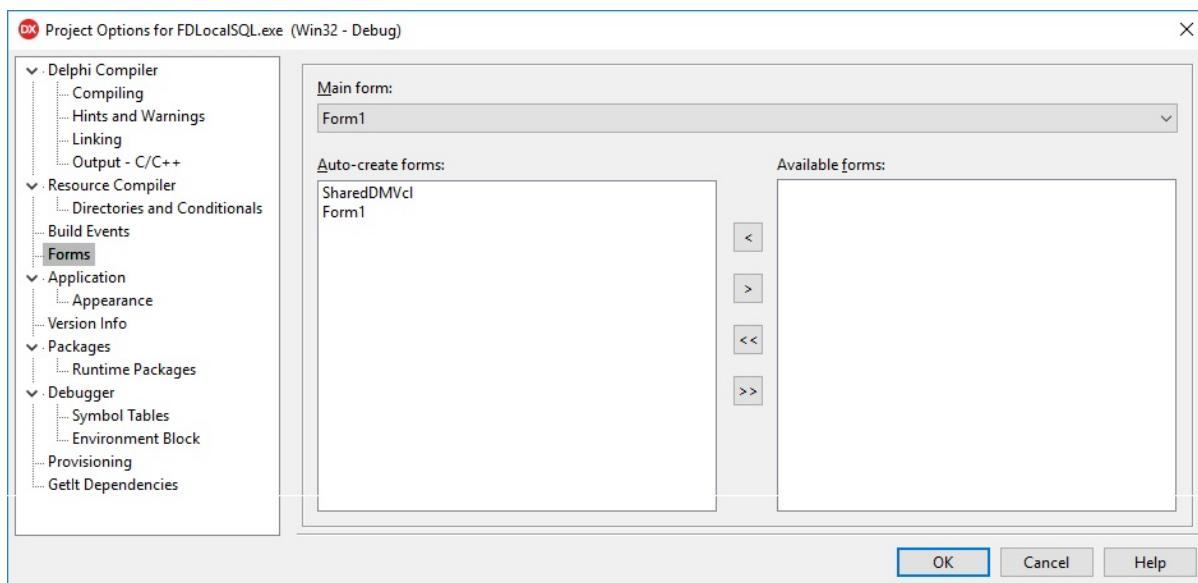
is used by many of the sample projects. These two units define data modules,

and you are welcome to use these units, in conjunction with DataPaths.pas, in sample and test projects that you create as you work through the examples in this book.

If you add SharedDMVcl and DataPaths to an existing project, there is one additional step that you must take in order to successfully use the FDConnection on these data modules. You must ensure that the data module is created before creating any form that needs to use the connection.

The following steps demonstrate one of the ways that you can ensure that the data module is created before the forms that need to use the FDConnection on the data module:

1. After adding SharedDMVcl (or SharedDMFmx) and DataPaths to an existing project, select Project | Options from Delphi's main menu (or press Ctrl-Shift-F11) to display the Project Options dialog box.
2. Select the Forms tab on the Project Options dialog box.
3. Move the data module to the top of the Available Forms list, as shown in Figure A-1.
4. Click OK to save your updated project options.



## Appendix A Code Download, Database Preparation, and Errata 513

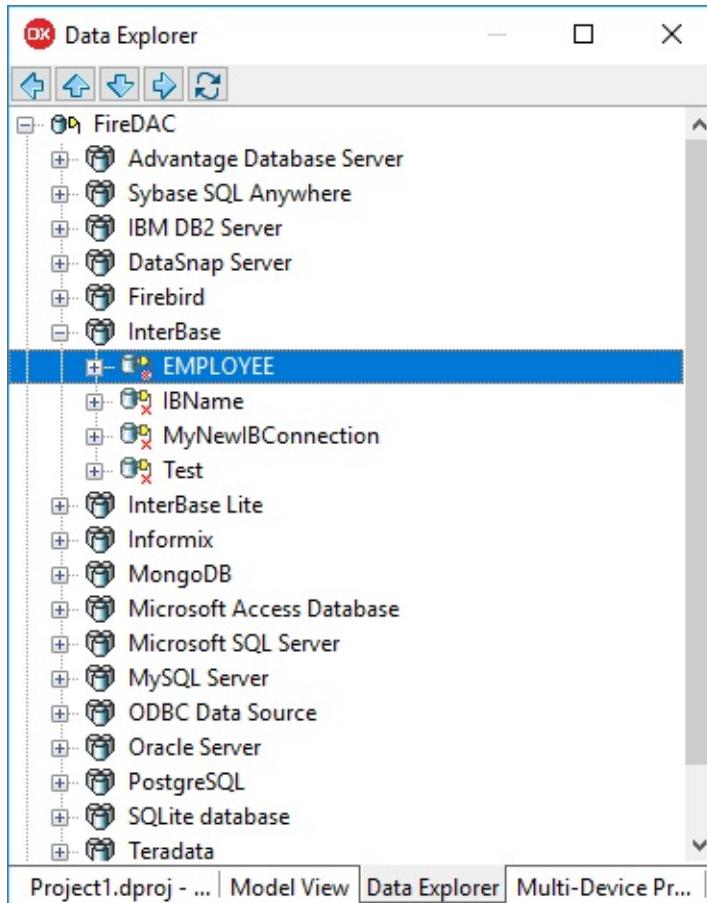
### **Figure A-1: The data module will be created before Form1**

#### **The EMPLOYEE Named Connection**

There are a couple of sample projects and a few demonstrations in this book that uses the EMPLOYEE named FireDAC connection to the employee.gdb

database. This named connection can be found in the FireDAC | InterBase section of the Data Explorer in Delphi's IDE, as shown in Figure A-2. While testing these sample projects with several different installations of Delphi 10.2

Tokyo, I discovered that at least one installation had an invalid path to the employee.gdb database configured in the EMPLOYEE named connection.



## 514 Delphi in Depth: FireDAC

**Figure A-2: The EMPLOYEE named connection is selected in the Data Explorer**

If you cannot open the EMPLOYEE named connection in the Data Explorer, you may have an invalid path you need to correct in the Database parameter of this connection definition. (You should be able to open this connection using the username **sysdba** and the password **masterkey**.) To correct the connection configuration, right-click the EMPLOYEE named connection in the Data

Explorer and select Modify. Correct the path to the employee.gdb database in the Database parameter of the FireDAC Connection Editor. See DataPaths.pas

for the correct location of this database for your version of Delphi.

## Appendix A Code Download, Database Preparation, and Errata 515

### InterBase UDF Definitions

During the testing of the code samples in *Chapter 14, The SQL Command Preprocessor*, I discovered that some of the FireDAC scalar functions did not work against the InterBase database. For example, the LENGTH function raised an exception noting that the STRLEN user defined function (UDF) was not

found.

It turns out that an SQL script that defines some InterBase UDFs did not get executed during my installation of Delphi. If you run into this problem, you will need to locate this SQL script and run it against any database against which you want to run the associated FireDAC scalar functions.

The script is named ib\_udf.sql, and it is located in the gds\_db\examples folder under the InterBase installation directory. On my machine, this directory was found in the following location (ignore the carriage return):

c:\ProgramData\Application Data\

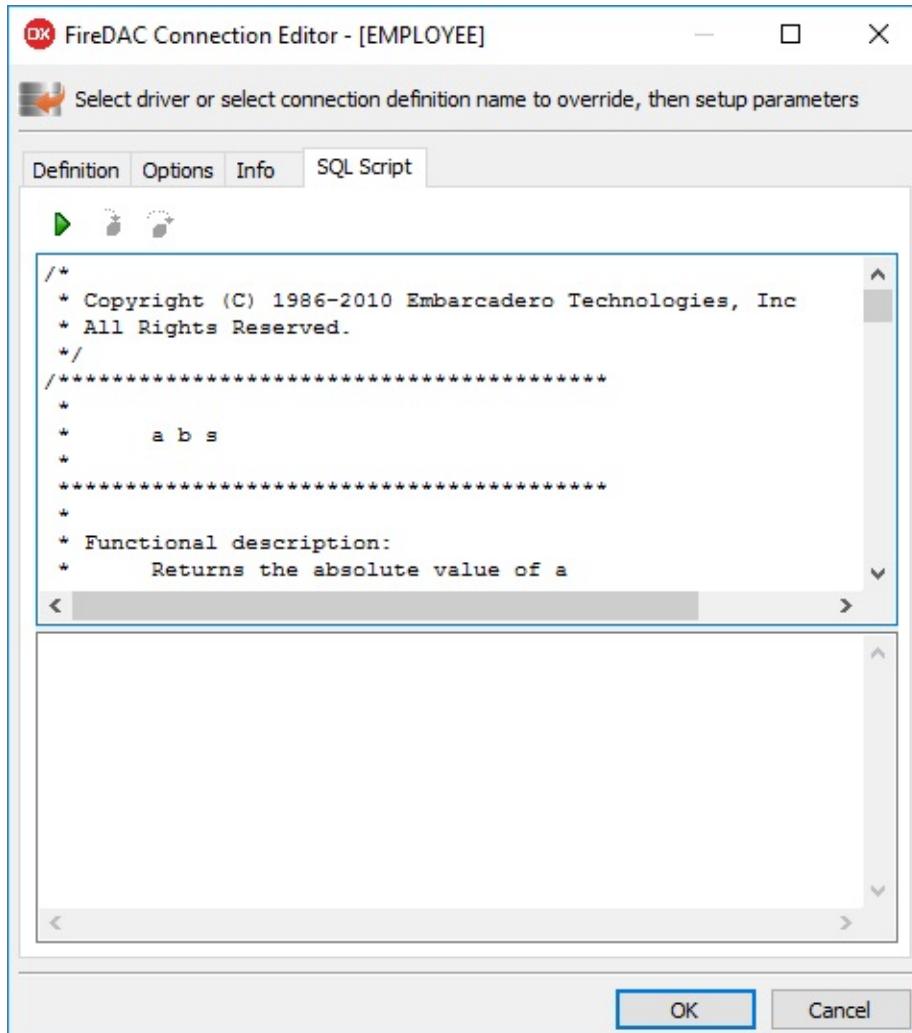
Embarcadero\InterBase\gds\_db\examples

Note that the c:\ProgramData folder is a hidden folder by default. If you do not see this folder from the Windows Explorer, you can type c:\ProgramData\ into the address line of the Windows Explorer and press Enter, and then the folders under that directory will become visible. You can also change your general folder viewing options to show hidden folders, files, and drives, and then ProgramData will always be visible from Windows Explorer.

If you need to execute the ib\_udf.sql script, you can do so easily for the employee.gdb database used in many of the projects in the code download by using the following steps:

1. Using the Data Explorer, expand the FireDAC node, and then the InterBase node to expose the named InterBase connections.
2. Right-click the EMPLOYEE node and select Modify to display the FireDAC Connection Editor for this named connection.
3. Using Notepad, or some other file editor, open the ib\_udf.sql script file from the directory given earlier in this section.
4. Copy the contents of this script file to the Windows clipboard.

5. Select the SQL Script tab of the FireDAC Connection Editor, and paste the contents of the script into the top pane, as shown in Figure A-3.



## 516 Delphi in Depth: FireDAC

**Figure A-3: The ib\_udf.sql script has been pasted into the SQL Script pane, ready for execution**

6. Click the green arrow in the displayed toolbar to execute the query.
7. You can now click the OK button to close the FireDAC Connection Editor. Your FireDAC scalar functions should now work properly.

Appendix A Code Download, Database Preparation, and Errata 517

## Errata

All of us who have worked on this book have tried to ensure that the descriptions contained here are accurate. Nonetheless, there are bound to be some errors that will come to light after this book has been published.

If substantive errors are discovered after this book has been published, we will post corrections on the errata page associated with this book. This page can be found at the following URL:

<http://www.JensenDataSystems.com/firedacbook/errata>

We suggest that you visit this page from time to time, in case a correction that applies to the work that you are doing have been posted.

Index 519

## **Index**

ApplyRange

—!—

defined, 228

using, 228–31

! character in macros, 400, 401

Array DML, 437

array limits, 437

—\$—

databases supporting, 426

`$( name)`, 300

defined, 8, 425

errors, and, 436

OnExecuteError event handler, 434

—&—

Params.ArraySize property, 430

& character in macros, 400

Array DML mode, 434

database, supporting, 435

asynchronous queries, 9

—?—

auto increment fields

? character in macros, 401

cached updates, and, 483

AutoCommitUpdates

importance of, 458

**—A—**

automatic connection recovery, 12

ActiveStoredUsage, 132

AVG, 266

AfterClose, 149

AfterOpen, 149

**—B—**

aggregate fields. *See* aggregates

AggregateField

batch command processing. *See* Array

Active property, 269

DML

Visible property, 269

batch move, 13

aggregates, 293

BDE

AggregateFields versus, 261

converting to FireDAC, 6

Aggregates versus, 261

BDE (Borland Database Engine)

creating, 261–66, 272–74

defined, 5

creating at runtime, 277–79

BigMemTable.xml file, 176, 198, 221, 232

defined, 260, 261

BindNavigator, 92

descriptive statistics, 261  
BindSourceDB class, 92  
expressions, defining, 266  
Bof (beginning-of-file), 154  
GroupingLevel, 268  
Borland SQL Links for Windows  
IndexName, 268  
defined, 6  
performance and, 272  
briefcase model, 11, 297, 439, 448, *See*  
AggregatesActive, 269  
*also persisting data*  
Append, 162  
brute force method, 453  
AppendRecord, 162  
520 Delphi in Depth: FireDAC

## —C—

centralized cached updates, 10, 486, *See*  
*also cached updates*  
cached updates, 10, 486  
CheckUpdatedRecords, 125  
advantages of, 439  
CloneCursor method, 379  
applying updates, 467  
cloned cursors. *See also* FDMemTables,  
ApplyUpdates method, 467  
cloning cursors  
AutoCommitUpdates property, 458  
cached updates and, 379

basics of, 449  
uses for, 369  
briefcase model, 297  
Columns editor, 89, 384  
brute force method, 451, 453  
conditional substitution, 411  
CachedUpdates property, 441  
connection definition file  
canceling a specific change, 449  
custom location for, 37  
canceling all changes, 450  
defining at runtime, 39  
canceling last change, 450  
connection pooling, 9, 28  
centralized model, 486  
threads and, 29  
centralized model and master-detail  
ConnectionDefFileAutoLoad property, 37  
tables, 481  
ConnectionDefFileName property, 37, 38  
centralized versus decentralized, 441  
ConnectionName property, 40  
ChangeCount property, 442  
connections, 15  
clearing the cache, 451  
ConnectionString property, 27  
cloned cursors and, 379  
constant substitution, 408  
deleted records and, 307

controls  
detecting cache state, 444  
data bound, 92  
detecting field-level changes, 447  
convert scalar function, 423  
displaying deleted records, 445  
CopyDataSet method, 362  
entering and exiting, 442  
COUNT, 266  
error-related event handlers, 469  
errors, and, 469  
FilterChanges property, 445, 450

**—D—**

LoadFromFile, 448  
data binding, 92  
LoadFromStream, 448  
Data Explorer, 82, 83  
managing the cache, 452  
data type mapping, 11  
OnReconcileError event handler, 469  
Database Explorer, 29  
OnUpdateError event handler, 469  
creating a named connection with, 30  
OnUpdateRecord event handler, 467  
databases  
persisting, 449  
supporting more than one, 397  
savepoints, 452  
dataset

SaveToFile, 448  
record state of, 151  
SaveToStream, 448  
DataSetField, 379  
UpdatesPending property, 442  
datasets. *See also* FireDAC datasets  
UpdateStatus property, 444

Append method, 162  
what has changed, 447  
Cancel method, 161  
Calculated fields, 284, 293  
current record of, 148  
Calculating performance, 152

Edit method, 161  
Cancel method, 161  
editing, 160

CancelRange, 231  
event handlers of, 149  
FieldByName, 146  
Index 521  
FieldByNumber, 146

## —E—

Fields, 146  
FindField, 146  
Edit method, 161  
IndexName property, 179  
editing, 160–63  
Insert method, 162  
Append, 162

InsertRecord method, 162  
AppendRecord, 162  
navigating, 161  
Cancel method, 161  
OnReconcileError, 163  
Insert, 162  
OnUpdateError, 163  
InsertRecord, 162  
Post method, 161  
Post method, 161  
scanning, 199  
Editing datasets, 160  
DataSnap, 7  
EditKey, 207  
FireDAC and, 219  
EditRangeEnd, 228  
datasource  
EditRangeStart, 228  
OnDataChange, 149  
EMPLOYEE database sample, 165  
OnStateChange, 149, 151  
EMS, 7  
OnUpdateChange, 149  
EnableControls method, 158  
date/time scalar functions, 421, *See*  
Eof (end-of-file), 154  
FireDAC scalar functions  
escape sequences, 423  
dBase tables, accessing with FireDAC, 41

example sorting DBGrid data, 190–94

DBCtrlGrid, 90

extract, transform, and load (ETL)

dbExpress defined, 5

operations, 9, 426

DBGrids, 89

columns editor, 89

—F—

Lookup up fields, using in, 288

sorting on-the-fly, 194

FDAdvancedFilters project, 234

suppressing columns in, 384

FDAggregateDemo project, 272

DBNavigator, 86

FDAggregatesAndGroupState project, 262

controlling button display, 87

FDArrayDML project, 426

hints, editing, 88

FDBasicCache project, 440

ShowHint, 87

FDBottomResourceOptions. *See*

decentralized cached updates, 10, *See also*

ResourceOptions

cached updates

FDBottomUpdateOptions, 70, *See*

centralized model, versus, 480

UpdateOptions

delayed updates. *See* cached updates

FDCachedMasterDetail project, 478, 482

Delphi Developer Days, 6

FDCachedUpdatesErrors project, 454

DisableControls method, 158

FDCalcFields project, 281, 284

distributed applications

FDCloningCachedCursors project, 376

FireDAC and, 219

FDCommand components, 99

dsEdit state, 163

using, 132

dsSetKey state, 228

FDConnection, 47

dynamic fields, 261

ConnectionMetaDataIntf property, 434

defined, 260

ConnectionMetaDataIntf.ArrayExecMo

FieldOptions property and, 296

de property, 434

Dynamic SQL. *See* SQL Command

ConnectionName property, 40

Preprocessor

FDConnectionDefs.ini, 33, 36

FDCopyClientDataSet project, 354

522 Delphi in Depth: FireDAC

FDCopyDataSetQuery2Query project, 359

defining using FieldDefs at runtime,

FDCopyingDataSets project, 350, 362

389

FDDesigntimeNestedFields project, 381

defining structure using FieldDefs, 339

FDExplicitTransactions project, 134

defining using FieldDefs at design time,  
FDExplorer.exe, 33  
389

FDFetchOptions. *See* FetchOptions

defining using Fields, 347

FDFieldDefsDesignTime project, 336

defining using Fields at design time, 339

FDFilter project, 168, 220, 221

defining using Fields at runtime, 347

FDFormatOptions. *See* FormatOptions

editing at design time, 395

FDIndexes, 188

loading from datasets, 364

FDIndexes project, 174

master-detail clone, 372

FDLocalSQL components. *See* Local SQL

nested datasets, 395

DataSets property, 504

nested datasets, defining using Fields at  
OnGetDataSet, 501  
runtime, 393

OnGetDataSet, errors and, 502

nested datasets, defining with FieldDefs

FDLocalSQL project, 499  
at runtime, 380–84, 385

FDMacros project, 412

nested datasets, limitations of, 395

FDMacroSubstitution project, 402  
nested datasets, referring to, 394  
FDManager components, 36, 47  
the role of, 339  
ConnectionDefFileAutoLoad, 38  
using, 132  
ConnectionDefFileAutoLoad property,  
FDMoniCustomClientLink, 143  
37  
FDMoniFlatFileClientLink, 143  
ConnectionDefFileName property, 38  
FDMoniRemoteClientLink, 141, 143  
private connections, creating, 39  
FDNavigation project, 149, 150, 154, 155,  
FDMapRules, 61  
159, 160  
NameMask property, 61  
FDParadoxODBCDynamic project, 45  
PrecMax property, 61  
FDParam, 116, 117, 119  
PrecMin property, 61  
FDParams, 112  
ScaleMax property, 61  
FDPersistentConnection project, 39  
ScaleMin property, 61  
FDPrivateConnection project, 40  
SizeMax property, 61  
FDQuery components, 97, 99  
SizeMin property, 61

Active property vs ExecSQL, 103  
SourceDataType property, 61  
cloned cursors and, 331  
TargetDataType property, 61  
Command.State, 140  
TypeMask property, 61  
Execute method, 430  
FDMasterDetail project, 251  
executing, 103  
FDMasterDetailClone project, 372  
Params property, 218  
FDMemTables, 395  
Params.ArraySize property, 430  
cached updates, 486, *See also* cached  
FDRuntimeNestedFieldDefs project, 385  
ClientDataSets versus, 329  
FDRuntimeNestedFields project, 389  
cloning cursor, 379  
FDSaveAndLoad project, 301  
CopyDataSet method, 362  
FDSchemaAdapter component, 441, *See*  
CopyDataSet method versus Data  
*also* cached updates  
property, 367  
FDSearch project, 197  
Data property, 364  
FDSQLPreprocessor project, 399

defined, 329

FDStanStorageBINLink components, 302

Index 523

FDStanStorageJSONLink components, 302

Unidirectional property, 56

FDStanStorageXMLLink components, 302

FieldDefs

FDStoredProc

nested datasets, defining at runtime, 385

Active property vs ExecSQL, 106

FieldOptions

ExecSQL, 106

UpdatePersistent property, 295

executing, 108

FieldOptions property, 296

FDTable components, 99

PositionMode property, 295

Live Data Window, 102

fields. *See also* virtual fields

FDTAdapter components

aggregate fields, 279

using, 132

AsBoolean, 146

FDTemporaryConnection project, 26

AsFloat, 146

FDTopResourceOptions. *See*

AsInteger, 146

ResourceOptions

AsString, 146

FDTxOptions. *See* TransactionOptions  
Calculated versus InternalCalc, 280  
FDUpdateOptions. *See* UpdateOptions  
creating FDMemTable structure at  
FDUpdateSQL components, 129, 467  
runtime, 345  
defining the query for, 121  
CurValue property, 447  
parameter naming patterns for, 128  
data fields, 259  
Update SQL editor, 127  
DefaultExpression, 283  
FDUpdateSQLDemo project, 123  
described, 147  
FetchOptions, 58  
dynamic versus persistent, 260  
AutoClose property, 56  
FieldByName, 146  
AutoFetchAll property, 56  
FieldByNumber, 146  
Cache property, 57  
FindField, 146  
CursorKind property, 56  
FireDAC scalar functions, using in, 283  
DetailCascade property, 57  
IsNull, 146  
DetailDelay, 258  
nested datasets, defining at design time,  
DetailDelay property, 57

381

DetailDelete property, 481

nested datasets, defining at runtime, 393

DetailOptimize property, 57

NewValue property, 447

DetailServerCascade property, 58

OldValue property, 447

DetailServerDelete property, 481

OnGetText event handler, 275, 278

general, 56

Value, 146

Items property, 57, 102

virtual fields, 174, 296

items to cache, 57

Fields Editor

items to fetch, 57

navigator, 88

LiveWindowFastFirst property, 58

files

LiveWindowParanoic, 217

persisting FireDAC datasets to, 298

LiveWindowParanoic property, 58

filter expressions, 243

LiveWindowParanoic property, 102

miscellaneous functions, 242

master/detail, 58

null comparisons, 236

Mode property, 56, 199

string functions, 239

RecordCountMode, 217  
wildcards, escaping, 242  
RecordCountMode property, 56  
Filter property, 232, 246  
RecsMax property, 56  
Filtered property, 232, 243  
RecsSkip property, 56  
filtering, 258  
RowsetSize property, 56  
defined, 197, 217  
524 Delphi in Depth: FireDAC  
dynamic master-detail, 258  
CancelUpdates method, 450  
Filtered property, 243  
ChangeCount property, 442  
FilterOptions property, 244  
CloneCursor method, 379  
issues to consider, 218  
cloned cursor and cached updates, 379  
limiting dataset size versus, 218  
cloned cursors and, 331  
loading from data file or stream, 219  
cloning cursors, 370  
navigating and, 246–49  
CommitUpdates method, 451  
OnFilterRecord event handler, 245  
CopyDataSet method, 362  
performance, 219, 220, 232  
CopyDataSet method versus Data

ranges, 217  
property, 367  
ranges, together with, 249  
CopyDataSet, cached updates and, 362  
ranges, using, 222–27  
Data property, 364  
searching versus filtering, 217  
detecting field-level changes, 447  
using, 232–45  
displaying deleted records, 445  
FilterOptions property, 232, 244  
EditKey method, 205  
FindFirst method, 246  
error-related event handlers, 469  
FindLast method, 246  
executing at design time, 100  
FindNearest method, 201  
executing at runtime, 100  
FindNext method, 246  
FDMemTables, loading from, 364  
FindPrior method, 246  
FieldOptions property, 296  
FireDAC  
filter expressions, 243  
and dataset persistence, 10  
Filter property, 232  
Array DML, 8  
FilterChanges property, 445, 450  
batch move, support for, 13

Filtered property, 232, 243  
cached updates, support for, 10  
filtering, 218, 232–45  
cross platform support for, 7  
FilterOptions property, 232, 244  
data type mapping, support for, 11  
FindKey method, 204  
databases supported by, 7  
FindNearest method, 204  
features of, 6  
GetGroupState method, 274  
local SQL, support for, 11  
Goto method, 207  
native drivers, 7  
GotoNearest method, 207  
ODBC support, 7  
Indexes property, 190, 201  
support for Dynamic SQL, 8  
IndexFieldNames property, 192, 201  
supported Delphi versions, 8  
LoadFromFile method, 219, 316, 448  
FireDAC Client/Server Add-On Pack, 8  
LoadFromStream method, 219, 312, 448  
FireDAC Connection Editor, 16  
LocalSQL property, 505  
configuring, 17  
Locate method, 213  
connection information from, 20  
Lookup method, 215

setting options from, 19  
master-detail views, 258  
FireDAC datasets. *See also* datasets  
MasterFields property, 253  
ActiveStoredUsage, 101, 132  
MasterSource property, 253, 255  
AggregatesActive property, 272  
nested datasets, 369, 395  
ApplyRange method, 228–31  
OnCalcFields event handlers, 280  
ApplyUpdates method, 467  
OnFilterRecord event handler, 232, 245  
cached updates. *See* cached updates  
OnReconcileError event handler, 469  
CachedUpdates property, 441  
OnUpdateError event handler, 469  
CancelRange method, 231  
OnUpdateRecord, 121  
Index 525  
OnUpdateRecord event handler, 463  
DAYOFMONTH, 420  
performance and, 218, 219, 220  
DAYOFWEEK, 420  
ranges, 217  
DAYOFTYEAR, 420  
RecordCount property, 217  
DEGREES, 418  
RevertRecord method, 449  
errors when using, 417

SavePoint property, 452  
EXP, 418  
SaveToFile method, 10, 309, 448  
EXTRACT, 420  
SaveToStream method, 10, 322, 448  
filter expressions, using in, 243  
searching, 215  
FLOOR, 418  
SetKey method, 205  
HOUR, 420  
SetRange method, 222  
IF, 421  
structure, copying, 349, 353  
IFNULL, 421  
structure, merging, 354, 356  
IIF, 421  
UndoLastChange method, 449  
INSERT, 415  
UpdatesPending property, 442  
InternalCalc fields, using in, 283  
UpdateStatus property, 444  
LCASE, 416  
FireDAC defined, 5  
LEFT, 416  
FireDAC Explorer, 33  
LENGTH, 416  
creating named connections with, 33  
LIMIT, 421  
FireDAC Monitor, 10, 142

Local SQL, versus, 504  
FireDAC Query Editor, 118  
localization, and, 414  
using, 22  
LOCATE, 416  
FireDAC scalar functions, 423  
LOG, 418  
ABS, 417  
LOG10, 418  
ACOS, 417  
LTRIM, 416  
ASCII, 415  
MINUTE, 420  
ASIN, 418  
MOD, 418  
ATAN, 418  
MONTH, 420  
ATAN2, 418  
MONTHNAME, 420  
BIT\_LENGTH, 415  
NEWGUID, 422  
CEILING, 418  
NOW, 420  
CHAR, 415  
numeric scalar functions, 419  
CHAR\_LENGTH, 415  
OCTET\_LENGTH, 416  
CHARACTER\_LENGTH, 415  
PI, 418

CONCAT, 415  
POSITION, 416  
CONVERT, 423  
POWER, 418  
COS, 418  
QUARTER, 420  
COT, 418  
RADIANS, 418  
CURDATE, 419  
RAND, 419  
CURRENT\_DATE, 419  
REPEAT, 416  
CURRENT\_TIME, 419  
REPLACE, 416  
CURRENT\_TIMESTAMP, 420  
RIGHT, 416  
CURTIME, 420  
ROUND, 419  
DATABASE, 421  
RTRIM, 416  
databases and support for, 414  
SECOND, 420  
date/time scalar functions, 421  
SIGN, 419  
DAYNAME, 420  
SIN, 419  
526 Delphi in Depth: FireDAC  
SPACE, 416  
Round2Scale property, 63

special character processing, 423  
sorting, 64  
**SQRT**, 419  
SortLocale property, 64  
string/character scalar functions, 417  
SortOptions property, 64, 171  
**SUBSTRING**, 416  
string handling, 63  
system scalar functions, 422  
**StrsEmpty2Null** property, 62  
**TAN**, 419  
**StrsTrim** property, 63  
**TIMESTAMPADD**, 421  
**StrsTrim2Len** property, 63  
**TIMESTAMPDIFF**, 421  
values preprocessing, 63  
**TRUNCATE**, 419  
Found, 248  
**UCASE**, 416  
**USER**, 422

—G—

using in indexes, 182  
**WEEK**, 421  
GetBookmark method, 160  
**YEAR**, 421  
GotoBookmark method, 160  
FireDAC.Comp.UI unit, 25  
group state  
FireDAC.Stan.ExprFuncs unit, 182, 235

defined, 274  
FireDAC.Stan.StorageBin unit, 302  
using, 274–77  
FireDAC.Stan.StorageJSON unit, 302  
GroupingLevel, 268  
FireDAC.Stan.StorageXML unit, 302  
FireDAC.UI.Intf unit, 25

—I—

FireDAC.VCLUI.Wait unit, 25  
FireMonkey components  
IDAPI defined, 5  
LiveBindings and, 92  
identifier substitution, 409  
foreign key  
IndexDefs, 172  
defined, 285  
indexes, 195  
FormatOptions, 65  
creating Field-based at runtime, 180  
BCD and DataSnap compatibility, **62**  
creating using IndexDefs, 172  
CheckPrecision property, 63  
defined, 165  
data mapping rules, 61  
DescFields property, 175  
DataSnapCompatibility property, 62  
distinct indexes, 186  
default field formats, 65  
Distinct property, 185

DefaultParamDataType property, 63  
example creating on-the-fly, 190–94  
FmtDisplayDate, 65  
expression indexes, 183  
FmtDisplayDateTime property, 65  
Fields property, 174  
FmtDisplayNumeric property, 65  
Filter property, 187  
FmtDisplayTime property, 65  
filter-based indexes, creating, 188  
FmtEditNumeric property, 65  
FindIndex method, 180  
InlineContentSize property, 62  
FindIndexForFields method, 180  
MapRules property, 60  
FireDAC scalar functions, using in, 182  
MaxBcdPrecision property, 62  
IndexByName method, 180  
MaxBcdScale property, 62  
IndexFieldNames property, 172  
MaxStringSize property, 62  
InternalCalc fields and, 280  
OwnMapRules property, 60  
Items property, 180  
parameters default type, 63  
order and searching, 201  
quotation identifiers, 64  
persistent, 167, 188, 172–89  
QuoteIdentifiers property, 64

purpose of, 166–67  
Index 527  
ranges using, 222  
localization  
ranges, using, 222  
FireDAC scalar functions, and, 414  
searching with, 207, 200  
Locate method, 208  
selecting persistent indexes at runtime,  
login dialog box, 12  
180  
LoginPrompt property, 23, 25  
sort options, 171  
Lookup, 208  
sorting and temporary indexes, 167  
Lookup fields, 292  
temporary, 167  
creating, 287  
temporary indexes, limitations of, 172  
DBGrids, using in, 288  
temporary indexes, sorting, 167  
temporary, creating at runtime, 171

**—M—**

unique, 166  
user defined, 168  
macro substitution, 406, 502  
IndexFieldNames property, 168  
macros  
Insert, 162

assigning at design time, 405  
InsertRecord, 162  
assigning at runtime, 406  
InterBase  
master-detail relationships  
generators, and, 483  
cached updates, and, 481  
InterBase developer edition, 16  
cloned cursors and, 372  
InternalCalc fields, 284, 293  
examples, 251  
master-detail views, 258

—L—

defining, 251  
parameter-based, 257  
LDW. *See* Live Data Window  
range-based, defining, 254  
Live Data Window, 58, 102  
MAX, 266  
LiveBindings, 96  
MIN, 266  
defined, 91  
MoveBy property, 155  
FireMonkey support, 85  
MS Access tables, accessing with  
LiveBindings Designer, 92  
FireDAC, 41  
VCL support, 85  
LiveBindings Designer

—N—

displaying, 93  
LiveBindings Expression Editor, 95  
named connection definition file  
LoadFromFile, 219  
creating, 37  
LoadFromStream, 219  
named connections, defined, 28  
local expression engine. *See* FireDAC  
navigating  
scalar functions  
basic navigation, 154  
Local SQL, 11, 505  
Bof (beginning-of-file), 154  
DataSets property, 495  
DBNavigator, 86  
defined, 487  
DisableControls method, 158  
errors configuring, 504  
EnableControls method, 158  
FDLocalSQL component, 490, 504  
Eof (end-of-file), 154  
implementing, 505  
GetBookmark method, 160  
initial setup, 491  
GotoBookmark method, 160  
OnGetDataSet event handler, 96, 501  
MoveBy property, 155  
OnGetDataSet, errors and, 502

RecNo, 156  
parameterized queries, versus, 504  
refreshing, 87  
528 Delphi in Depth: FireDAC  
scanning, 157  
named, 119  
VCL controls for, 86  
positional, 119, 121  
navigating datasets, 161  
queries and stored procedures and, 121  
nested datasets, 395, *See also*  
Params property, 27  
FDMemTables, nested datasets  
Params.ArraySize property, 430  
creating at design time, 380–84  
parent-child relationships. *See* master-detail  
creating at runtime, 393  
relationships  
defined, 379  
performance  
example of, 380  
aggregates and, 272  
FieldDefs, runtime, 385  
datasets and, 218, 219, 220  
Fields, creating at design time, 381  
filtering versus searching, 232  
Fields, creating at runtime, 393  
Locate versus FindKey, 209  
limitations of, 395

master-detail performance, 258  
reasons for using, 380  
persistent connections, 28, 32, 36  
referring to, 394  
creating at runtime, 36  
nested transactions, 12  
defined, 15  
New Field dialog box, 265  
persistent fields. *See also* virtual fields  
numeric scalar functions, 419, *See*  
defined, 260  
FireDAC scalar functions  
persistent indexes, 167, 172–89  
aggregates and, 268

## —O—

persisting data, 10, 327  
database, in, 297  
ODAPI defined, 5  
defined, 298  
ODBC data source name (DSN), 41  
file formats, saving, 302  
ODBC drivers, 41  
files, in, 309  
ODBC Select Data Source wizard, 45  
formats, loading files, 302  
OnCalcFields event handlers, 280  
formatted storage, 304  
OnDataChange event handler, 150  
loading from files, 317, 316

one-to-many relationships. *See* master-merging when loading, 316  
detail views  
pretty print, storing, 304  
OnExecuteError event handler, 434  
SaveToFile, 309  
OnFilterRecord event handler, 232, 245,  
version, writing, 309  
246  
what to persist, 308  
OnReconcileError event handler, 163  
Post method, 161  
OnUpdateError event handler, 163  
predicates, 109, 218  
OnUpdateRecord event handler, 121  
defined, 8  
Options. *See* TransactionOptions  
private connections, 28, 39  
defined, 15

## —P—

projects  
FDAdvancedFilters, 234  
Paradox tables, accessing with FireDAC,  
FDAggregatesAndGroupState, 262, 275  
41  
FDArrayDML, 426  
parameters, 218  
FDBasicCache, 440  
advantages of, 110

FDCachedMasterDetail, 478, 482  
defining at design time, 114  
FDCachedUpdatesErrors, 454  
defining at runtime, 121  
FDCalcFields, 281, 284  
FireDAC Query Editor and, 118  
FDCloningCachedCursors, 376  
Index 529  
FDCopyClientDataSet, 354  
understanding, 231  
FDCopyDataSetQuery2Query, 359  
using, 222–27  
FDCopyingDataSets, 350, 362  
RecNo, 156  
FDDesigntimeNestedFields, 381  
record navigation, 86  
FDFieldDefsDesignTime, 336  
record state changes, detecting, 148–51,  
FDFilter, 168, 220, 221  
148–51  
FDIndexes, 174  
reFind conversion tool, 6  
FDLocalSQL, 499  
ResourceOptions, 70  
FDLookupField, 285  
ArrayDMLSize property, 69  
FDMacros, 412  
AutoConnect property, 69  
FDMacroSubstitution, 402

AutoReconnect property, 70  
FDMasterDetail, 251  
Backup property, 68  
FDMasterDetailClone, 372  
BackupExt property, 68  
FDNavigation, 149, 150, 154, 155, 159,  
BackupFolder property, 68  
160  
CmdExecMode property, 69, 140  
FDParadoxODBCDynamic, 45  
CmdExecTimeout property, 69  
FDPersistentConnection, 39  
command execution, 69  
FDPrivateConnection, 40  
command text processing, 67  
FDRuntimeNestedFieldDefs, 385  
connection resources, 70  
FDRuntimeNestedFields, 389  
DefaultParamType property, 67  
FDSaveAndLoad, 301  
DefaultStoreExt, 68  
FDSearch, 197  
DefaultStoreExt property, 301  
FDSQLPreprocessor, 399  
DefaultStoreFolder, 68  
FDTemporaryConnection, 26  
DefaultStoreFormat property, 68, 302,  
312

—Q—

DirectExecute property, 67  
EscapeExpand property, 67, 423  
queries. *See also* Structured Query  
KeepConnection property, 70  
Language  
MacroCreate property, 67, 398, 423  
asynchronous, 140  
MacroExpand property, 67, 398, 423  
defining parameters at design time, 114  
MaxCursors property, 70  
defining parameters at runtime and, 121  
ParamCreate property, 67  
monitoring, 144  
ParamExpand property, 67  
parameterized, 121, 218  
persistence mode, 70  
sql injection, 112  
Persistent property, 68  
QuotedStr function, 233  
PersistentFileName property, 68, 301  
ServerOutput property, 70  
**—R—**  
ServerOutputSize property, 70  
SilentMode property, 69  
RAD Server, 7  
StoreItems property, 68, 305, 308  
ranges  
StoreMegeData property, 316  
canceling, 231

StoreMegeMeta property, 316  
defined, 217  
StoreMetaData property, 68, 69  
filters, together with, 249  
StorePrettyPrint property, 69, 303  
index-based searches versus, 222  
StoreVersion property, 69  
indexes required, 222  
UnifyParams property, 67  
speed of, 222  
530 Delphi in Depth: FireDAC

—S—

macro substition, 406  
macro substitution, 502  
scanning datasets, 157  
macros, assigning at runtime, 406  
record by record, 199  
numeric scalar functions, 419  
search key buffer, 204  
showing processed SQL, 399  
searching  
special character processing, 423  
defined, 197  
SQL substitution, 401  
filtering versus searching, 217  
string substitution, 401  
FindKey versus FindNearest, 202  
string/character scalar functions, 417  
FindKey versus Locate, 208

system scalar functions, 422  
FireDAC datasets, 215  
SQL injection, 112, 219  
GotoKey method, 207  
SQLite  
  GotoNearest method, 207  
  Local SQL, support for, 487, 496  
  Locate method, 213, 208  
  statistics. *See* aggregates  
  Locate versus FindKey, 208  
  StatusBar, 150, 198  
  Lookup, 208  
  stored procedures  
  Lookup versus Locate, 213  
  defining parameters at design time, 114  
  ranges versus, 222  
  defining parameters at runtime and, 121  
  scanning records, 199  
  parameterized, 121  
  search key buffer, 204  
  sql injection, 112  
  SetKey method, 205  
  string scalar functions. *See* FireDAC scalar  
  using Variants, 215  
  functions  
  SetRange method, 222  
  string/character scalar functions, 417  
  SetRangeEnd method, 228  
  StringGrids, 211

SetRangeStart method, 228  
structure  
ShowHint, 87  
creating at runtime using Fields, 345  
simultaneous transactions, 12  
Structured Query Language  
sorting  
batch command processing. *See* Array  
InternalCalc fields and, 280  
DML  
special character processing  
creating flexible statements, 397  
FireDAC scalar functions, 423  
executing, 105  
speed  
heterogeneous queries, Local SQL and,  
datasets and, 220  
504  
SQL Command Preprocessor, 8, 423, *See*  
named parameters, 397, 423  
*also* FireDAC scalar functions  
parameterized queries, versus Local  
assigning macros, assigning at design  
SQL, 504  
time, 405  
positional parameters, 397, 423  
conditional substitution, 411  
substitution variables  
constant substitution, 408

path, using in, 300  
convert scalar function, 423  
**SUM**, 266  
date/time scalar functions, 421  
system scalar functions, 422, *See* FireDAC  
defined, 397  
scalar functions  
errors when using, 417  
escape sequences, 423  
FireDAC scalar functions, 423

**—T—**

identifier substitution, 409  
temporary connections, 16  
literals, escape sequences, 408  
defined, 15  
Index 531  
using Params, 26

**—U—**

temporary indexes, 167  
searching and, 201  
unique index, 166  
TFDGUIxWaitCursor component, 24  
Update SQL editor, 127  
TFDPhysIBDriverLink component, 24  
UpdateMode property, 80  
TFDStorageFormat enumeration, 302  
upWhereAll, 79  
TField. *See* fields  
upWhereChanged, 79

TranactionOptions  
upWhereKeyOnly, 79  
AutoCommit property, 77  
UpdateOptions, 75  
AutoStart property, 77  
AutoCommitUpdates property, 74, 451,  
AutoStop property, 77  
458, 482  
DisconnectAction property, 78  
AutoIncFields property, 73, 484  
disconnection action, 77  
automatic incrementing, 73  
EnableNested property, 78  
CheckReadOnly property, 74, 165  
isolation level, 76  
CheckRequired property, 74, 455  
Isolation property, 76  
CheckRequired property, 165  
nesting, 78  
CheckUpdatable property, 74  
Params property, 77  
CheckUpdatable property, 165  
StopOptions property, 77  
CheckUpdatedRecords, 125  
TransactionOptions, 78  
CountUpdatedRecords property, 74  
AutoCommit, 133  
EnableDelete property, 72  
automatic committing, 76

FastUpdates property, 74  
DBMS-specific parameters, 77  
FetchGeneratorsPoint property, 73  
ReadOnly property, 76  
general updating, 72  
update ability, 76  
GeneratorName property, 73, 484  
transactions, 140  
KeyFields property, 74  
committing, 134  
locking, 72  
dirty read isolation, 139  
LockMode property, 72  
explicit, 140  
LockPoint property, 72  
implicit, 133  
LockWait property, 72  
isolation, 138  
Mode property, 125  
nesting, 139  
posting changes, 75  
read committed isolation, 139  
property, 72, 75  
repeatable read isolation, 139  
ReadOnly property, 72  
rolling back, 134  
RefreshDelete property, 73  
starting, 134  
refreshing, 73

TStopWatch record, 152, 197

RefreshMode property, 73

TStringField. *See* fields

RequestLive property, 72

TThread

UpdateChangedFields property, 75

connection pooling and, 29

UpdateMode property, 75, 80

TThread class, 9

UpdateTableName property, 75

TVarRec record, 213, 225

TxOptions. *See* TransactionOptions, *See*

## —V—

TransactionOptions

VarArrayCreate function, 210

VarArrayOf function, 210

virtual fields, 296

532 Delphi in Depth: FireDAC

aggregates, 279

InternalCalc fields, 284

Calculated fields, 284

InternalCalc fields and FireDAC scalar

defined, 259

functions, 283

FieldOptions and, 296

Lookup fields, 292, 285