Getting Started With

pureQuery
Ideal for application developers and administrators

by:
VITOR RODRIGUES
ZEUS COURTOIS
HEATHER LAMB
CHRISTINA SHEETS
KATHRYN ZEIDENSTEIN

DB2 ON CAMPUS BOOK SERIES
GETTING STARTED WITH

pureQuery

A book for the community by the community

VITOR RODRIGUES, ZEUS COURTOIS, HEATHER LAMB, CHRISTINA SHEETS, KATHRYN ZEIDENSTEIN

FIRST EDITION
Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
3-2-12, Roppongi, Minato-ku, Tokyo 106-8711

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.
The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

References in this publication to IBM products or services do not imply that IBM intends to make them available in all countries in which IBM operates.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.
Trademarks

IBM, the IBM logo, DB2, z/OS, Optim, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at “Copyright and trademark information” at www.ibm.com/legal/copytrade.shtml.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.
Table of Contents

Preface ............................................................................................................................. 17
Who should read this book? ......................................................................................... 17
How is this book structured? ....................................................................................... 17
A book for the community ............................................................................................ 18
Conventions .................................................................................................................... 19
What’s next? ................................................................................................................... 19
About the Authors .......................................................................................................... 21
Contributors .................................................................................................................... 23
Acknowledgements ......................................................................................................... 24
PART I – OVERVIEW AND SETUP ............................................................................... 25
Chapter 1 - Introduction to pureQuery.......................................................................... 27
1.1 The big picture: What is pureQuery? ..................................................................... 27
1.1.1 What’s in a name? .......................................................................................... 28
1.1.2 What do we mean by platform? .................................................................... 29
1.2 pureQuery application programming interfaces .................................................. 30
1.3 pureQuery runtime ................................................................................................. 31
1.4 pureQuery application development environment .............................................. 33
1.5 pureQuery monitoring services .......................................................................... 35
1.6 Career path ............................................................................................................ 38
1.7 The pureQuery community .................................................................................. 38
1.7.1 Popular community Web sites ....................................................................... 39
1.8 Competitive products ............................................................................................ 40
1.9 Exercises ............................................................................................................... 42
1.10 Review Questions ................................................................................................. 42
Chapter 2 - Installing pureQuery (Optim Development Studio) ................................. 45
2.1 Installation: The big picture .................................................................................. 45
2.2 Installation prerequisites ....................................................................................... 46
2.3 Installing Optim Development Studio .................................................................. 47
2.3.1 Installing from the launchpad ....................................................................... 48
2.3.2 Applying Optim Development Studio Fixpack 1 ......................................... 54
2.4 Running Optim Development Studio .................................................................... 55
2.4.1 Adding pureQuery support to projects ....................................................... 55
2.5 Summary ................................................................................................................ 58
2.6 Exercise ............................................................................................................... 58
2.7 Review Questions ................................................................................................. 59
PART II – TOOLING ..................................................................................................... 61
Chapter 3 - Code generation and customized Java Editor Integration ....................... 65
3.1 pureQuery Tooling: The big picture ................................................................. 65
3.1.1 Generating pureQuery code ........................................................................ 66
3.2 Overview of the different types of wizards for code generation within the pureQuery tooling
### Chapter 4 - pureQuery tooling Views

#### 4.1 pureQuery tooling: The big picture

#### 4.2 SQL Outline view

- Gather the SQL statements issued by the application
- Gain insight into where SQL statements are issued within your application
- Pages of the SQL Outline view
- Enhanced impact analysis within database applications

#### 4.3 Designing and binding packages for static SQL development

- Designing database packages
- Binding database packages

#### 4.4 pureQuery Analysis view

#### 4.5 Enforce data privacy from modeling through development

#### 4.6 Summary

#### 4.7 Review Questions

### Chapter 5 - Developing for performance

#### 5.1 Developing for performance: The big picture

#### 5.2 Identify the time and frequency of execution and cost of a statement

- Gathering and viewing performance metrics
- Gathering and viewing explain data

#### 5.3 Optimizing your application for performance

- Optimize your applications by using Optim Query Tuner
- Comparing performance results after optimizing your application

#### 5.4 Summary

#### 5.5 Review Questions

### PART III – SECURITY, PERFORMANCE, AND INSIGHT

#### Chapter 6 - The Client Optimizer: pureQuery for Existing JDBC Applications

#### 6.1 Client Optimizer: The big picture

#### 6.2 Overview of how to use the Client Optimizer

#### 6.3 JDBC driver requirements for using Client Optimizer functionality

#### 6.4 pureQuery properties

#### 6.5 Capturing SQL statements

- pureQuery properties during capture
- In practice: Capture SQL statements in the Human Resources Manager application
Chapter 8 - Extended Insight

8.1 Extended Insight: the big picture
8.2 Common Problems in Multi-Tiered Monitoring
8.3 Monitoring pureQuery Clients
  8.3.1 More Monitored Layers
  8.3.2 Identifying pureQuery Applications Using Extended Insight
8.4 Extended Insight For A Variety of Application Types
  8.4.1 Monitoring Plain JDBC or CLI applications
  8.4.2 Monitoring pureQuery Applications associated with pureQueryXml
8.5 Summary
8.6 Review Questions

PART IV – API, RUNTIME AND SERVICES

Chapter 9 - Persistence Frameworks and Data Access: Context for pureQuery

9.1 Persistence Frameworks and Data Access: the big picture
9.1.1 pureQuery and Persistence Frameworks
9.2 pureQuery APIs in Context
9.3 pureQuery Data Access Overview
  9.3.1 pureQuery Beans as Transfer Objects
  9.3.2 Advanced pureQuery bean assembly using pureQuery Handlers
  9.3.3 Query over Collections with pureQuery
  9.3.4 XML entity mapping with pureQuery
  9.3.5 Overriding SQL Statements in Applications
9.4 Summary
9.5 Review Questions

Chapter 10 - Inline Programming Style

10.1 Inline programming style: The big picture
10.2 Instances of the com.ibm.pdq.runtime.Data interface
  10.2.1 In practice: Overview of the “In practice” sections in this chapter
  10.2.2 In practice: Create a Data object in the displayLogin() method
10.3 Executing SQL SELECT statements
  10.3.1 Choosing the return type
  10.3.2 Overriding the default cursor attributes
  10.3.3 In practice: Execute a simple SELECT statement that uses cursor attributes
    Complete the “Company Employees” screen
10.4 Executing SQL statement that have parameters
  10.4.1 In practice: Specify a scalar as a parameter to an SQL statement
    Complete the “Login” screen
  10.4.2 In practice: Specify a pureQuery bean as a parameter to an SQL statement
    Complete the “Employee Information” screen
10.5 Executing SQL statements that modify information in a database
  10.5.1 Executing an SQL statement one time by using the method: int update (String sql, Object... parameters)
10.5.2 Executing an SQL statement with generated keys one time by using the method: ROW update (String sql, Class<ROW> returnClass, String[] columnNames, Object... parameters) ..................................................... 198

10.5.3 Executing an SQL statement multiple times ............................................... 200

10.5.4 In practice: Execute an SQL statement that modifies the database -- Complete the "Change Employee Information" screen .......................................................... 201

10.6 Advanced features ............................................................................................. 202

10.7 Summary ............................................................................................................ 203

10.8 Review questions ............................................................................................... 203

10.9 Exercise ............................................................................................................. 205

Chapter 11 - Annotated-Method Programming Style ................................................ 207

11.1 Annotated-method Programming Style: The big picture ................................ 207

11.2 Annotated-method interfaces .............................................................................. 210

11.2.1 In practice: Overview of the "In practice" sections in this chapter ................. 210

11.2.2 In practice: Observe the HumanResourcesData interface ............................ 211

11.2.3 In practice: Create an instance of an interface of annotated methods in the displayLogin() method .................................................................................... 211

11.3 Generating an implementation class for an interface of annotated methods ... 212

11.3.1 Specifying options to the pureQuery Generator .......................................... 212

11.3.2 Generating an implementation class in Optim Development Studio ............ 215

11.4 Annotated methods ............................................................................................ 215

11.4.1 Declaring the return type ............................................................................ 217

11.4.2 In practice: Execute a simple SELECT statement that uses cursor attributes -- Complete the "Company Employees" screen .......................................................... 219

11.5 Executing SQL statements that have parameters ............................................. 221

11.5.1 In practice: Specify a scalar as a parameter to an SQL statement -- Complete the "Login" screen .................................................................................................. 223

11.5.2 In practice: Specify a pureQuery bean as a parameter to an SQL statement -- Complete the "Employee Information" screen ....................................................... 225

11.5.3 Special case scenarios for parameter types ............................................... 227

11.6 Executing SQL statements that modify information in a database .................... 228

11.6.1 Retrieving generated keys ........................................................................... 228

11.6.2 In practice: Execute an SQL statement that modifies the database -- Complete the "Change Employee Information" screen .......................................................... 229

11.7 Tooling Support for Annotated Methods and Generated Code ......................... 231

11.7.1 Generating XML file to override the SQL statements in interfaces ............ 231

11.7.2 EMFT JET template customization for code generation ............................ 232

11.8 Using static SQL with annotated-method style applications .............................. 233

11.8.1 Binding the SQL statements in the interfaces .......................................... 234

11.8.2 Running an annotated-method style application in STATIC execution mode ............................................................................................................................... 236

11.9 Advanced features ............................................................................................. 238

11.10 Summary .......................................................................................................... 238

11.11 Review questions ............................................................................................. 239
11.12 Exercise ........................................................................................................... 240

Chapter 12 - Stored Procedures ................................................................. 243
  12.1 Stored procedures: The big picture ..................................................... 243
  12.2 The StoredProcedureResult interface .................................................. 244
    12.2.1 Getting the updated values of OUT and INOUT parameters ........ 244
    12.2.2 Getting the query results of the stored procedure ..................... 245
    12.2.3 Closing the StoredProcedureResult object .............................. 247
  12.3 Executing SQL stored procedure calls in the inline style .................... 247
    12.3.1 In practice: Use the inline style to execute a stored procedure -- Complete the “Increase Employee Bonuses” screen ................................................................. 249
  12.4 Executing SQL stored procedure calls in the annotated-method style ...... 251
    12.4.1 In practice: Use the annotated-method style to execute a stored procedure -- Complete the “Increase Employee Bonuses” screen ............................................. 252
  12.5 Summary ............................................................................................... 253
  12.6 Questions ............................................................................................... 254
  12.7 Exercise ................................................................................................. 255

Chapter 13 - Handlers ....................................................................................... 261
  13.1 Handlers: The big picture ........................................................................ 261
  13.2 Specifying handlers in the inline style ................................................... 263
    13.2.1 In practice: Specify a handler using the inline style -- Complete the “Company Employees” screen ................................................................. 264
  13.3 Specifying handlers in the annotated method style ............................. 266
    13.3.1 Specifying handlers by using the @Handler annotation ................ 266
    13.3.2 Specifying handlers as method parameters .................................. 267
  13.4 Writing handler implementation classes ............................................. 268
    13.4.1 Writing a ResultHandler<RES> implementation ....................... 269
    13.4.2 Writing a RowHandler<ROW> implementation ......................... 270
    13.4.3 Writing a CallHandlerWithParameters<CAL> implementation ...... 272
    13.4.4 How to write a ParameterHandler implementation .................... 274
    13.4.5 In practice: Write and specify a ParameterHandler implementation ................................. 274
  13.5 Summary ............................................................................................... 275
  13.6 Review questions .................................................................................... 276
  13.7 Exercise ................................................................................................. 277

PART V - BEST PRACTICES AND INTEGRATION ............................................. 279

Chapter 14 - Best Practices ............................................................................ 281
  14.1 Best Practices: The big picture .............................................................. 281
  14.2 Choose your style: inline vs annotated method style .......................... 281
  14.3 Perform Joins with Table collation ...................................................... 282
  14.4 Working with Exception categorization ............................................... 283
  14.5 Customizing the code generation templates ........................................ 283
  14.6 Recapture the SQL in your application ............................................... 285
  14.7 Custom handlers and annotated method style ..................................... 286
Preface

Keeping your skills current in today's world is becoming increasingly challenging. There are too many new technologies being developed, and little time to learn them all. The DB2® on Campus Book Series has been developed to minimize the time and effort required to learn many of these new technologies. The content of this book is based in the features available in the version 2.2.0.1 of IBM® Optim™ Development Studio and IBM Optim pureQuery Runtime. At the time this book was ready for publishing, new features of pureQuery became generally available. For more information about these new features, refer to the “What's New?” page in this link:


Who should read this book?

This book is intended for anyone who needs to learn about pureQuery, IBM's high-performance data access platform. pureQuery delivers benefits in the entire spectrum of the application life cycle, including the design, development, management and governance stages. This book should be read by all individuals that participate in the stages of the application life cycle, especially application developers and database administrators.

Before reading this book, it is recommended that you have some familiarity or knowledge regarding topics addressed by the following books, also part of the DB2 on Campus Book Series:

- Getting Started with Eclipse
- Getting Started with IBM Data Studio for DB2
- Getting Started with Java
- Getting Started with DB2 Express-C

How is this book structured?

This book is structured in five major parts:

- Part I provides an overview of pureQuery and contextualizes it in the enterprise database application spectrum. Chapter 1 introduces pureQuery and describes its
main components: API, runtime, tooling and monitoring services. Chapter 2 covers installation of IBM’s pureQuery.

- Part II covers the pureQuery tooling, available as part of IBM’s Optim Development Studio product.

- Part III takes a step back and describes how pureQuery supports the broader goals of security, performance and insight for new and existing Java™ and JDBC applications. It covers topics such as client optimization for new and existing applications and monitoring tools which give the application centric DBA insight into layers of the runtime stack they have never been able to monitor before.

- Part IV digs into the internals of pureQuery. This part starts with Chapter 9, which covers in detail the existing persistence and Object-relational mapping frameworks and positions pureQuery among them. The next chapters describe the API, runtime and monitoring services of pureQuery.

- Part V covers pureQuery best practices and the integration of pureQuery with existing technologies and frameworks.

Exercises are provided in selected chapters. Most chapters contain review questions to help you learn the material; answers to those questions are included in Appendix A.

A sample application is also provided with this book and described in Chapter 4. The source code for the application, as well as code samples developed throughout the book, can be found in the provided zip file `GettingStartedWithPureQuery.zip` accompanying this book.

**A book for the community**

This book was created for the community; a community consisting of university professors, students, and professionals (including IBM employees). The online version of this book is released to the community at no-charge. If you would like to provide feedback, contribute new material, improve existing material, or help with translating this book to another language, please send an email of your planned contribution to `db2univ@ca.ibm.com` with the subject “pureQuery book feedback.”
Conventions

Many examples of commands, SQL statements, and code are included throughout the book. Specific keywords are written in uppercase bold. For example: A NULL value represents an unknown state. Commands are shown in lowercase bold. For example: The dir command lists all files and subdirectories on Windows®. SQL statements are shown in upper case bold. For example: Use the SELECT statement to retrieve information from a table.

Object names used in our examples are shown in bold italics. For example: The flights table has five columns.

Italics are also used for variable names in the syntax of a command or statement. If the variable name has more than one word, it is joined with an underscore. For example: CREATE TABLE table_name

What’s next?

In addition to the books in the section “Who should read this book?“, we recommend you to review the following books in the DB2 on Campus book series for more details about related topics:

- Getting Started with DB2 Application Development
- Getting Started with Infosphere™ Data Architect
- Getting Started with SOA

The following figure shows all the different eBooks in the DB2 on Campus book series available for free at ibm.com/db2/books
The DB2 on Campus book series
About the Authors

Vitor Rodrigues works at Google as a Software Engineer designing and developing tools for the web. Prior to that, he spent several years as a Software Engineer at the IBM Silicon Valley Lab working on Optim™ Development Studio and Data Studio products. Previously held positions at IBM include Technical Enablement and Quality Assurance roles in the Data Studio and DB2 pureXML organizations. Vitor graduated in Computer and Software Engineering from University of Minho, Portugal. He is an IBM Certified Solution Developer for XML and Related Technologies and an IBM Certified Database Administrator - DB2 9 DBA for Linux®, UNIX®, and Windows. Vitor has co-authored several articles and tutorials for developerWorks®.

Zeus Courtois is a Software Engineer at the IBM Silicon Valley Lab working on Optim Development Studio for the the pureQuery tooling team. Previously, as an intern within IBM, held positions in the Technical Enablement Team for DB2 pureXML and Data Studio. Zeus has a bachelors degree in Computer Science and a Masters of Science in Information Systems from Texas A&M Int'l University. Zeus has co-authored several articles for the pureQuery tooling in IBM developerWorks.

Heather Lamb is a software engineer at IBM in Herndon, Virginia. She worked for three years as a developer for the IBM Optim pureQuery Runtime and Optim Performance Manager Extended Insight products. Currently she is a technical consultant with IBM’s Global Business Services. Her subject of expertise is software development. She previously worked as a software developer on IBM’s JDBC and SQLJ Driver, and spent three years as a communications officer in the United States Air Force. She holds a master’s degree in Computer Science from Stanford University.

Christina Sheets works as a software engineer for IBM. She is one of the developers of the Optim pureQuery Runtime software. She attended the University of Florida, where she received a master's degree in Computer Engineering, as well as a bachelor's degree in Computer Science and Mathematics. Christina lives with her husband in California. In her free time, she helps with various ministries at her church.
Kathryn Zeideinstein is a member of the Optim Solutions technical enablement team and has responsibility for community building and communications with the technical community. She has many years of experience with IBM starting out as an Information Developer for DB2 for z/OS®, managing the SQL Standards team, managing editor for the Information Management zone on developerWorks and as product manager and marketing manager for several Information Management products. She has authored or co-authored numerous article on developerWorks and in other publications, including topics ranging from business intelligence to content analytics. She led the team the produced the DB2 On Campus book Getting Started with Data Studio for DB2. Kathy holds a master’s degree in Professional Writing from Illinois State University.
The following people edited, reviewed, provided content, and contributed significantly to this book.

<table>
<thead>
<tr>
<th>Contributor</th>
<th>Company/University</th>
<th>Position/Occupation</th>
<th>Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vladimir Bacvanksi</td>
<td>SciSpike Inferdata</td>
<td>Founder Owner and Vice President</td>
<td>External Review.</td>
</tr>
<tr>
<td>Kevin Campbell</td>
<td>Univar</td>
<td>Manager, Enterprise Architecture</td>
<td>External Review.</td>
</tr>
<tr>
<td>Daniel Galvin</td>
<td>Galvin Consulting, Inc.</td>
<td>DB2 Consultant</td>
<td>External Review.</td>
</tr>
<tr>
<td>Petter Graff</td>
<td>Inferdata</td>
<td>Vice President and Partner</td>
<td>External Review.</td>
</tr>
<tr>
<td>Raul F. Chong</td>
<td>IBM</td>
<td>DB2 On Campus Program Manager</td>
<td>Review and guidance.</td>
</tr>
<tr>
<td>Christopher Farrar</td>
<td>IBM, Silicon Valley Laboratory</td>
<td>Software Architect, pureQuery runtime</td>
<td>Review and guidance.</td>
</tr>
<tr>
<td>Nazila Malekpour</td>
<td>IBM, Silicon Valley Laboratory</td>
<td>Software Developer, Optim Development Studio</td>
<td>Review.</td>
</tr>
<tr>
<td>Nisha Nair</td>
<td>IBM, Silicon Valley Laboratory</td>
<td>Software Developer, Optim Development Studio</td>
<td>Review.</td>
</tr>
<tr>
<td>Lakshman Sakaray</td>
<td>IBM, Lenexa</td>
<td>Team Lead, Optim Development Studio</td>
<td>Review.</td>
</tr>
<tr>
<td>Mabel Thong</td>
<td>IBM, Silicon Valley Laboratory</td>
<td>QA Engineer, Optim Development Studio</td>
<td>Review.</td>
</tr>
</tbody>
</table>
Acknowledgements

We greatly thank the following individuals for their assistance in developing materials referenced in this book:

Becky Nin for her guidance on the pureQuery tooling chapters.

Anshul Dawra for his guidance on pureQuery runtime for the tooling team.

Sonali Surange, for her guidance on the pureQuery tooling chapters and her several IBM developerWorks articles that were a guide to the tooling chapters in this book.

Fay Wand et al, for the IBM developerWorks article "Integrating JPA and pureQuery: Leveraging DB2 static execution for the Java Persistence API".

Rafael Coss for his assistance in finding external reviewers for this book.

Natasha Tolub for designing the cover of this book.

Susan Visser for assistance with publishing this book.
PART I – OVERVIEW AND SETUP
Chapter 1 - Introduction to pureQuery

pureQuery software is a key differentiator of IBM Integrated Data Management solutions. It provides a high-performance data access platform that makes it easier to develop, optimize, secure, and manage data access.

In this chapter you will learn:

- The big picture: What is pureQuery?
- How pureQuery can optimize database access, improve developer productivity, and help DBAs gain insight into the database interactions of applications
- What products deliver pureQuery technology
- How knowledge of pureQuery can help your career
- Web sites where you can get more information and join a community

1.1 The big picture: What is pureQuery?

pureQuery is a database access platform that includes features of interest to both developers and DBAs. Indeed, pureQuery can help developers and DBAs work better and more effectively together, leading to more efficient database access, reduced time to solve problems, and improved ability to manage and monitor database applications.

In this section, we'll define pureQuery as a platform and explain what we mean by that. We'll also go over the components of that platform and then discuss how aspects of the platform get delivered to the public as software products. Note that although pureQuery capabilities are strongest in support of Java application environments, there are benefits from pureQuery to other environments such as .NET, and those benefits are planned to grow over time.
1.1.1 What's in a name?

**pureQuery** is the name IBM has chosen to represent a data access platform that can help optimize and secure data access. It was first released publicly in October 2007 as part of what was then known as the “Data Studio” family (now known as the “Optim™” product family). (More on the history of pureQuery can be found on Wikipedia [1]).

Because you may be familiar with the term pureXML® from what you’ve read in *Getting Started with DB2 Express-C*, it’s interesting to get a little background on how this data access platform ended up with the name pureQuery. Stephen Brodsky, an IBM Distinguished Engineer in the IBM Optim organization, in [one of his blog entries](#), says it best:

> So how did this platform get the name pureQuery? It didn't start out that way (as is often the case with new products or technologies). We had two internal code names for the project, Data Zero (because it's the data access layer for [Project Zero](#)) and JLinQ, because it compares very favorably with Microsoft®’s LINQ initiative (all the power of database access without having to change the C# and Visual Basic languages – we achieved all the major features of LINQ but stayed pure standard Java™). pureQuery embraces the full database access without watering down the SQL or database capabilities. It avoids least common denominator approaches that dilute the ability to access what you really need in a database and integrates all aspects of the life cycle in both runtime and tooling, “Pure” means no compromises and first class Java-SQL-database integration.

> As the project moved closer to release, it was time to choose a name. Because the project represented application plus database integration at what we feel is the best, truest form, without all the mess that sometimes bogs software down, early users called it “pure data access” or “pure data query (PDQ).” The pure concept, and the PDQ acronym, really resonated with the marketing team (the same team that named pureXML), and the pureQuery name was born. [2]

---

It’s important to note here that since Steve wrote that blog entry, a subset of pureQuery benefits have been extended beyond Java data access, including to .NET, and by the time this book is published, CLI. The focus of this Getting Started book is on the support for Java environments.
1.1.2 What do we mean by *platform*?

IBM calls pureQuery a “platform” because it seems to be the best word to express the vision of pureQuery as a technology that spans the data management lifecycle. In other words, there are aspects of pureQuery that are embedded in design, development, test, deployment, monitoring, and optimization. (For more information about how IBM defines the data management lifecycle, see the e-book *Getting Started with Data Studio for DB2*.)

pureQuery consists of the facets shown in *Figure 1.1*, which can be used as needed for any particular project or task and which are delivered in specific products related to those tasks.

![Figure 1.1 - Facets of pureQuery data access platform](image)

- **Application programming interfaces** (APIs), built for ease of use and for simplifying the use of best practices for enhanced database performance when using Java.
- A *runtime*, which provides optimized and secure database access
- An Eclipse-based *integrated database development environment (IDE)* for enhancing development productivity, improving data access performance during development, and improving collaboration between developers and DBAs
- **Monitoring services**, to provide developers and DBAs with previously unknown insights about performance of Java and CLI database applications.
Let’s look at each of these facets in detail.

### 1.2 pureQuery application programming interfaces

The pureQuery APIs are built for ease of use, enhanced productivity, and to encourage use of best practices for Java development when accessing DB2, Informix® Dynamic Servers or Oracle® databases. These APIs are available at development time to optimize execution using Optim pureQuery Runtime. Optim Development Studio provides integrated tooling to support quick and easy application development using the pureQuery APIs. A full pureQuery runtime license is required for the resulting applications in production deployments.

When compared with JDBC, the pureQuery APIs allow developers to build applications with far less handwritten code yet still with greater control over database access than provided by many object-relational frameworks.

pureQuery directly utilizes SQL for relational database access, in-memory collections, and XML, so developers do not need to learn a new query language.

The pureQuery API provides both inline SQL and Java annotated method styles of programming. Both styles are designed to be significantly easier to use than JDBC. Each style has its advantages depending on the use case. For more details on choosing a style, see Chapter 15.

The inline style is similar to JDBC in that the SQL is defined within the Java code as a string object, however unlike JDBC, pureQuery supports an editing environment that is SQL aware, and pureQuery’s content assist makes it easy to add accurate, error-free SQL. Inline style also provides a variety of methods which implement common database access patterns to provide value over JDBC. The inline style is described further in Chapter 5.

In the annotated method style, described in Chapter 6, the SQL is provided as separate annotations and is invoked via Java methods, as part of a user-defined interface. The annotated method style (also known as method style) has some advantages. It shields the application from specific interface implementations, which means you can switch to another persistence technology if needed without changing your program. Also, with method style, your application can use static SQL when accessing DB2 databases once it has been bound, with a simple switch of a runtime flag, which means developers can develop and
test their applications using dynamic SQL, and with no additional changes, that same
application can be bound and take advantage of the performance, security, and stability of
static SQL execution (described in Appendix D).

**Note:**
If you are not familiar with static SQL and the benefits of using it, see Appendix D or the
article entitled *No excuses database programming for Java* at

The pureQuery APIs are designed with Web 2.0 in mind -- custom result handlers map
results to plain old Java objects (POJOs), XML, or JavaScript Object Notation (JSON).

In addition to improved ease-of-use, performance is easier to build into the application
using pureQuery. For example, it’s easy to specify retrieval of only the first row of results,
to reduce network and processing time for time-critical user interfaces. Other options for
reducing network cost include the use of batching updates across tables, known as
*heterogeneous batching*, and by the ability to query and join across in memory Java
collections and the database.

### 1.3 pureQuery runtime

The pureQuery runtime is the engine that provides improved performance, security, and
manageability and is delivered in the product *Optim pureQuery Runtime*. You deploy the
runtime on the application server (or wherever the application runs). The pureQuery
runtime provides flexible access from a wide variety of frameworks and data access
mechanisms, as shown in *Figure 1.2*. 
For **Java applications**, pureQuery runtime enhancements are independent of the deployment components and environment. This means pureQuery runtime also enhances applications running within servers such as WebSphere® and other popular Web application servers. pureQuery can accelerate new Java applications that use the pureQuery API, or existing Java applications, even those applications in which the SQL is generated by a framework, such as Hibernate or OpenJPA. For more information on pureQuery integration with JPA, please refer to Chapter 16 and our Resources section.

For both pureQuery applications and for those that are written using a framework, the pureQuery runtime captures the application SQL and related metadata for use by developers or DBAs to share, review, analyze, optimize, revise, and restrict SQL execution. There are a few extra steps involved in gathering this information in a process called **client optimization**, which is described further in Chapter 9.

If the target database is DB2, the captured SQL can be bound into packages for static execution to improve performance, enhance security, improve manageability, and reduce costs.
Client optimization enhances OpenJPA applications, just like JDBC applications, in a wide variety of environments. However Open JPA applications on WebSphere Application Server V7 or later releases that are accessing DB2 can take advantage of enhanced integrations with pureQuery runtime to quickly get those applications up and running using static SQL and heterogeneous batching capability when accessing DB2 data, without requiring client optimization steps. You can find out more about this capability in Chapter 16.

For .NET applications that access DB2, use the Optim pureQuery Runtime to enable a command-line-based capture and bind of dynamically running SQL statements to improve performance, security, and manageability.

**Note:**
This e-book is focused on Java, but if you’re interested in the capabilities that exist for .NET, see the developerWorks tutorial entitled Optimizing your existing .NET applications using pureQuery here:


### 1.4 pureQuery application development environment

The pureQuery development environment is currently focused on Java application development and optimization and is delivered in Optim Development Studio, an integrated development environment based on open source Eclipse. This is the product you will be using to do the exercises in this book. Optim Development Studio supports pureQuery development against the DB2 family and Informix® Dynamic Server. A screenshot is shown in Figure 1.3.
Figure 1.3 - Optim Development Studio provides enhanced tools for Java database access development and tuning

Optim Development Studio includes the following key productivity features:

- As shown in Figure 1.3, you can visualize linkages among the Java source code, the SQL (generated or not), and the database objects themselves to speed up problem isolation and dependency analysis. An important benefit of this capability is helping DBAs speak to developers in a language developers understand. This is important in real world environments in which developers are often insulated from the SQL that gets issued to the database when object-relational mapping tools are used.
As shown in Figure 1.3, analyze potential SQL performance “hotspots” quickly and easily directly from your desktop.

Collaborate with peers to review and optimize application SQL. You can launch Optim Query Tuner, a separately available product, directly from the development environment to get expert tuning advice for DB2.

Show SQL errors in the Java editor automatically without having to execute the application as shown in the partial screenshot below in Figure 1.4.

![Figure 1.4 - Integrated SQL error and semantic checking in Optim Development Studio Java editor](image)

Optim Development Studio includes a copy of the pureQuery runtime for use on your workstation. Use Optim Development Studio on its own to build better Java applications more quickly, or combine it seamlessly with other Eclipse-based tools you may need, such as Data Studio, Optim Database Administrator, InfoSphere™ Data Architect, or Rational® Application Developer for WebSphere Software.

1.5 pureQuery monitoring services

pureQuery also delivers client-side monitoring capabilities. This client side monitoring capability provides insights that prove useful for developers and DBAs. For developers,
monitoring services in pureQuery enable the display of performance metrics in Optim Development Studio, as you saw in the screenshot shown in Figure 1.3.

For DBAs, there are even more statistics being gathered and displayed in Optim Performance Monitor using the Extended Insight Feature that help them understand the activity in the software stack across the database client, the application server, and the network.

Before pureQuery, insight into the Java driver stack and the ability to see what line of a Java application was executing SQL was unavailable for DBAs. Traditional database monitoring tools listen on the server side only. They watch and report events that happen inside the database server, e.g. buffer pool hits, cache hits, lock contention, etc. But they have no insight into how the details of the Java application itself affected the database time, the amount of time spent in the network, the connection pool in the application server, and so on.

Why is that a problem? Because, without a unified view across the stack, it is hard for a DBA to tell where the problem resides. The application says it issued the query so it must be the database server’s problem. So the DBA is often the one trying to figure out where the problem is and/or working with the network administrator, the systems administrator, and the application developer comparing notes and traces to isolate the issue.

pureQuery, as provided in the Optim Performance Manager Extended Edition, Extended Insight Feature, provides correlation information that enables tracking of performance information for any particular database transaction across the entire software stack. This enables DBAs to quickly assess the problem source as well as provide a more realistic assessment of response time. With Extended Insight (EI), the Java application clients are now sending additional data to your performance monitor, with information like time spent in application, in connection pooling, in the driver, or in the network, as shown in Figure 1.5.
This capability provides a fundamental shift from traditional monitoring. Now DBAs have an end-to-end database monitoring solution for a complete view of performance across the database client, the application server, the database server, and the network between them. And that information can be displayed in graphs and charts for easy identification of misbehaving transactions.

DBAs can now:

- Monitor performance indicators that better reflect end user experience and proactively detect negative trends in:
  - Response times for database APIs
  - Network congestion between the application and the database, or
  - Other factors key to sustaining service level agreements

- See where workloads, transactions, and SQL requests spend their time across the database client, the application server, the database server, and the network between them.
• Isolate problems to the correct layer in the stack, including visibility into database related performance indicators in WebSphere application servers

With these insights, problems that took days to isolate before can now be found in minutes. The problems can be solved by one person, rather than having to route a problem from DBA to net admin, to AIX and WAS admin, to developer, just to try to gather enough information and skills to investigate every layer of the stack.

1.6 Career path

Because the vision of pureQuery is to support a wide variety of heterogeneous database environments, learning the tools and APIs associated with pureQuery can help you be more productive as a developer or DBA in an environment where support of mixed databases is required.

Application developers who use the pureQuery API can build applications that run both on DB2 (and which can easily be converted to static SQL for optimized applications) and on Informix Dynamic Server and Oracle databases. Additional databases are planned to be added to this list over time. And the process to optimize existing Java applications using client optimization is supported across all those databases.

In addition, the emphasis on performance in the development cycle with pureQuery will help application developers grow database and query tuning skills that can help them grow to a DBA or specialized query tuning position.

1.7 The pureQuery community

Because pureQuery is a platform that can span a variety of data servers, programming languages, and IT roles, pureQuery will benefit a wide variety of communities – any of the DB2 communities, Oracle communities, or Java development communities for example. There is also a developerWorks discussion forum on Optim Development Studio and pureQuery that many people in the community and in the software labs monitor and respond to at http://www.ibm.com/developerworks/forums/forum.jspa?forumID=1086&categoryID=19
1.7.1 Popular community Web sites


A great resource is the Integrated Data Management community space (Figure 1.6) that aggregates links to downloads, forums, blogs (such as the Integrated Data Management experts blog), technical articles, videos, and more, in a single location at [http://www.ibm.com/developerworks/spaces/optim](http://www.ibm.com/developerworks/spaces/optim).

![Integrated Data Management (Optim and Data Studio)](image_url)

**Figure 1.6 - Integrated Data Management community space**

When you go to the community space, you will see a variety of portlets that contain links to forums, videos, and more. There is a tab on the space that includes links to all downloads, including Optim Development Studio, which contains the pureQuery runtime for development use on your workstation. And there is a tab with links to many existing articles and tutorials on using pureQuery capabilities as delivered in the Optim product family.
Finally, the pureQuery platform page on ibm.com provides a resource to keep you current on pureQuery products, benefits, performance, and FAQs.

http://www.ibm.com/software/data/optim/purequery-platform/

1.8 Competitive products

Because pureQuery is a platform, it serves to complement some existing technologies while competing in other areas on a product level. If we focus on Java data access rather than products such as performance monitors, pureQuery is complementary to Java frameworks.

Figure 1.7 positions Java data access technologies on a continuum from those on the left that are very data-centric, that is, for developers who want and need more control over the data access code, to those on the right, that provide ease-of-use features for object-oriented programmers but which provide generally less control over what SQL is generated. The right side of the continuum often provides slower data access performance.

Figure 1.7 - pureQuery in the context of other Java development environments
Because coding in JDBC is so tedious, it only makes sense that Java developers would turn to frameworks such as Spring, iBatis (now called myBatis), and JPA to help with their object-relational mapping, among other things. pureQuery does provide productivity boosts that many are looking for in a Java framework, but it does not provide the full persistence or container management associated with Hibernate or JPA. As a framework alternative, pureQuery attempts to provide a balance point between enhanced productivity and full SQL control.

However, pureQuery is different from other frameworks in that it can complement any existing framework. Developers can continue to use their framework of choice and add pureQuery to get the added benefits of performance enhancement, improved security, and improved manageability. Integrations with other frameworks are described in Part V.

Of course, if you are writing new applications and don’t require full container management, then pureQuery is an excellent choice in terms of performance and productivity when your data is stored in DB2, Informix Dynamic Server, or Oracle databases.

As shown in the figure above, pureQuery attempts to provide the best of both worlds:

- It gives the developer the ability to fully control the SQL where needed
- It helps the developer create good SQL without having to learn a new query language
- It offers a very simple API
- It provides basic object-relational mapping at development time, with no need to map complex relationships between entities and objects
- It can provide performance, security, and manageability benefits to those who continue to use full persistence frameworks, using client optimization.

As mentioned earlier, pureQuery can use client optimization to take any existing Java application, even if coded in a framework, and “uplift” it so that those applications can obtain many of the same benefits they would get when coding to the pureQuery APIs, notably static SQL and enhanced visualization for improved problem determination. The client optimization technology can help regardless of whether your application is a complicated Hibernate application or an out-of-the-box Java/JDBC program for which you may not even have the source code.
1.9 Exercises

We will use this first exercise to ensure your environment is ready to install Optim Development Studio and to ensure you have all the required software downloaded for future exercises. Follow these instructions:

1. Ensure your computer is using one of the supported Windows or Linux operating systems as described in this technote: http://www-01.ibm.com/support/docview.wss?rs=4014&uid=swg27015710

An IBM or Sun® Java Runtime Environment at version 1.5.0 or higher must be installed on your system and for development you need the **IBM Data Server Driver for JDBC and SQLJ V3.57**. For applications using the db2jcc4.jar file: IBM Data Server Driver for JDBC and SQLJ V4.7 or later. (These prerequisites are all included with Optim Development Studio for development purposes).

2. Download **Optim Development Studio**. If you do not already own it, you can download a 30-day trial from: http://www.ibm.com/developerworks/downloads/im/datastudiodev/. If you are a university student, your university may have an academic license to use it – if not, use the trial version above. You will learn how to install the product in Chapter 2.

3. If you don’t already have DB2, you can download **DB2 Express-C**. We will use the free version of DB2, DB2 Express-C, for the examples used in this book. To download the latest version of DB2 Express-C, click this link and choose the appropriate file to download for the operating system you are using.

4. Explore the Optim Development Studio and pureQuery information topics, which are located in the Integrated Data Management Information Center at this location.

1.10 Review Questions

1. What are the four facets that comprise the pureQuery platform?

2. True or false: pureQuery client monitoring capabilities are only of value to DBAs? Why or why not?

3. True or false: pureQuery competes directly with Java frameworks such as Hibernate.
4. For which development environments can pureQuery provide benefit?

5. What is the name of the capability that enables existing Java, .Net, or CLI applications to take advantage of pureQuery static SQL execution capabilities?

6. Which of the following products delivers the pureQuery development tooling?
   A. Data Studio
   B. Optim Development Studio
   C. Rational Application Developer for WebSphere Software
   D. Optim Performance Manager Extended Edition

7. Which of the following products delivers pureQuery client monitoring capabilities for use by DBAs in monitoring database and application interactions?
   A. Data Studio
   B. Optim pureQuery Runtime for z/OS
   C. Optim Performance Manager Extended Edition
   D. DB2 Application Development Client

8. Which of the following capabilities is **not** included in Optim Development Studio:
   A. Visualization of SQL performance “hot spots”
   B. Correlate SQL statements with originating line of Java source code
   C. .NET visual development
   D. Generate data access layer for create, read, update, and delete operations

9. Which product is required for a production application to take advantage of pureQuery benefits, including client optimization, heterogeneous batching and other APIs, or static execution?
   A. Optim Development Studio
   B. Optim pureQuery Runtime
   C. DB2 Application Development Client
   D. WebSphere Application Server
10. Which of the following database servers cannot be accessed using pureQuery at the time of publication?

A. Oracle

B. DB2 for zOS

C. Informix Dynamic Server

D. Microsoft SQL Server
Chapter 2 - Installing pureQuery (Optim Development Studio)

In this chapter you will learn how to install the pureQuery development capabilities in your system, including the runtime, integrated development environment and the API. This whole package is included for development use in Optim Development Studio, which can be downloaded as a no-charge, 30-day trial package, and this is what you will learn how to install in this chapter.

If you want to move a pureQuery application into production or do any testing on a nondevelopment machine, you will also need to obtain the separate package called IBM Optim pureQuery Runtime. For information on how to obtain the pureQuery runtime standalone installation package, please contact your Academic Initiative representative or your IBM Sales Representative.

2.1 Installation: The big picture

Optim Development Studio can be installed either using the Launchpad GUI, which launches IBM Installation Manager, or silently, which means you create a response file of your chosen installation options, and then run that response file. Silent install is mainly useful for larger installations in which installation must be pushed out to many machines.

This chapter focuses on the Launchpad installation. It assumes you do not already have IBM Installation Manager installed. This means that an installation of Optim Development Studio will also install IBM Installation Manager, which can be used later to install other IBM products. Figure 2.1 shows the Optim Development Studio Launchpad welcome screen:
2.2  Installation prerequisites

With respect to operating system version and level requirements, Optim Development Studio is available on Linux (32 bit) and Windows 2003, 2008, XP, and Vista. As for hardware requirements, installation of Optim Development Studio requires a minimum of 1GB RAM memory and 750MB of disk space.

The installation process for Optim Development Studio requires one of the following web browsers: Microsoft Internet Explorer 6 SP1, Mozilla 1.6 or 1.7, and Firefox 2.0 or higher.

The data sources supported by Optim Development Studio 2.2.0.1 are:

- DB2 for Linux, Unix and Windows versions 8.2, 9.1, 9.5 and 9.7
- DB2 for iSeries® V5R3 and V5R4
- DB2 for z/OS versions 8.1 and 9.1
- IBM Informix Dynamic Server versions 10.0, 11.1 and 11.5
- Oracle Database versions 10g, 10g R2 and 11g

For the latest information on Optim Development Studio system requirements, review the product’s web page: [http://www.ibm.com/software/data/optim/development-studio/](http://www.ibm.com/software/data/optim/development-studio/)

### 2.3 Installing Optim Development Studio

A free 30-day trial of ODS can be obtained from the [product’s download page](http://www.ibm.com/software/data/optim/development-studio/). You can download three variants of the package:

- **Web install using IBM Installation Manager (IM)**: It includes IBM Installation Manager install package and ODS product information that lets you install ODS from within IM.
- **Web install if you already have IBM Installation Manager installed**: this package provides all the information required to install ODS from an Installation Manager installation already present in your system.
- **Local install using Download Director or HTTP**: this package contains all the binaries that make ODS and should be used when you want to install the product from your local drive instead of using a web repository.

These installation packages are highlighted in figure *Figure 2.2*.

<table>
<thead>
<tr>
<th>Operating system</th>
<th>Version</th>
<th>Size</th>
<th>Download method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web install using IBM Installation Manager (recommended)</td>
<td>V2.2</td>
<td>480MB</td>
<td>FTP</td>
</tr>
<tr>
<td>Web install if you already have IBM Installation Manager installed</td>
<td>V2.2</td>
<td>372MB</td>
<td></td>
</tr>
<tr>
<td>Local install using Download Director or HTTP</td>
<td>V2.2</td>
<td>785MB</td>
<td>FTP</td>
</tr>
</tbody>
</table>

**Figure 2.2 - Optim Development Studio installation packages**

While web install reduces the amount of information that needs to be downloaded, local install provides more flexibility, as you can download the package at one time and install it later, even without internet connection. For that reason, we will be describing that installation method. From the [product’s download page](http://www.ibm.com/software/data/optim/development-studio/), download the package highlighted in *Figure 2.2, Local install using Download Director or HTTP*, and proceed to the next section, where we describe the installation process in detail.
2.3.1 Installing from the launchpad

As stated earlier, starting the installation process from the Launchpad causes Installation Manager to be automatically installed if it is not already on your computer. In most cases, the default settings are sufficient, so all you need to do is accept the license, click the Next button until the Finish button is active, and then click the Finish button. After a few minutes, your installation will be complete and Optim Development Studio will be up and running!

These are the detailed installation instructions for Optim Development Studio:

1. After you unzip the download package, start the launchpad as follows:
   - Windows: Execute the setup.exe file located in the ibm_optim_dev_studio_v22_win directory as shown in Figure 2.3.

![Figure 2.3 - Click setup.exe from unzipped Data Studio package](image)

   - Linux: Execute the setup command from the root path where you've put the image.

2. Select a language and then click OK. The Welcome screen comes up. In the left pane, select Install Product as shown in Figure 2.4.
This launches Installation Manager. You will then see a screen that lets you choose which packages to install.

3. Assuming you don't already have Installation Manager on your machine, you will select the default settings to install both Installation Manager and Data Studio as shown in Figure 2.5. Then click Next.
Figure 2.5 - Install both Installation Manager and Data Studio packages

After accepting the license, click Next. You will then be presented with a screen that lets you specify the location directory for shared resources and for Installation Manager itself. You can keep the defaults; however, you’ll want to take note of the fact that you should choose a drive with more space than you think you need just for Optim Development Studio in case you decide to shell-share with other Eclipse-based products in the future,

4. As shown in Figure 2.6, take the default and then click Next.
Figure 2.6 - Select location for shared resources and Installation Manager

5. You will then see a screen that lets you choose whether to create a new package group or extend an existing one. Because we are installing on a machine that does not include any existing package groups, we can only select the radio button to Create a new package group, as shown in Figure 2.7.
In the next screen, take the default option to install the Eclipse that is included with the Optim Development Studio installation.

**Note:** If you already have an Eclipse 3.4.2 on your machine, you can choose to extend that IDE instead of installing an additional copy. This adds the functions of the newly installed product, but maintains your IDE preferences and settings. This option is available at installation time, by selecting an existing Eclipse installation directory.

6. The next screen lets you choose any additional translations you may wish to install. Select any that are appropriate and then click Next.

7. The next screen shows the lists of features to be installed; take the defaults and then click Next.

8. Finally, you are presented with a summary screen from which you can click the Install button as show in Figure 2.8.
Figure 2.8 - Review summary information and then click Install

Installation Manager will begin the installation. There may be a pause in the progress bar at some point; be sure to wait and not interrupt the processing. When the product successfully installs, you will see the screen shown in Figure 2.9.
Getting Started with pureQuery

54

Figure 2.9 - Congratulations! A successful install.

9. From the success screen shown in Figure 2.9, click on Finish to bring up Optim Development Studio.

2.3.2 Applying Optim Development Studio Fixpack 1

After you have finished installing Optim Development Studio 2.2, you should apply all the existing fixpacks. At the time of writing of this book, a fixpack is available that lets you upgrade your ODS instance to the 2.2.0.1 level.

This fixpack can be installed in two ways:

- By starting IBM Installation Manager and selecting the option Update Packages.
- By downloading the fixpack archive file and install it manually by setting up a local repository in your machine. Instructions for this installation method can be found at the official Optim Development Studio 2.2.0.1 webpage.
2.4 Running Optim Development Studio

Optim Development Studio provides additional functionality over what Data Studio provides. We strongly recommend that you read the book *Getting Started with IBM Data Studio for DB2* in order to learn more about this integrated development environment, its perspectives, views and how to use it for database administration and development.

2.4.1 Adding pureQuery support to projects

Optim Development Studio provides pureQuery tooling that delivers functionality such as code generation, performance monitoring, content assist for SQL in Java code, an SQL outline view, and more. This tooling is available for projects with pureQuery support, which can be added to any project that has a Java nature in Optim Development Studio.

**Java projects**

Optim Development Studio inherits the Java development functionality from Eclipse. If you are not familiar with Java development using the Eclipse integrated development environment, we recommend that you read the book *Getting Started with Eclipse* before you read the rest of this book.

To add pureQuery support to an existing Java project:

1. Right click in the project and select *pureQuery* -> *Add pureQuery Support*…

![Figure 2.10 - Adding pureQuery support to a Java project](image)

2. This will bring you to the *Add pureQuery Support* wizard.

3. In the first page of the wizard, select the SAMPLE connection, which will be used in this book by Optim Development Studio for all database related tasks. For instructions on how to create the SAMPLE database, please refer to the book *Getting Started with DB2 Express-C*. 
4. Click Next and you will be brought to the second and last page of the wizard, shown in Figure 2.11:
5. As shown in Figure 2.12, you can specify the settings for the pureQuery support being added to the Java project. These settings allow you to automatically configure the project’s build path to include the pureQuery JAR files, enable SQL capturing for JDBC applications and specify database application settings like the Default schema and Default path. Detailed information about SQL capturing for JDBC applications will be covered in Chapter 6 - The Client Optimizer: pureQuery for Existing JDBC Applications.
6. Click Finish and pureQuery support will be added to your project.

The following chapters will describe how you can leverage the pureQuery support provided by Optim Development Studio.

2.5 Summary

Optim Development Studio, based on open source Eclipse, is the recommended development environment for developing pureQuery applications. Throughout this book, we will use Optim Development Studio in our examples and exercises. In this chapter you learned the product prerequisites, which databases are supported, and how to install Optim Development Studio from the Launchpad. Finally, you learned how to add pureQuery support to an existing Java project, which is a wizard-driven process that enables you to specify a database connection, add the appropriate pureQuery JAR files to the project, and specify other options that will be described in more detail in subsequent chapters.

2.6 Exercise

In this exercise, you will perform a basic installation of Optim Development Studio on Windows.

Steps:

1. The Optim Development Studio image should have been downloaded from http://www.ibm.com/developerworks/downloads/im/datastudiodev/. Locate the file in the directory where you downloaded it. Unzip the files into any directory you wish.

2. Locate files. Navigate to the directory (or drive) containing the unzipped Data Studio product installation files.

3. Launch the Data Studio Launchpad by double-clicking on the setup.exe file. From the Launchpad, click the Install Product option on the left pane of the window.

4. Accept the default option to install both packages – Installation Manager and Optim Development Studio.

5. Accept the license agreement, and then click Next.

6. Select installation folders for shared resources and for Installation Manager. Select one folder for the shared resources used by Data Studio and other products who may be installed later with Data Studio and one folder for Installation Manager. Ensure sufficient space exists for the installation. Use the default drive and directory settings for this example (shown below):
Chapter 2 – Installing pureQuery (Optim Development Studio)  59

Shared Directory: C:\Program Files\IBM\SDPShared
Installation Manager: C:\Program Files\IBM\Installation Manager\eclipse

Click the Next button to continue.

7. Select the option to install a new package group for Optim Development Studio and choose the default installation directory:
   C:\Program Files\IBM\ODS2.2

8. In the next screen, take the default option to install the Eclipse that is included with the Optim Development Studio installation, then click Next.

9. Choose any additional translations you may wish to install, then click Next.

10. The next screen shows the lists of features to be installed; take the defaults and then click Next.

11. When the summary screen comes up, click Install.

12. When the successful installation screen comes up, click Finish and select the workspace to open up as follows:
   C:\Documents and Settings\Administrator\IBM\rational\GettingStarted

2.7 Review Questions

1. List and describe the different ways one can download and install the Optim Development Studio package.

2. Which Operating Systems are supported by Optim Development Studio? (answer is c))
   A. MacOS, Windows
   B. Linux, AIX, Windows
   C. Linux, Windows
   D. Linux, MacOS, Windows

3. What is the option the user needs to perform in order to enable pureQuery tooling support over a Java project in Optim Development Studio?
   A. Enable the workspace preferences setting Activate pureQuery Tooling Support
   B. Enable the project preferences setting Activate pureQuery Tooling Support
   C. From the project’s context menu, select pureQuery -> Add pureQuery Support…
D. From the project’s context menu, select `pureQuery -> Enable pureQuery Tooling Support…`
PART II – TOOLING
Chapter 3 - Code generation and customized Java Editor Integration

This chapter introduces how to use pureQuery tooling in Optim Data Studio (ODS) to generate pureQuery code as well as many other benefits that the tooling provides to boost your productivity as a developer.

3.1 pureQuery Tooling: The big picture

The pureQuery tooling from ODS makes Java development with SQL simpler, faster and more productive than ever before. You can rapidly develop or customize the SQL inside Java applications to your needs using a sophisticated SQL editor integrated inside the Java editor. Furthermore, through a set of easy-to-use wizards, the tooling provides the ability to generate pureQuery Java code to access and manipulate data from tables, views and stored procedures.
In this chapter you will learn about:

- Generating pureQuery code from different database objects
- EMFT JET template customization for code generation
- Java Editor integration: content assist, SQL validation

3.1.1 Generating pureQuery code

The auto-generation of code increases your productivity as a developer by decreasing the amount of repetitive programming for the basic CRUD operations. Using the pureQuery tools from ODS, you can generate pureQuery data access code, sample programs or JUnit test cases and run them without writing a single line of code for simple operations. Another commonly used time-saving tool that pureQuery code generation provides is the ability to write an SQL statement manually and then automatically generate pureQuery Java code from it. Users have found this to be one of the greatest time savers provided by pureQuery code generation.
Using the pureQuery tooling, you can generate pureQuery data access code from tables, views and stored procedures from any IBM database (DB2 LUW, DB2 for z/OS, IDS, iSeries) as well as for Oracle databases. Developers often start working on applications either with a sample application, using tooling to generate stubs and then modify them to fit their needs, or simply by developing the entire application from scratch. Whichever the approach is, the pureQuery tooling enhances the user’s productivity for all scenarios.

Figure 3.2 - Action to generate pureQuery code from a database table

To generate pureQuery code from a database table right-click on the table from the Data Source Explorer and select Generate pureQuery Code… from the context menu as shown in Figure 3.2. A wizard, seen in Figure 3.3, opens asking you to enter values specifying the source folder within a Java project to put the generated code into and a Java package for the generated Java classes. For the Name field, enter a value which will be used as the name of the table as the default value. Enter the name of a super class for the bean if applicable. When developing your application using the annotated-method style, select the option Generate annotated-method interface for table and enter the Java package name for the pureQuery interface and the interface name. Some of these values may already be populated with defaults.
Figure 3.3 - Generating a pureQuery bean and interface from a database table

Click Next, and on the second page of the wizard you can select to generate a test program to execute some of the generated CRUD (create, retrieve, update and delete) methods of the annotated-method style interface you selected in the previous page. You can also select the option to generate an inline-style class. These test programs can be generated either as a class with a main method or as JUnits. For the test classes, you can either select to include the connection information such as the database connection string (url) and user ID or pass these two values as arguments. The password has to be passed as an argument when executing your test class. For generating the bean, you can choose to change the field names as required and get the mapping to the database automatically generated for you. The pureQuery APIs, for example, use beans to hold the result of a select statement—a bean represents a row/record in a database. You can also choose to generate public fields or protected fields with public getters and setters. If you need custom Java type mappings, you can simply change the default ones in the Bean Fields part of the code generation wizard as shown in Figure 3.4, or use the @Handler annotation.
Depending on your selection, the pureQuery tooling can generate some sample statements to retrieve, insert, update, delete and merge data from your table. When generating these statements, the pureQuery tooling uses the primary keys definition on your table to construct the \texttt{WHERE} clause of your SQL statements where applicable. You can choose to either use \texttt{"*"} or to list all columns explicitly for the \texttt{SELECT} statements as shown in Figure 3.5. After clicking \texttt{Finish} the wizard will generate the code for you. You can easily customize this code to your needs by modifying the SQL statements generated, adding new and modifying existing methods and corresponding SQL to your pureQuery interface. If using annotated-method style, the implementation to the pureQuery interface will automatically be re-generated by the pureQuery generator if you have the \textit{Build Automatically} setting checked under the \textit{Project} menu.
You can also create data access code for a large number of tables quickly by selecting those tables, right-clicking and selecting *Generate pureQuery Code...* The SQL CRUD statements however, are generated separately for each table; therefore, relationships among tables are not considered.

### 3.2 Overview of the different types of wizards for code generation within the pureQuery tooling.

As illustrated in the previous section, a user can generate pureQuery code for the data access part of the application from a table using the pureQuery tooling. Similar types of code generation wizards exist in the tooling: for example, generating pureQuery code from an SQL statement (such as a JOIN), generation of code for calling stored procedures and from SQL statements, or helping users to quickly generate JUnit test classes for their
interfaces, to mention a few. Refer to the following article at www.ibm.com/developerworks/data/library/techarticle/dm-0709surange/ for further details.

### Note:
You can access the help from any of the wizards by pressing F1 at any time, or by clicking the question mark icon at the bottom left of each wizard’s page. This help will provide you with more detailed information and instructions of the respective pureQuery tool being invoked.

#### 3.2.1 Stored procedure code generation wizard
For stored procedures, the pureQuery tooling generates a bean to represent the parameter values (IN, OUT, INOUT) if any exists, and other beans for each result set returned. An interface containing the `CALL` statement is generated if the user chooses annotated-method style code. You can optionally ask the wizard to generate test classes with a main method or JUnits for running the generated code.

#### 3.2.2 Generating or updating JUnit testcases for existing interfaces
Developers and testers often write tests such as JUnit test cases to test basic functionality. With the pureQuery tooling a user can create or update those testcases fast and easy from a pureQuery interface by right-clicking on the interface and selecting纯Query — Generate JUnit. Note that if an annotated method takes a parameter, the test method contains comments to indicate that you must assign a value to such parameters. For example, the following is a test case generated to test a method that retrieves an `Employee` object based on the employee number `empno`. The user must assign a `String` value to the `empno` variable.

```java
@Test
public void test2_getEmployee() throws Exception{
    String empno = null;
    //TODO: empno = String value
    //Execute SQL statement
    Employee bean = data.getEmployee(empno);
}
```

#### 3.3 SQL editor integration inside the Java editor
The pureQuery tooling provides SQL integration within the built-in Eclipse Java editor. This provides Java developers with a tremendous boost in productivity by helping them catch errors in their SQL at development time. You can also test those SQL statements, launch Visual Explain and use SQL content assist functionality to help you easily build a query without leaving the Java editor. This SQL integration features are automatically added to
the Java editor when you choose pureQuery support for your project. You can use these features for applications using the pureQuery APIs and for JPA, Hibernate or JDBC applications. DB2 LUW, DB2 for z/OS, IDS, iSeries and Oracle databases are also supported. For more details on these features and many more, refer to the article at http://www.ibm.com/developerworks/data/library/techarticle/dm-0709surange/.

3.3.1 SQL validation within the Java editor

Java developers often write applications that contain complex SQL and you as a Java developer may not be an expert in writing the SQL queries you need. It is also sometimes difficult to remember the exact table or column names. You must either refer to some complex schema drawing or DDL, or you may mistype the column name in your SQL statement. The Java editor reports SQL errors while you type SQL statements, and shows them in the same way that Java language errors are shown. Figure 3.6 shows an example SQL statement with a column name misspelled, missing the letter R. Errors in SQL keywords (syntactic) and schema, table, column or other database names (semantic) errors are detected.

![Figure 3.6 - SQL errors are flagged while typing in the Java editor](image)

3.3.2 SQL semantic and syntactic content assist for SQL inside Java editor

The Java editor provides SQL content-assists and validates syntax by using live database connection settings. To get SQL content assist inside the Java editor, press (Ctrl + Spacebar) the Ctrl key and spacebar (or your default content assist keystroke) to popup the content-assist window, as shown in Figure 3.7. The content-assist popup window helps you:

- Discover and select SQL keywords
- Discover and select from the list of table and column names
- Use content-assist proposals to see database types and primary keys
3.3.3 Test and run SQL within your Java program

Developers often spend significant time writing test applications to test a SQL statement for validity or expected results. With the pureQuery tooling, you can run SQL with or without parameters at design time. Right-click on any SQL inside your Java program and select pureQuery → Run SQL, provide any parameters if needed and view the results in the SQL Results View as shown in Figure 3.8.

3.4 Summary

Java development with inline SQL is made simpler and more convenient with an SQL editor integrated inside the Java editor. This provides Java developers with a tremendous boost in productivity by helping them catch errors in their SQL at development time. With some easy to use wizards, you can quickly generate data access Java code from tables, views or stored procedures. To generate code, the pureQuery tooling uses EMFT JET templates and can be overridden to provide customized code generation. Data access code can be generated from tables, views and stored procedures from any IBM database as well as for Oracle databases.

3.5 Review Questions

1. Which of the following databases are NOT supported by the pureQuery tooling for data access code generation?
1. Which of the following database platforms is supported by pureQuery?
   A. Informix (IDS)
   B. MicroSoft SQL Server
   C. iSeries
   D. Oracle

2. Which of the following ODS views would a user go to generate data access code from a database table?
   A. SQL Outline
   B. Data Project Explorer
   C. Data Source Explorer
   D. Code Generation View

3. Which of the following settings/values are used for proposing SQL content-assist and validation within the Java editor?
   A. Default Schema
   B. Default Path
   C. Database Connection
   D. All of the above

4. A user can generate data access code through the pureQuery tooling from which of the following database objects?
   A. Table
   B. View
   C. Stored Procedure
   D. All of the above

5. If the user wishes to use their own pureQuery Runtime jars in their project and not the ones that come with the ODS installation, what settings must the user change?

6. When or why would a user decide to work with an offline connection?

7. When or why would a user want to override SQL statements found in the pureQuery interfaces?

8. If the user either generates or updates their JUnit test cases for existing pureQuery interfaces using the pureQuery tooling, what must the user do before executing the JUnits if their annotated-method takes a parameter?
Chapter 4 - pureQuery tooling Views

In the following chapters you will learn how you can use the Client Optimizer feature of the pureQuery runtime to collect SQL statements from your application (Chapter 9) and then use the Static Binder utility (Chapter 10) to bind those statements into DB2 packages to gain better performance and a better authorization model for DB2 applications. Optim Data Studio’s (ODS) pureQuery tooling includes a set of features to assist you in performing these tasks faster and more easily whether you are a developer or an application-centric database administrator (DBA).

4.1 pureQuery tooling: The big picture

The pureQuery tooling from ODS provides a set of views and editors to help you optimize your new or existing applications (JPA, Hibernate, JDBC, pureQuery API) without changing a single line of code. The tooling includes problem isolation and impact analysis capabilities through the introduction of the **SQL Outline** view and the **pureQuery Analysis** view.

The **SQL Outline** view gives developers the ability to drill down into the application to find where a specific SQL statement was executed. It improves the collaboration between DBAs and developers, for example, by allowing developers to quickly and easily gather and export all statements into a file for the DBA to analyze and give feedback:

![Figure 4.1 - SQL Outline View](image)
The *pureQuery Analysis* view gives developers the ability to drill down into the SQL execution for “where used” analysis.

### 4.2 SQL Outline view

If your pureQuery enabled project contains the source code, you can gain insight and view the statements on the SQL Outline that your application issues without having to execute your application in capture mode—refer to Chapter 10 for more information on capturing SQL statements. This is true for JPA, Hibernate, JDBC or any application that use the pureQuery API. You may not see all of the SQL statements within your applications if you have coded your statements based on variables and then passed those variables to the JDBC methods, for example. In such cases, you may see the SQL Outline showing an expression and not the actual SQL statement. You can get detailed help of the SQL Outline at any time by selecting it and pressing F1 which will launch the built-in Help in ODS.

![Figure 4.2 - SQL Outline view Java tab](image)

**4.2.1 Gather the SQL statements issued by the application**

If your application is written using the pureQuery API, then the outline view will automatically show all your SQL statements. However if your applications is coded directly in JDBC and not in the pureQuery API, if you want to view all SQL statements issued by your application to the database, then you will have to run your application using the Client Optimizer feature from the pureQuery runtime. To leverage this feature, you must **Enable SQL capturing and binding for JDBC applications** when either adding pureQuery support to your project or from the project’s properties page as shown in Figure 4.3.
When enabling this feature, a `pdq.properties` file will be added to your project. By setting property `pdq.captureMode=ON`, you can enable/disable the capturing of statements being executed by your application.

**Note:**

The `pdq.properties` file, as well as other files added to your project under the `pureQueryFolder`, come with content-assist capabilities. For the `pdq.properties` file, press the `Ctrl + Space` keys after typing “pdq.” and a list of properties with their respective description will show.

You must execute each statement in order to capture it. Once you have executed and gathered all the SQL statements you can export them to a SQL file. You can give this file to
a DBA for optimization or view the statements yourself in the SQL Editor. Right-click on the Java file node in the Java page and select **Show in SQL Editor** as shown in Figure 4.4.

![Figure 4.4 - Export all SQL statements within a Java file to a SQL script file](image)

**4.2.2 Gain insight into where SQL statements are issued within your application**

Often, database application developers and DBAs have difficulty isolating poorly performing SQL, finding the statement in the source code or simply trying to determine which of the statements were issued against the database by a specific application. Finding where exactly in an application a specific statement was executed involved gathering and analyzing database driver traces. These traces often give limited information, making it difficult to get the exact place or line within a Java source file.

The SQL Outline makes these tasks easy by correlating SQL statements to a particular line number of a Java class even when a framework such as JPA or Hibernate generates the SQL. Once you have captured your SQL using client optimizer, all the SQL in your application is visible in the SQL Outline, including JDBC executed statements or SQL written using the pureQuery API.
Figure 4.5 shows a SQL statement that was executed through a JDBC `executeQuery` method and captured through the Client Optimizer. Either double-clicking on the SQL statement or right-clicking and selecting `Find in Source` will take you to the exact line number where the statement was executed.

4.2.3 Pages of the SQL Outline view

The SQL Outline view contains three tabbed pages that show you the relationship between DB2 packages, Java files and SQL statements in different ways. Click the tabs at the bottom of the view to see the different pages. Each view emphasizes a different perspective of the SQL useful for troubleshooting and analysis activities.

4.2.3.1 Database Page

The database page of the SQL Outline groups SQL statements by schema and table of the database objects that the statements refer to. The database page is useful for analyzing the extent to which changes to a certain table or view will require changes to an application. The same statement may fall under more than one schema or table if the statement makes reference to more that one of those objects, as in case of joins for example. By expanding the nodes, you can get information such as which columns are
referenced by the statement, the Java project, package, class, method and line number where the statement was executed. Figure 4.6 shows the line number where the statement was executed (line 27) and where it was prepared (line 24).

Figure 4.6 - Database Page

4.2.3.2 Java Page

The Java page of the SQL Outline is usually the most interesting to Java data access developers since it associates SQL statements with java classes, and presents them by qualified class name. It groups statements by Java projects that are open and enabled with support for pureQuery. This page shows nodes of the methods issuing and or executing the statement. By expanding these nodes, the user can see the SQL statement issued to the database.
4.2.3.3 Database Packages

The Database Packages page groups all SQL statements by packages. If you were to bind your application, this would be the same structure of the packages that you created on your database.

**Note:**

The Database Package page may sometimes be empty while the other two pages show some content. This may be due to your database not supporting static SQL and the creation of database packages as is the case for projects associated with an IDS or Oracle database connection. Another reason may be due to not having executed your application using the Client Optimizer feature from the pureQuery runtime. After running your application using the Client Optimizer, you would have gathered all statements and configured them into packages needed to bind and for the creation of database packages for static SQL execution.

*Figure 4.7 - Java Page*

*Figure 4.8* shows two packages: one with one SQL statement that came from the execution of a JDBC application, and the other from an interface generated by right-clicking on the EMPLOYEE table and selecting *Generate pureQuery Code...* You can configure the database packages before binding through the Default.genProps file. Use the Default.bindProps file to specify options for determining how SQL statements are bound into DB2 packages. These two files are found under the pureQueryFolder, refer to the ODS build-in help for more details.
4.2.4 Enhanced impact analysis within database applications

With the SQL Outline, you can gain insight into what parts of your application use certain tables, columns or views by determining the objects referenced in the statements. The database design often goes through changes during the application development stage. The inability to gain insight into many different areas of an application can make simple changes to a schema or a table a risky operation, as those changes may adversely affect other parts of the application without the knowledge of the user.

The impact analysis features of the pureQuery tooling help DBAs and developers to work together to come up with database changes that would have the least impact on an application, thus mitigating risks.

In the Database page, for example, you can gain insight into how much your application might need to change if a given table or view were to change. By expanding the table, you can gain insight into all the SQL statements affected and their location within your application. In the case of a change in a table’s column, such as a name change or simply by dropping it, you can use the SQL Outline filter and provide the name of the affected column to view all the affected statements. The filter can either be invoked by right-clicking anywhere on the SQL Outline and selecting Filter… from the context menu or by clicking on the inverted triangle on the toolbar and selecting Filter…

For example, if you want to assess which statements are impacted by dropping the BONUS column on the EMPLOYEE table to then go to the application and change the code where necessary, you would use the filter as shown in Figure 4.9. You can see the statements that use the BONUS column only and narrow down to the specific place where those
statements where executed within your application to make appropriate changes as a result of dropping the column.

Figure 4.9 - Assessing the impact analysis of dropping a column within an application

The filter also includes settings to filter statements by types such as `SELECT`, `UPDATE`, `MERGE` as well as provides you the ability to search for text within the SQL statement in the form of a regular expression that conforms to the Java `Pattern` class.

4.3 Designing and binding packages for static SQL development

ODS, through its pureQuery tooling, makes static SQL development and deployment easy by providing you with tools to design, deploy and test database packages effectively on several database from test to production.
4.3.1 Designing database packages

ODS provides an editor with content-assist for the Default.genProps file where you can provide properties such as the package name, collection and version of the database package to be deployed to the database as shown in Figure 4.10.

Figure 4.10 - Changing the properties before deployment

Once you have entered your properties, save the file and click Yes on the dialog asking you to re-build your project for the changes to take effect. The SQL Outline will automatically change to reflect these properties if needed such as a change in the package name.

You can also design packages through the pureQueryXML Editor for capture files. In this editor, for example, you can create a new package and drag-and-drop statements from one package to another within the same capture.pdqxml file or delete statements that you do not wish to run as static SQL—see Figure 4.11.
If you notice a statement that is not very efficient, you may want to replace it with a more optimal one. However, you may not have the source code to change this statement or the change may be considered too risky.

For example, you may want to change a statement to take advantage of a newly added index, to limit the number of rows returned or simply to add an `ORDER BY` clause. With the pureQueryXML editor, you can edit the statement by providing an alternative one that is equivalent but better in performance. This alternative statement will now be executed and included in your database package when you execute the bind operation, as shown in Figure 4.12. For further details refer to the ODS build-in documentation or refer to the article at [https://www.ibm.com/developerworks/data/library/techarticle/dm-0812surange/](https://www.ibm.com/developerworks/data/library/techarticle/dm-0812surange/).
You can preview the packages in design mode to ensure that it has the right name, the correct SQL and so on, on the Database packages page of the SQL Outline. You can repeat the “design→preview” cycle by modifying the `Default.genProps` or the capture file through the pureQueryXML editor until the package is ready for binding. To ensure that the change in properties did take place, select the package on the SQL Outline and view the properties on the Properties View of ODS.

Note:
You can use the pureQueryXML editor to replace statements with better performing ones even if you are not going to use static SQL, as is the case with developers using an IDS or Oracle database. The runtime will execute the replacement statement instead of the original captured one (the one in your application). To allow for the replacement SQL statement to be executed, the `enableDynamicSQLReplacement` property set to TRUE has to be set on your `pdq.properties` file.

4.3.2 Binding database packages

The pureQuery tooling also provides an editor with content-assist where you can provide `bindOptions` and set an `isolationLevel` to bind, for example. These options are set on the `Default.bindProps` file under the `pureQueryFolder`. The following is an example of specifying one isolation level and a qualifier when binding:

```
my.pdq.DAO.code.EmployeeData = -isolationLevel CS -bindOptions “QUALIFIER DB2ADMIN”
```

When the package is ready for binding, you can bind to the development, test or production database without changing your development environment. To bind a package, right-click on the package in the SQL Outline and select Bind… Select a connection of the database to bind to and click Finish, the Console View will print details of the binding process or print any errors or warnings if any as shown in Figure 13.13.

Figure 4.13 - SQL Outline – deploying database packages
You can also invoke the bind command at the project level by right-clicking on the Java project and selecting pureQuery → Bind pureQuery Application. This action will bind all interfaces and capture files that you may have in your project. You can also bind a specific pureQuery interface, capture file, or a JAR file that may contain interfaces by right-clicking on the artifact and selecting pureQuery → Bind…

As show in Figure 4.14, to verify that your package was deployed properly to the database, either double-click or right-click on a specific package in the SQL Outline and select Find in Data Source Explorer.

4.4 pureQuery Analysis view

The pureQuery Analysis view shows how the values of a returned result set flow through the Java application. For example, you can use it to see how and where a retrieved column value of a statement is used. It can help you answer questions such as: Is it sent back to the database or printed out in a report or simply to the Console View?

Figure 4.15 shows how the EMPNO column, of the EmployeeData interface generated in the previous chapter, flows through the application: right-click on the column in your interface and select pureQuery → Source Code Analysis → Analyze Column Value Dependency. Note that the database icons highlight where the value is retrieved from the database and updated to the database.
4.5 Enforce data privacy from modeling through development

Production databases often contain sensitive information such as credit card or social security numbers. When data architects use InfoSphere Data Architect to create data models for such databases, they can identify which attributes or columns contain sensitive information. Then by associating this model to a database connection in the Data Source Explorer (DSE) view, developers can easily see or track the columns that are identified as containing such sensitive information. Using the SQL Outline and pureQuery Analysis views you can see where within your application those private columns are being used in context to ensure that the data is not being used inappropriately such as printing out sensitive data from those columns.

In the SQL Outline, applications that use SQL containing columns with privacy information show the SQL with a privacy icon (ющим). You can use the filter in the SQL Outline to show only the SQL statements that contain a column with privacy information by selecting to see only SQL containing columns marked as private. The results are shown in Figure 4.16.
Figure 4.16 - Viewing SQL statements that have columns with private information

The pureQuery Analysis view takes this a step further where you can filter and, for example, see where in the code is the privacy column `SALARY` printed to the Console View. Use the filter dialog in the pureQuery Analysis view and select *Code that prints column values to the console* and click *OK*. Now position your cursor on the `SALARY` column in the `EmployeeData` interface and select *pureQuery → Source Code Analysis → Analyze Column Value Dependency* as shown in Figure 4.17.
The pureQuery tooling from ODS provides a set of views and editors to help you optimize your new or existing applications without changing a single line of code. The tooling includes problem isolation and impact analysis capabilities through the introduction of the SQL Outline view and the pureQuery Analysis view.

The SQL Outline makes tasks easy by correlating SQL statements to a particular line number of a Java class even when a framework such as JPA or Hibernate generates the SQL. With the SQL Outline, you can gain insight into what parts of your application use certain tables, columns or views by determining the objects referenced in the statements.

The impact analysis features of the pureQuery tooling helps DBAs and developers to work together to come up with database changes that would have the least impact on an application, thus mitigating or eliminating risks.

The pureQuery Analysis view shows how the values of a returned result set flow through the Java application. For example, you can use it to see how and where a retrieved column value of a statement is used. It can help you answer questions such as: Is it sent back to the database or printed out in a report or simply to the Console View?
4.7 Review Questions

1. If the developer wants to export the SQL statements executed by the application so that a DBA can review and optimize the statements, from which ODS view would the user invoke this functionality?

   A. SQL Outline
   B. pureQuery Analysis
   C. Data Source Explorer
   D. SQL Explorer

2. If a user wishes to see the SQL statements that make use of a specific table under a specific schema, which of the following tabs within the SQL Outline would the user look?

   A. Database page
   B. Java page
   C. Database Packages page
   D. Schema Page

3. In which of the following property files found under the `pureQueryFolder` within your project would a user specify properties such as the name and version of the database package to be bounded?

   A. Default.bindProps
   B. Default.genProps
   C. pdq.properties
   D. package.properties

4. If the developer wishes to see how the values for a returned result set flow through the Java application, which of the following views would the developer refer to?

   A. SQL Outline
   B. pureQuery Analysis
   C. Data Source Explorer
   D. Java Source Flow

5. The Filter dialog in the SQL Outline, lets the user filter SQL to only show statements by which of the following settings:

   A. Database objects
   B. SQL statement types
   C. SQL containing columns marked as private
D. all of the above

6. If the developer has a data model enforcing data privacy associated to a connection in the DSE, how would the user choose to see in the SQL Outline only those SQL statements containing columns with the information flagged as private?

7. The SQL Outline may not always show all the statements found within the source code of your pureQuery enabled project. One possibility could be that the statement string was assigned to a variable and the variable was passed to the JDBC method that prepares or executes it. What would the user need to do in order to view all statements that were executed by the application regardless of how they appear in your source code?

8. How do you enable the capturing of SQL statements that are executed within your pureQuery enabled project?
Chapter 5 - Developing for performance

In this chapter we will discuss how tools can help you gain better insight into the performance of specific SQL statements and get advice to improve and tune your database. Later chapters will discuss ways pureQuery can help you solve the performance problems you might uncover in this chapter. Chapter 6 and Chapter 7 will discuss how through client optimization you can collect SQL statements and then use the pureQueryXML Editor (Chapter 3) to replace a specific statement with an optimized one in order to increase performance. Chapter 4 will introduce how, through tooling, you can use the Static Binder utility and create database packages to run static SQL and increase performance when executing your application.

5.1 Developing for performance: The big picture

With ODS, through the SQL Outline, you can gain insight into how much time was spent executing a specific SQL statement and how often was that statement executed. You can also gather and view Explain data to quickly find the most costly queries.

Figure 5.1 - Identify the SQL statements and the frequency of their execution
From the SQL Outline or through the Java Editor, take advantage of ODS integration with Optim Query Tuner (OQT) to get expert advise on improving your queries and database design through its easy-to-use advisors.

5.2 Identify the time and frequency of execution and cost of a statement

Though the SQL Outline, you can drill down and see the performance metrics for any SQL statement that was executed providing you with information to quickly and easily identify the statements that would need to be optimized.

**Note:**
You can view performance metrics for applications that use the pureQuery API—method-style or inline-style, as well as JDBC, Hibernate or JPA applications.

The performance metrics provide you information such as:
- Number of times the SQL ran
- Total execution time—adds the time for each run of the same SQL
- Longest time the SQL took to run
- Average of time that the SQL took to run
- Minimum time the SQL took to run

5.2.1 Gathering and viewing performance metrics

To view the performance metrics when executing your Java application, you must run your application under the PureQuery configuration and not as a regular Java Application.
Figure 5.2 - Executing your application to gather performance metrics

For this example, we will gather the performance metrics by executing the
EmployeeInlineSample.java class generated from the EMPLOYEE table of the
SAMPLE database—refer to Chapter 3 for generating pureQuery code from a table. To
execute the class, right-click on the file and select Run As → Run Configurations… On the
Run Configurations dialog, create a new PureQuery configuration and enter your database
password under the Arguments tab under Program arguments and click Finish as shown in
Figure 5.2.

After your application has finished executing, right-click on your Java project and select
pureQuery → Show SQL Outline from the context menu. On the SQL Outline, right-click
and select Show Data → Show Performance Data and expand the nodes in the Database
page until you see the SQL statements. Note that you will see the performance metrics for
only those statements that were executed when you ran your application as shown in
Figure 5.3.
Figure 5.3 - View the execution time and frequency for statements executed

If a specific SQL statement was executed more than once, expand the statement node and view the times for each and every execution of that statement.

Imagine that your application has executed many statements and you are interested in finding which statements take the longest or are executed more frequently so that you can concentrate your tuning efforts on these statements. The SQL Outline can be viewed in a table format where you can sort the column you are interested in, either in ascending or descending order.

Right-click on the SQL Outline and select Show SQL in Table, then click on the heading of the Max Time column to show the information in descending order as shown in Figure 5.4.

Figure 5.4 - Finding statements that took the longest to execute
5.2.2 Gathering and viewing explain data

You can gain insight into the health of your SQL statements by obtaining information such as the predicted cost, whether indexes are used, or the number of joins used for example—Oracle and IBM databases, including IDS, are supported.

To view explain information, right-click on the SQL Outline and select Show Data → Show EXPLAIN Data. If you would like to see this explain information for a specific SQL statement, right-click on the statement in the SQL Outline and select Retrieve EXPLAIN Data from the context menu. To retrieve the explain information for all statements within an application right-click on the Java project and select Properties. On the left-hand-side expand the pureQuery category and select Background EXPLAIN. Then select to Enable project specific settings and to Enable Background EXPLAIN and click OK. Explain information will start to be gathered for all of the statements in the SQL Outline belonging to that specific Java project. Once all the explain data has been gathered, it will automatically show next to SQL statements if you have chosen to view EXPLAIN Data in the SQL Outline as shown in Figure 5.5.

![Figure 5.5 - View explain information for SQL statements within your application](image)

You can also change the view of the SQL Outline to show in a table format—Figure 5.4—in order to sort the columns if you are interested in finding the statement with the highest cost, for example.

5.3 Optimizing your application for performance

Once you have identified which statements take the longest to execute or any other statement that you believe needs optimizing you can do one or a combination of the following:

- replace a poorly performing statement with an optimized one using the pureQueryXML Editor
• bind your application to use static SQL
• enable literal replacement with parameters during the capture process
• use the Optim Query Tuner to give you advice on improving your queries or database design.

To replace a poorly performing statement with an optimized one, to use static SQL or both, refer to Chapter 4.

5.3.1 Optimize your applications by using Optim Query Tuner

DBA’s are frequently more interested in fine tuning SQL statements than developers are. However, if this task could be made easier, developers could take certain steps to try to improve application performance during the development stage to avoid expensive or risky changes during the production stage. With Optim Query Tuner (OQT), you can easily get advice on writing more efficient SQL or get enough information to ask for new indexes or statistics from the database administrator.

Note:
For more information on this topic, refer to the Developer Works article at https://www.ibm.com/developerworks/data/library/techarticle/dm-0906optimdeveloper/ or the article SQL Tuning on the IBM Data Management Magazine, Issue 2, 2009 pp. 22-25.

Optim Query Tuner is a separate product that integrates with ODS and can be launched from either of the following places by selecting the Open Query Tuner menu:

• right-click on a SQL statement in the Java Editor
• right-click on a SQL statement in the SQL Outline
• right-click on a SQL statement in the pureQueryXML Editor

To launch OQT, right-click on the SQL statement which took the maximum time to execute—Figure 14.4—and choose Open Query Tuner from the context menu.
Figure 5.6 - Getting advice from Optim Query Tuner

Now view the advice and make changes accordingly. In the example above, Optim Query Tuner recommends that you update statistics. This can be run directly from OQT. You might need certain permissions from the DBA to run these statistics.

5.4 Comparing performance results after optimizing your application

Once you have made modifications either directly to your application or to your database in order to boost performance, you can re-run your application while gathering performance metrics and compare the new metrics gathered with the older ones prior to optimization.

Save your current performance metrics by clicking on the Save Current Performance Data button and enter a name to describe the metric values, then click OK as shown in Figure 5.7.
Figure 5.7 - Saving performance metrics for later comparisons

Re-run your application using the PureQuery configuration as shown in Figure 5.2. On the SQL Outline click on the Compare checkbox. Figure 5.8 shows the comparison between the times prior to optimization and post optimization. The changes undertaken to improve performance were to run the recommendations provided by OQT and running the application using static SQL.

Figure 5.8 - Comparing performance metrics among different executions

The left side of the Total Time column shows the time taken after optimization. Note that the execution of the MERGE and UPDATE statements after optimizing took about one-third of the time—executing the MERGE statement before optimization took three times as much as after optimization.

5.5 Summary

With ODS, through the SQL Outline, you can gain insight into how much time was spent executing a specific SQL statement and how often was that statement executed as well as gather and view explain data to quickly find the most costly queries. The SQL Outline can be viewed on a table format where you can sort the column you are interested in either in
ascending or descending order to view which statements take the most time to execute, for example.

Once you have identified which statements take the longest to execute or any other statement that you believe needs optimizing you can easily replace a poorly performing statement with an optimized one using the pureQueryXML Editor. Developers can take certain steps to improve application performance during the development stage to avoid expensive or risky changes during the production stage.

With Optim Query Tuner, which easily integrates with ODS, you can easily get advice on writing more efficient SQL or get enough information to ask for new indexes or statistics from the database administrator. Save the performance metrics, replace your SQL with better performing one and re-run your application and compare the new results with the old ones.

5.6 Review Questions

1. Which of the following information can be seen in the SQL Outline pertaining to the execution a SQL Statement?
   A. Number of Times Run
   B. Total Time
   C. Max Time
   D. all of the above

2. Performance metrics for the execution of SQL statements can be gathered for which of the following types of applications?
   A. JPA
   B. JDBC
   C. Hibernate
   D. all of the above

3. Under which Configuration within ODS must a user run their application in order to gather the performance metrics?
   A. Java Application
   B. PureQuery
   C. Eclipse Application
   D. Java Applet

4. In which of the following ways can you optimize your application for performance through ODS?
   A. replace poorly performing SQL through the pureQueryXML Editor
B. bind to create database packages to use static SQL
C. user OQT for advice on improving queries and database design
D. all of the above

5. From where within ODS can you launch OQT for advice on optimizing queries or the database design for performance?
   A. SQL in the Java Editor
   B. SQL in the SQL Outline
   C. SQL in the pureQueryXML Editor
   D. all of the above

6. Describe the steps that a user would need to do in order to see the performance metrics after executing their application?

7. Describe the steps that a user would need to do in order to see which statements take the longest or least time to execute?

8. Explain the differences between the Retrieve EXPLAIN Data menu in the SQL Outline and the Background EXPLAIN subsection of the pureQuery category in the projects Properties.
PART III – SECURITY, PERFORMANCE, AND INSIGHT
Chapter 6 - The Client Optimizer: pureQuery for Existing JDBC Applications

In this chapter, you will learn how to optimize existing JDBC applications by using the pureQuery Client Optimizer.

6.1 Client Optimizer: The big picture

The pureQuery Client Optimizer can help you enhance the performance, security, and maintainability of your existing applications, usually without changing the source code. The Client Optimizer, also known as “pureQuery for existing applications,” lets you optimize applications at the runtime level. You can use it to optimize JDBC applications, as well as applications developed using persistence frameworks that are based on JDBC.

Figure 6.1 shows some of the many benefits that the pureQuery Client Optimizer provides for JDBC applications.

![Diagram showing benefits of the Client Optimizer]

**Figure 6.1 - Some of the benefits of using the Client Optimizer**

**Static SQL:** With the Client Optimizer, you can convert dynamically-executed SQL into static SQL packages. Doing so will help you deliver consistent performance and an enhanced security model with your application. For a brief description of static SQL and its powerful benefits, see Appendix C.
Captured-only SQL: You can use the Client Optimizer’s “captured-only” feature to restrict the SQL statements that an application can execute to those in a whitelist that you approve, even if the SQL statements execute dynamically. This can help you to improve the security of any application. This is particularly useful in environments that do not offer static SQL.

SQL Replacement: Using the Client Optimizer for an existing JDBC application makes it easy to substitute executed SQL with optimized equivalent SQL. This can help you to leverage the DBA skills in your organization to optimize SQL statements that are performing poorly, without modifying the application.

SQL Literal Substitution: The SQL Literal Substitution feature allows you to convert non-parameterized SQL statements that your application executes into parameterized SQL statements. This feature is particularly useful for applications that generate SQL statements during runtime. SQL Literal substitution can enable you to use static SQL, captured-only SQL, or both for applications that generate non-parameterized SQL statements. Additionally, for SQL statements that execute dynamically, it can reduce the total number of SQL statements in the cache, thereby increasing the cache-hit rate for statements.

Locating SQL Statements: The Client Optimizer helps you to identify quickly which parts of your application execute particular SQL statements. When your application executes SQL statements, the Client Optimizer can record stack traces that indicate which part of your application executed the statements.

In addition to the enhancements that you can make to your application using the Client Optimizer, you can leverage the integration between the Client Optimizer and the pureQuery tooling in IBM Optim Development Studio to easily analyze and improve the SQL statements in your existing applications. Using the Client Optimizer for existing JDBC applications also increases the amount of information that is available about the SQL statements that your application executes, if you use the Extended Insight feature (See Chapter 11 for more details).

6.2 Overview of how to use the Client Optimizer

You must follow four basic steps to using the Client Optimizer to optimize a JDBC application. All of the steps deal with a particular type of file, called a pureQueryXml file, which describes the SQL statements in the application. The extension of a pureQueryXml file must be either .pdqxml or .xml. In the first and fourth step, properties known as pureQuery properties determine the behavior.

1. **Capture**: Direct pureQuery to capture the SQL statements from your application into a pureQueryXml file. Figure 6.2 shows pureQuery creating a pureQueryXml file containing the SQL statements that your application executes. A
pureQueryXml file that pureQuery’s capture functionality creates is called a capture file.

![Diagram](image)

**Figure 6.2 - During capture, the Client Optimizer captures SQL statements that your application executes into a pureQueryXml file.**

2. **Configure:** Process the pureQueryXml file with the Configure utility. This simple procedure prepares the file so that the SQL statements in it can be bound in the database, and so that pureQuery can reference the statements in the file during the execution of the application. If you are using IBM Optim Development Studio, the tooling runs the Configure utility automatically. Figure 6.3 shows the Configure utility configuring a pureQueryXml file.

![Diagram](image)

**Figure 6.3 - The Configure utility prepares a pureQueryXml file for use in binding and execution.**

3. **Bind:** If you intend to use static SQL, use the StaticBinder utility to bind the SQL statements that are in the pureQueryXml file into packages in the database. If you do not intend to use static SQL, you can skip this step. Figure 6.4 shows the StaticBinder utility binding packages of SQL statements in the database.
Getting Started with pureQuery

1. **Bind**

   - **Configured pureQueryXml File**
   - **pureQuery StaticBinder Utility**
   - **Packages of SQL statements**
   - **Database**

   **Figure 6.4** - The StaticBinder utility binds the SQL statements in a pureQueryXml file into packages in the database.

4. **Run:** Specify your pureQueryXml file to pureQuery and run your application. pureQuery compares every SQL statement that the application executes to the statements that are in the file. You can set properties to indicate how pureQuery should treat the SQL statements that the application executes. Figure 9.5, Figure 9.6, Figure 9.7, and Figure 9.8 show some of the ways in which you can direct pureQuery to use the SQL statements in the pureQueryXml file. Some of the figures consider whether an SQL statement is a candidate for static execution. This is determined by the values of the `isBindable` attributes in the pureQueryXml file. Section 9.6.2.1 discusses this attribute.

   **pureQuery Properties:**
   - `pdq.executionMode=STATIC`
   - `pdq.allowDynamicSQL=FALSE`  

4. **Run: Static mode, do not allow dynamic SQL**

   **Figure 6.5** - You can direct pureQuery to execute all SQL statements statically. pureQuery only executes SQL statements that are in the pureQueryXml file and that are marked as candidates for static execution.
Chapter 6 – The Client Optimizer: pureQuery for Existing Applications 111

pureQuery Properties:
pdq.executionMode=STATIC
pdq.allowDynamicSQL=TRUE
pdq.capturedOnly=TRUE

Run: Static mode, allow dynamic statements, captured-only mode

Figure 6.6 - You can direct pureQuery to execute only SQL statements that are in the capture file. You can specify that it execute statements marked as candidates for static execution statically, that it execute the remaining statements dynamically.

pureQuery Properties:
pdq.executionMode=STATIC
pdq.allowDynamicSQL=TRUE
pdq.capturedOnly=FALSE

Run: Static mode, allow dynamic SQL, allow statements not in the pureQueryXml file

Figure 6.7 - You can direct pureQuery to execute statements marked as candidates for static execution statically, and that it execute all other statements dynamically.
6.3 JDBC driver requirements for using Client Optimizer functionality

In order to use the Client Optimizer, make sure that pureQuery supports the JDBC driver that you are using, and that its version meets the requirements of the pureQuery release.

If you are using the IBM Data Server Driver for JDBC and SQLJ as your JDBC driver, all that you need to do is have the pureQuery JAR files `pdq.jar` and `pdqmgmt.jar` in the classpath of the JDBC driver when the application creates a `javax.sql.DataSource` or `java.sql.Connection` object.

If you are using an Oracle database, your application must use a `javax.sql.DataSource` object to create Connection objects. Your `DataSource` object will need to be an instance of one of several `DataSource` classes that pureQuery provides. Each of these classes wraps or extends a `DataSource` object that an Oracle JDBC driver provides. Refer to the IBM Optim pureQuery Runtime documentation for information on how to use the Client Optimizer for Oracle databases.

6.4 pureQuery properties

pureQuery has a number of properties, called `pureQuery properties`. Several of these properties control the behavior of the Client Optimizer. The names of all of the pureQuery properties start with “pdq.”. For example, there is a property named `pdq.captureMode`. 
There are several ways to specify pureQuery properties. The most common ways are in a properties file and as the value of the IBM Data Server Driver for JDBC and SQLJ property pdqProperties. For each property, pureQuery looks for that property in several locations, in a particular order. pureQuery uses the first value that it finds. If pureQuery does not find the property, pureQuery uses the default value of that property. The following lists describe the standard approaches for specifying pureQuery properties.

**Connection through a DataSource object:** If the application creates a connection to the database by using a javax.sql.DataSource object, pureQuery searches for pureQuery properties in these locations, in the order in which they are listed.

1. A file named pdq.dataSourceName.properties in your application classpath, where dataSourceName represents the name of the DataSource object
2. A file named pdq.appwide.properties in your application classpath
3. The value of the property pdqProperties in the DataSource object. The implementations of javax.sql.DataSource that the IBM Data Server Driver for JDBC and SQLJ provides, as well as those that pureQuery provides, have a method setPdqProperties(String) that you can use for setting this. When you specify pureQuery properties in this way, you do not need to include the "pdq." in the property names.
4. A file named pdq.properties in the classpath that indicates the location of the JDBC driver.

**Connection through the DriverManager class:** If the application creates a connection to the database by using the java.sql.DriverManager class, pureQuery searches pureQuery properties in these locations, in the order in which they are listed.

1. The value of the property pdqProperties in the URL. When you specify pureQuery properties in this way, you do not need to include the "pdq." in the property names.
2. A java.util.Properties object that is passed as a parameter to a java.sql.DriverManager.getConnection method. When you specify pureQuery properties in this way, you do not need to include the "pdq." in the property names.
3. A file named pdq.properties in the classpath.

If you use a properties file to specify pureQuery properties, each line of the file should contain the name of a property, then an equal sign (=), and finally the value for that property. If you want a line in a properties file to be a comment that pureQuery ignores, make # the first character of the line. The backslash character (\) is an escape character, so if you want to use a backslash in the value of a property, you need to specify two backslashes (\\). Listing 6.1 provides an example of a properties file that specifies the value C:\path\captureFile.pdqxml for the property pdq.pureQueryXml, the value
ON for the property pdq.captureMode, and the value DYNAMIC for the property pdq.executionMode.

# This is a comment.
pdq.pureQueryXml=C:\path\captureFile.pdqxml
pdq.captureMode=ON
pdq.executionMode=DYNAMIC

**Listing 6.1 - An example of specifying pureQuery properties in a properties file**

As will be described later, you must specify pureQuery properties to use Client Optimizer functionality. The pureQuery properties must be specified when your application creates java.sql.Connection objects. If they are not, the application uses standard JDBC functionality.

Refer to the IBM Optim pureQuery Runtime documentation for a complete list of the pureQuery properties, along with information about their use.

### 6.5 Capturing SQL statements

The first step in using the Client Optimizer is to capture all of the SQL statements in the application into a pureQueryXml file. You tell pureQuery to capture SQL statements by setting certain properties in the pureQuery properties. pureQuery captures each SQL statement only once. pureQuery considers two SQL statements to be the same if the text of the SQL is identical (including capitalization and white space) and they have identical execution characteristics. Additionally, pureQuery only captures SQL statements that a JDBC application executes successfully. Therefore, if an application prepares a statement without executing it, or if an application attempts to execute an SQL statement but the execution fails, pureQuery will not capture that statement.

If you want pureQuery to capture all of the SQL statements that your application executes, clearly your application needs to successfully execute all of the statements while you are performing the capture. Usually, the easiest way to do this is to direct pureQuery to capture during the execution of the tests of your application. You can capture SQL statements incrementally; that is, if you capture SQL statements into a pureQueryXml file, you can later capture again, specifying the same pureQueryXml file. If you do this, pureQuery simply adds the newly captured SQL statements to your existing pureQueryXml file. You can even capture during static execution, if you set the pureQuery properties correctly.

If you capture SQL statements to create a pureQueryXml file, and then you modify your application, IBM recommends that you delete your pureQueryXml file and capture again to replace it. Otherwise, your pureQueryXml file can become bloated with information about SQL statements that your application no-longer executes, as well as inaccurate information about which parts of your application execute certain SQL statements.
6.5.1 pureQuery properties during capture

You must use pureQuery properties to direct pureQuery to capture. Four properties are particularly important in determining whether pureQuery captures.

- **pdq.captureMode** -- This is the main property that determines whether pureQuery captures SQL statements in the application. The possible values for this property are **ON** and **OFF**. **ON** directs pureQuery to capture. The default value is **OFF**.

- **pdq.pureQueryXml** -- This property specifies the absolute or relative path of the capture file. The extension of the file must be either `.pdqxml` or `.xml`. If the file specified by this property already exists, pureQuery simply adds new statements to this file. If you are using a pureQuery-enabled project in IBM Optim Development Studio, you should specify a path such that the file is located in the directory `pureQueryFolder`. The tool helps to manage `pureQueryXml` files in that folder.

- **pdq.allowDynamicSQL, pdq.capturedOnly** -- These properties are not required for capture. However, their values can prevent pureQuery from capturing. When you are capturing SQL statements for an application, the property **pdq.capturedOnly** must be either not set, or set to the value **FALSE**. If the property **pdq.executionMode** is set to the value **STATIC**, then the property **pdq.allowDynamicSQL** must be either not set or set to the value **TRUE**.

You can use several other pureQuery properties to control the capture behavior. These include `pdq.maxNonParmSQL, pdq.captureStatementBatchSQL, pdq.sqlLiteralSubstitution, pdq.outputPureQueryXml, pdq.maxStackTracesCaptured, pdq.stackTraceDepth, pdq.packagePrefixExclusions, pdq.traceFile`, and **pdq.traceLevel**. Refer to the IBM Optim pureQuery Runtime documentation for details about the features provided by these properties.

6.5.2 In practice: Capture SQL statements in the Human Resources Manager application

The **GettingStartedWithPureQuery** project contains the “Human Resources Manager” sample application. Chapter 4 describes this application. The class `hrm.inProgress.jdbc.HumanResourcesManager` implements the program as a JDBC application. Locate this class. We will perform capture on this application. Locate the test application `test.TestHumanResourcesManager`. This application runs the Human Resources Manager application to execute all of its SQL statements. Open the test file and verify that it imports the class `hrm.inProgress.jdbc.HumanResourcesManager`. Then, you are ready to capture.
6.5.2.1 Setup the pureQuery properties
We will specify our pureQuery properties in a file named `pdq.dataSourceName.properties`. The class `hrm.HRMDataSourceFactory` defines the `javax.sql.DataSource` object that Human Resources Manager uses. The name of the `DataSource` that the application uses is `DataSourceHumanResourcesManager`. Therefore, the name of our properties file is `pdq.DataSourceHumanResourcesManager.properties`. Locate this file in the `src` directory of the project, and change it so that it contains the required properties. Listing 6.2 shows the new contents of the file. Modify the path of the `pureQueryXml` file so that the path provides an appropriate location on your computer. In the listing, the line for the `pdq.pureQueryXml` property wraps onto two lines. This is only so that it will fit on the page; do not include any line breaks in the value of the property.

```plaintext
pdq.captureMode=ON
pdq.pureQueryXml=C:\pathOfWorkspace\GettingStartedWithPureQuery\pureQueryFolder\captureFile.pdqxml
```

Listing 6.2 - The properties to set in your properties file

6.5.2.2 Run the application
Your application now has the pureQuery properties set correctly, and it has `pdq.jar` and `pdqmgmt.jar` in the classpath. When we run the application, pureQuery should capture the SQL statements that the application executes. Run `test.TestHumanResourcesManager` as a Java application. When it has finished, refresh your project and look for the capture file that you specified in your properties. An error will probably appear because we have not yet specified a root package name for the new `pureQueryXml` file. We will do that in the next section.

6.6 Processing the pureQueryXml file with the Configure utility
After you have used pureQuery to capture a `pureQueryXml` file, you must configure it with the `pureQuery Configure` utility. The Configure utility modifies the specifications of the SQL statements so that they can be bound in the database. The utility also performs checks to ensure that certain aspects of the file are valid. You can customize how the utility configures a `pureQueryXml` file by specifying options. Anytime you create or change a `pureQueryXml` file, it is very important that you process it with the Configure utility before you bind it or use it during execution. If you do not, the statement declarations in the file could be in an invalid state that could cause unexpected results. If you are using a `pureQuery-enabled` project in IBM Optim Development Studio, you must specify the options that the Configure utility should use, but IBM Optim Development Studio runs the Configure utility automatically. If you are not, you must run the Configure utility from the command line.
6.6.1 Specifying options for the Configure utility

pureQuery provides multiple utilities. All of these utilities allow you to specify options on the command line. For the Configure utility and for some of the other utilities, you can also specify the pureQueryXml files that the utility should process, along with the options it should use for each file, in a type of file that is known as an options file. Refer to Appendix D for a description of options files. pureQuery allows two types of options files: genProps and bindProps. The Configure utility uses genProps files.

An options file is a text file that lists interfaces and pureQueryXml files to be processed, and the options to apply to each interface or file. Each interface and pureQueryXml file must be listed on a separate line in the options file, followed by an equal sign (=), and then any options that pureQuery should use when it processes that item. Additionally, in each options file, you can provide default options on a line that starts with defaultOptions. pureQuery uses each option that you specify on the defaultOptions line for all the items, except for those items where the option is specified with a different value. Listing 6.3 shows an example of how you would provide default options and pureQueryXml files in an options file.

defaultOptions = <options to use as defaults>
C:\path1\captureFile1.pdqxml = <options for captureFile1.pdqxml>
C:\path2\captureFile2.pdqxml = <options for captureFile2.pdqxml>

Listing 6.3 - Example of listing default options and pureQueryXml files in an options file

Refer to the documentation for IBM Optim pureQuery Runtime to see the options that are available for the Configure utility.

6.6.1.1 Specifying option files on the command line

When you specify an options file to the Configure utility on the command line, the utility configures all of the pureQueryXml files that are in the options file.

6.6.1.2 Specifying option files in Optim Development Studio

pureQuery-enabled projects in IBM Optim Development Studio contain a folder named pureQueryFolder. Optim Development Studio automatically runs the Configure utility for the pureQueryXml files that are in pureQueryFolder. Therefore, it is important that you place the pureQueryXml files that your application uses in pureQueryFolder. The folder also contains two options files: one named Default.genProps, and one named Default.bindProps. Notice the file Default.genProps in Figure 6.9. IBM Optim Development Studio uses this options file to provide the options that it uses when it runs the Configure utility.
When you save changes to Default.genProps, a dialogue like the one in Figure 6.10 might pop up with a warning message that indicates that the options you are specifying for interfaces might have changed, and that you might need to rebuild your project for its implementation classes to reflect your changes. The dialogue will ask if you want to rebuild the project for the changes to take affect. This occurs because Default.genProps files also specify options for generating implementation classes of interfaces in the pureQuery annotated method style (Section 6.3). When you do not make changes to any of the options that affect interfaces, you can select No on the dialogue.

Figure 6.10 - When you save changes to Default.bindProps, a dialogue asks if you want to rebuild your project to reflect any changes.

**6.6.2 Statement sets in pureQueryXml files**

The Configure utility groups the SQL statements into *statement sets*. In a pureQueryXml file, *statementSet* elements represent the statement sets. When the StaticBinder utility binds a pureQueryXml file against IBM DB2 databases, packages in the database correspond to statement sets in the pureQueryXml file. Because packages must have unique names in the database, the Configure utility requires that you specify a *root package name* that it will use to create the names of the packages. You specify the root package name to the utility by using the option `-rootPkgName`. 
The utility uses a limit known as the **SQL limit** to determine an approximate upper limit on the number of statements in each statement set. The default SQL limit is 100; you can change the limit by specifying the option `-sqlLimit` to the Configure utility.

When the Configure utility splits a statement set into multiple statement sets, the utility appends letters (A, B, C, and so forth) to the root package name to get distinct names for the statement sets. The **base package name** of a statement set is the root package name along with any letters the Configure utility appended. In the pureQueryXml, this is the value of the `name` attribute of the `package` element of a statement set.

### 6.6.2.1 How statement sets dictate what the StaticBinder utility binds in the database

In a pureQueryXml file, statement sets have `isBindable` attributes that indicate which SQL statements the StaticBinder utility should bind. The value `true` indicates that the utility should bind the item with the attribute, and `false` indicates that it should not. Each statement set has two types of `isBindable` attributes. These are their descriptions:

- **Statement-set level `isBindable` attributes**: Each of these attributes refers to a statement set, and it indicates whether any packages should be bound in the database for the statement set. The `statementSet` elements contain package elements. The statement-set level `isBindable` attributes are in the `package` elements.

- **Statement-level `isBindable` attributes**: Each of these attributes refers to a single SQL statement, and it indicates whether the SQL statement should be included in the package. The `statementSet` elements contain `statement` elements, each of which represents an SQL statement. The `statement` elements contain `statementAttributes` elements, which have `isBindable` attributes.

When the StaticBinder utility binds from a pureQueryXml file, it binds the statement sets for which the statement-set level `isBindable` attributes are set to true. Within each of these statement sets, it binds the statements for which the statement-level `isBindable` attribute is set to true.

You can change the `isBindable` attributes for specific statement sets and SQL statements. IBM recommends that you do this only by using the pureQueryXML editor that IBM Optim Development Studio provides. If you edit the file directly, you must set the `configureStatus` element to `REQUIRED` for each statement set in which you modify a statement-set level or statement-level attribute.

There are some SQL statements that certain database versions cannot bind, or that can cause problems when certain database versions attempt to bind them. When the Client Optimizer captures SQL statements that it considers potentially problematic, it sets their `isBindable` attributes to `false`. 
6.6.2.2 Changes to statement sets

When you configure a pureQueryXml file that has previously been configured, the default behavior of the Configure utility is to attempt to preserve how statements are grouped into statement sets by processing only the statement sets that need to be processed. You can specify the option `--cleanConfigure` with the value `TRUE` to direct pureQuery to process all of the statements without attempting to preserve their groupings.

In a pureQueryXml file, each `statementSet` element has a `configureStatus` attribute that is set to `REQUIRED`, `AUTO`, or `FINAL`. These values dictate how the Configure utility treats the statement sets. The utility does not configure a statement set for which the configure status is `FINAL`, if the statement set has a base package name. The utility only configures `AUTO` packages when necessary, and it always configures `REQUIRED` packages. When you specify the option `--cleanConfigure` with the value `TRUE` to the Configure utility, the utility makes no attempt to preserve the groupings of the statements in the file. It configures all of the SQL statements and creates new statement sets to contain them.

If you manually edit a statement set, you must change the `configureStatus` to `REQUIRED`. Otherwise, the Configure utility might not configure the statement set, and its resulting contents could be invalid. When the Configure utility configures a statement set, it performs some checks on the contents of the statement set, and it stores information in the statement set that the StaticBinder utility needs to bind the packages. Changes to the contents of a statement set can result in changes to the information necessary to bind the package correctly. IBM recommends that you only modify pureQueryXml files by capturing, by configuring, and by using the IBM Optim Development Studio pureQueryXML editor. Doing so causes the `configureStatus` attribute to be managed appropriately, and it otherwise helps to keep the contents of the file in a legal state.

6.6.3 DDL statements

DDL statements can cause problems in applications that execute SQL statically. When an application runs DDL statements that modify tables or other database objects, the IBM® DB2 Database automatically invalidates any packages that reference those database objects. If the application then attempts to execute an SQL statement in an invalidated package, the database automatically tries to re-bind the package. The access plans that the re-bind uses can be different from those of the original package, and they might be less optimal than the original plans. Consequently, if your application runs DDL statements, you should review the SQL statements to determine if they will cause problems for the database packages that your application will use.

The Configure utility puts DDL statements and non-DDL statements into separate statement sets. It also generates a separate file with the extension `.ddl` that contains the DDL statements that are in the file. This file can help you to determine which DDL statements your application executes, and you can use it to know the SQL statements that
create the database objects that need to exist to bind the DML SQL statements in your pureQueryXml file. Be aware that if your application drops these database objects, the database will invalidate the packages in the database.

By default, the isBindable attributes are set to false for DDL statements. However, you can specify the option -markDDLForBind to the Configure utility to change the value of the attribute. The Configure utility sets the attributes for DDL statements to the value that you specify for this option. You might want to bind packages of DDL statements if you want to give users the privileges necessary to execute certain packages of DDL statements, without allowing them to execute all DDL statements.

6.6.4 Files with the .org extension

When you run the Configure utility for a pureQueryXml file, it creates a file with the extension .org that is a copy of the original file. You do not need to do anything with this file. It allows you to refer back to the old version of the file, if you need to. For example, if you specify the wrong options, you might want to start over with the original version of the file. If another file with the same name already exists when the Configure utility creates a file with the .org extension, the utility overwrites that file.

6.6.5 Running the Configure utility

If you are using IBM Optim Development Studio, the tool runs the Configure utility automatically for the pureQueryXml files in the folder pureQueryFolder, using the options that you specify in Default.genProps. Therefore, if you are using this tool, all you have to do is to put an entry in Default.genProps for your pureQueryXml file in which you specify -rootPkgName, the option that the Configure utility requires. If an error occurs, IBM Optim Development Studio displays the error message on the console.

If you are not using IBM Optim Development Studio, you need to run the Configure utility from the command line. To do so, you must have pdq.jar and pdqmgmt.jar in your classpath. Listing 6.4 shows the command.

```java
java com.ibm.pdq.tools.Configure
```

Listing 6.4 - The command for running the Configure utility on the command line

As you know, you must specify the -rootPkgName option when you run the command. Additionally, you must specify the file or files that the utility must process. If you only need it to process one file, you can specify the file name with the option -pureQueryXml. Alternatively, you can use the option -optionsFile to specify an options file that lists one or more files that you want pureQuery to process. You can optionally specify other options to customize the behavior of the Configure utility. Specify the option -help to see the available options.
6.6.6 In practice: Configure the pureQueryXml file that was captured

Let's now configure the pureQueryXml file that we captured in Section 6.5.2. We start by adding the necessary options to the file in the project. The only option that we will specify is `-rootPkgName`, with the value `HRMCO` (for “Human Resources Manager,” and “Client Optimizer”). Locate the `Default.genProps` file in the directory `pureQueryFolder` and add the line shown in Listing 6.5. Make sure to give the correct path for the pureQueryXml file, and to put the entire entry on a single line.

```
C:\pathOfWorkspace\GettingStartedWithPureQuery\pureQueryFolder\captureFile.pdqxml=-rootPkgName HRMCO
```

Listing 6.5 - A line in `Default.genProps` that specifies the root package name for the pureQueryXml file

If you are using IBM Optim Development Studio, you have done all that you need to do; the tool will run the Configure utility automatically. If the error that occurred before you specified the root package name is still on the console, clear the console. If it appears again, check the contents of `Default.genProps`. You might not have added the root package name correctly.

If you are not using IBM Optim Development Studio, you need to run the Configure utility on the command line. On the command line, specify the command in Listing 6.6. Make sure that `pdq.jar` and `pdqmgmt.jar` are in the classpath, and that you provide the `correct path` for `Default.genProps`.

```
java com.ibm.pdq.tools.Configure -optionsFile "C:\pathOfWorkspace\GettingStartedWithPureQuery\pureQueryFolder\Default.genProps"
```

Listing 6.6 - The command for configuring our pureQueryXml file from the command line

6.7 Binding the SQL statements in the pureQueryXml file

If you want to use static SQL, you must use the pureQuery StaticBinder utility to bind the SQL statements in your pureQueryXml file. The StaticBinder utility binds packages in the database that correspond to the statement sets in the pureQueryXml file. As described in Section 9.6.2.1, the utility only binds packages and statement sets for which the `isBindable` attributes are set to the value `true`. You can run the StaticBinder utility either from the command line or by selecting menu options in IBM Optim Development Studio. Refer to Chapter 7 for a description of the StaticBinder utility and how to specify what packages it should bind in the database.

6.7.1 In practice: Bind the SQL statements in your pureQueryXml file

If you are using IBM® DB2® Database for your database, use the StaticBinder to bind the statements in your pureQueryXml file in the database.
6.7.1.1 Specify options for the pureQueryXml file

As for the Configure utility, you can use an options file to specify options for the StaticBinder utility. Notice the options file named Default.bindProps in Figure 6.11. It is located in the pureQueryFolder directory in the GettingStartedWithPureQuery project.

![Figure 6.11 - The options file Default.bindProps in a pureQuery-enabled project](image)

Add a line to Default.bindProps for the pureQueryXml file. The StaticBinder’s defaults are fine for our purposes, so do not specify any options on the line. The StaticBinder can bind the file even without the line, but it is a good practice to add it. Adding everything that must be bound to the options file makes it easy to bind the application from the command line. Add a line as shown in Listing 6.7. Remember to change the path so that it is correct for your pureQueryXml file. If you were to add options for the pureQueryXml file, you would specify them after the equal sign (=).

```
C:\pathOfWorkspace\GettingStartedWithPureQuery\pureQueryFolder\captureFile.pdqxml
```

Listing 6.7 - An entry for our pureQueryXml file in Default.bindProps

6.7.1.2 Run the StaticBinder utility.

If you are using IBM Optim Development Studio, bind the pureQueryXml file by right-clicking on it in the Package Explorer view and selecting pureQuery and then selecting Bind…. This is shown in Figure 6.12. Then, choose the connection that you want to use for the bind and click Finish, as shown in Figure 6.13.
Figure 6.12 - Right-click on the pureQueryXml file and select pureQuery, and then select Bind...
If you are not using IBM Optim Development Studio, run the StaticBinder from the command-line by putting pdq.jar, pdqmgt.jar, and the JAR files for the IBM Data Server Driver for JDBC and SQLJ in the classpath and specifying the command in Listing 6.8. This command binds captureFile.pdqxml using any options that Default.bindProps provides. Since we did not specify any options for the file or for defaultOptions in Default.bindProps, the StaticBinder utility will just use the default settings. In the command, you will need to change the URL, username, and password so that they are correct for your database. You will also need to change the path of the options file and the pureQueryXml file so they are correct for your files.

```java
java com.ibm.pdq.tools.StaticBinder -url yourURL -username yourUsername -password yourPassword -optionsFile "C:\pathOfWorkspace\GettingStartedWithPureQuery\pureQueryFolder\Default.bindProps"
```
6.8 Running an application while specifying a pureQueryXml file

When you have a pureQueryXml file that you have configured and possibly used for a bind, you can direct pureQuery to use it while your application is executing. Use the pureQuery property `pdq.pureQueryXml` to specify the path of the pureQueryXml file that pureQuery should use during execution. The path can indicate the location of the file on the disk, or it can indicate the location of the file in an archive file that is in the classpath. Use pureQuery properties to indicate what you want pureQuery to do during execution.

6.8.1 Static execution

Possibly the most common reason for specifying a pureQueryXml file during execution is to have SQL statements execute statically. You can set the value of the pureQuery property `pdq.executionMode` to either `STATIC` or `DYNAMIC`. Set it to the value `STATIC` to direct pureQuery to execute statically the statements in the pureQueryXml file that have both statement-set level and statement-level `isBindable` attributes set to the value `true`.

You can set the property `pdq.allowDynamicSQL` to dictate what pureQuery does when the application attempts to execute statements that are not in the pureQueryXml file. This property also determines what pureQuery does when the application attempts to execute statements that are in the pureQueryXml file with an `isBindable` attribute set to `false`. The value `TRUE` is the default, and it directs pureQuery to execute the statements dynamically. The value `FALSE` directs pureQuery to throw exceptions instead of executing the statements. Some SQL statements cannot be bound. If your application needs to execute any such statements, you must either set the property `pdq.allowDynamicSQL` to the value `TRUE` or you must leave the property unset.

6.8.2 Captured-only execution

You can use the captured-only feature to restrict the SQL statements that your application can execute, regardless of whether they are bound in the database. The SQL statements in your pureQueryXml file provide a whitelist of SQL statements that you allow, and pureQuery blocks the execution of all other SQL statements. This allows you to improve the security of applications that contain SQL statements that are not bound. It also helps you to improve the security of your application in environments where you are not using static SQL, such as in an environment in which you are using a database other than the IBM DB2 Database. Set the pureQuery property `pdq.capturedOnly` to the value `TRUE` to use this feature. The default value is `FALSE`.
6.8.3 Replacing SQL statements with alternate statements

Section 9.9 describes how to specify a pureQueryXml file during execution if you want to use it to replace SQL statements defined in the application.

6.8.4 SQL literal substitution

Section 9.10 describes how to specify a pureQueryXml file during execution to convert SQL statements that contain literals into parameterized SQL statements.

6.8.5 Capturing during execution

You can direct pureQuery to capture new statements while using a pureQueryXml file for execution. To use this feature, set `pdq.captureMode` to the value `ON`. You must also allow pureQuery to execute the new SQL statements; that is, you cannot use captured-only execution, and you cannot disallow dynamic SQL.

6.8.6 In practice: Run the Human Resources Manager application while specifying the pureQueryXml file

Modify properties in the file `pdq.DataSourceHumanResourcesManager.properties` to use either static execution or captured-only execution.

If you want to use static execution, you must have used the StaticBinder to bind the SQL statements. Set the properties as they are in Listing 6.9. Be sure to set the path of the pureQueryXml file so that it is correct for your file. Observe that these settings also set `pdq.allowDynamicSQL` to `FALSE`.

```
pdq.captureMode=OFF
pdq.pureQueryXml=C:\pathOfWorkspace\GettingStartedWithPureQuery\pureQueryFolder\captureFile.pdqxml
pdq.executionMode=STATIC
pdq.allowDynamicSQL=FALSE
```

Listing 6.9 - The properties to set in your properties file for static execution

If you want to use captured-only execution, set the properties as they are in Listing 6.10.

```
pdq.captureMode=OFF
pdq.pureQueryXml=C:\pathOfWorkspace\GettingStartedWithPureQuery\pureQueryFolder\captureFile.pdqxml
pdq.capturedOnly=TRUE
```

Listing 6.10 - The properties to set in your properties file for captured-only execution

6.9 Specifying alternate SQL statements for execution

You can use the Client Optimizer to replace SQL statements in your application, without modifying the application itself. A replacement statement must have the same metadata and parameter markers as the original statement.
If you are using IBM Optim Development Studio, you can add a replacement SQL statement by opening a pureQueryXml file with the tool’s pureQueryXML Editor. Right-click on the statement that you want to replace and select Edit Statement, as shown in Figure 6.14. Then, enter the new statement that you want to use, as shown in Figure 6.15.

![Figure 6.14 - Edit a statement in a pureQueryXml file by using the pureQueryXML Editor](image)

![Figure 6.15 - In the pureQueryXML Editor, type the replacement SQL statement that you want to use](image)

If you are not using IBM Optim Development Studio and you want to use this feature, you will need to modify the pureQueryXml file directly. In a pureQueryXml file, statementDescriptor elements describe the SQL statements that the application executes. To replace an SQL statement, locate the statementDescriptor element for that statement, and set the new SQL statement as the value of the processedSql element. After you make changes to statement sets, for use in static execution, you must set the value of their configureStatus attributes to REQUIRED.

If you specify to the StaticBinder utility a pureQueryXml file that contains replacement statements, and the isBindable attributes are set to true for the statements, the utility binds the alternate statements instead of the original statements. If you are using dynamic SQL, pureQuery uses the original SQL statements by default. Use the pureQuery property pdq.enableDynamicSQLReplacement to direct pureQuery to use the replacement SQL statements. The value TRUE directs pureQuery to use the replacement SQL statements; the value FALSE directs pureQuery to use the original SQL statements. The default value is FALSE.

The ability to specify replacement SQL statements is a very powerful feature that can help you to replace SQL statements in situations in which you cannot modify the source code. Be aware, however, that using this feature can reduce the security of your application, if you do not sufficiently protect the pureQueryXml file. Someone could maliciously add replacement SQL statements to a pureQueryXml file to cause the application to execute them instead of the SQL statements in the application. For SQL statements that execute
dynamically, replacing SQL statements is not allowed by default, and you must set 
`pq.enableDynamicSQLReplacement` to `TRUE` to allow it. For SQL statements that 
execute statically, if you cannot sufficiently control access to your `pureQueryXml`, you can 
check your `pureQueryXml` file before each bind to ensure that it does not contain 
replacement statements.

### 6.10 SQL Literal substitution

Sometimes applications execute SQL statements that contain literal values instead of 
parameter markers. For example, if an application searches the `EMPLOYEE` table for an 
employee with an employee number that the user supplies, the application might use string 
concatenation to create an SQL statement with a literal value, as shown in Listing 6.11. In 
this example, if the user supplied the employee number `000010`, then the application 
would create the SQL statement shown in Listing 6.12.

```java
String employeeNumber = // value supplied by user
String sql = "SELECT * FROM EMPLOYEE WHERE EMPNO = '" + employeeNumber + 
"';
PreparedStatement preparedStatement = connection.prepareStatement (sql);
ResultSet resultSet = preparedStatement.executeQuery();
```

Listing 6.11 - An example of code that creates and executes an SQL statement with a 
literal value

### Listing 6.12 - The SQL statement that would be created by the code in Listing 6.11 if 
the user provided the employee number "000010"

```sql
SELECT * FROM EMPLOYEE WHERE EMPNO = '000010'
```

Listing 6.13 shows an example that creates an SQL statement with a parameter, rather 
than one with a literal value.

```java
String employeeNumber = // value supplied by user
String sql = "SELECT * FROM EMPLOYEE WHERE EMPNO = ?";
PreparedStatement preparedStatement = connection.prepareStatement (sql);
preparedStatement.setString (employeeNumber);
ResultSet resultSet = preparedStatement.executeQuery();
```

Listing 6.13 - An example of code that creates and executes an SQL statement with a 
parameter

### 6.10.1 Advantages of parameterized SQL statements

SQL statements that have parameters have many advantages over those that have literal 
values. For example, SQL statements that are constructed by concatenation are 
vulnerable to SQL injection. In addition, if the user supplies the literal values, then there 
can be an enormous number of very similar SQL statements. The large number of distinct 
statements can have negative performance impacts, such as by flooding the dynamic 
statement cache with statements that are executed only once, and thereby significantly
reducing the cache-hit rate. Having a large number of distinct statements that are
generated during runtime might also prevent you from being able to have your application
run all SQL statements statically, since it might not be feasible to capture every statement
that could execute.

6.10.2 How to use pureQuery SQL literal substitution

You can use the Client Optimizer to replace the literal values in SQL statements with
parameter markers, thereby potentially improving the security and performance of your
application. This is easy to do by setting a property, and it does not require changes to
your application.

To use the literal substitution feature for static or captured-only execution, you must set the
pureQuery property `pdq.sqlLiteralSubstitution`. This property can have one of
three values: `ENABLE`, `DISABLE`, and `NOT_SET`. `NOT_SET` is the default value. The
effects of the property depend on whether you are capturing to create a pureQueryXml file
or executing while specifying a pureQueryXml file. In general, use the value `ENABLE` if you
want to use literal substitution and the value `DISABLE` if you do not. The lists below
describe the specific meanings of the different values.

These are the effects of the different values during capture:

- **ENABLE**: pureQuery substitutes literal values with parameter markers in
  as many of the statements that it captures as possible. pureQuery then
  executes the parameterized versions of these statements instead of the
  original versions. Consequently, the database and the JDBC driver are
  only aware of their parameterized forms.

- **DISABLE**: pureQuery does not replace literal values; it captures SQL
  statements how they are executed by the application. The maximum
  number of SQL statements containing literals that pureQuery captures
  can be limited by the value of the pureQuery property
  `pdq.maxNonParmSQL`, if that property is set.

- **NOT_SET**: If the pureQueryXml file does not already exist, pureQuery
  does not replace literal values. If the pureQueryXml already exists (which
  means this is an incremental capture), pureQuery uses the value that was
  set during the previous capture.

These are the effects of the different values when you specify a pureQueryXml file during
static or dynamic execution:

- **ENABLE**: pureQuery substitutes literal values with parameter markers in
  the SQL statements that execute, and tries to match the statements to the
  parameterized statements in the file. If pureQuery cannot find a matching
  parameterized statement, pureQuery tries to match the original statement
to the statements in the file that contain literals. In some circumstances, pureQuery also performs literal substitution on SQL statements that are not included in the pureQueryXml file. The details of those circumstances are beyond the scope of this book.

- **DISABLE**: pureQuery tries to match the SQL statements that execute with literals to statements in the file that contain literals. pureQuery does not convert statements with literals into parameterized statements.

- **NOT_SET**: pureQuery uses the value that was set during the previous capture.

Note that if you set the property to **ENABLE** during capture, you should set it to **ENABLE** or **NOT_SET** during execution. Similarly, if you set it to **DISABLE** during capture, you should set it to **DISABLE** or **NOT_SET** during execution. Otherwise, pureQuery might not find SQL statements that are in the file.

pureQuery does not perform literal substitution for all SQL statements that contain literal values. For example, if a statement contains both a parameter marker and a literal, like the statement in Listing 6.14, pureQuery assumes that the presence of the literal is intentional and does not remove it.

```
SELECT * FROM EMPLOYEE WHERE SALARY > ? AND BONUS > 1000
```

**Listing 6.14 - An SQL statement that contains both a parameter marker and a literal value**

Refer to the IBM Optim pureQuery Runtime documentation for more information about the literal substitution feature, including details about the situations in which pureQuery does not perform literal substitution.

### 6.11 Locating SQL statements in your application

In IBM Optim Development Studio, you can use the SQL Outline view to review SQL statements in your application and to determine where they are specified in your application. Even without using the Client Optimizer, you can use this view to locate many of the SQL statements. If you use the Client Optimizer to capture SQL statements in an application, the SQL Outline can display information about all of the SQL statements that it executed. Refer to Section 13.2 for more information about this feature.

You can also look in a pureQueryXml file that the Client Optimizer created during capture to determine where your application prepares and executes SQL statements. When pureQuery captures SQL statements, it records stack traces in the pureQueryXml file to indicate which parts of the application prepared the statement, and which parts executed it. Refer to the IBM Optim pureQuery Runtime documentation for more information about the stack traces that are captured.
6.12 Aspects of JDBC application that may cause problems during static execution

If your application contains code that follows certain coding processes, you might not be able to execute it statically, or you might need to modify your application to be able to execute it statically. The following list describes several of these situations and coding practices.

- **Using an UPDATE or DELETE statement containing a WHERE CURRENT OF clause to modify a java.sql.ResultSet object created by a SELECT statement, while another ResultSet object is open for an identical SELECT statement:** During static execution, pureQuery cannot always determine which of the ResultSet objects the UPDATE or DELETE statement should modify.

- **Running DDL statements to modify tables that are accessed in the bound database packages:** When your application runs these DDL statements, the database automatically invalidates the packages that reference the tables.

- **Using SET statements to set special registers:** Several SET statements that set special registers can cause problems when run statically. You can use the StaticBinder utility option -bindOptions to set them, instead.

- **VALUES statements:** The VALUES statements that IBM DB2 Database for Linux, UNIX, and Windows supports when running statically are different from those it supports when running dynamically.

- **XQUERY statements:** IBM DB2 Database for Linux, UNIX, and Windows only supports XQUERY statements during dynamic execution. It does not support these statements during static execution.

The reasons why these situations can cause problems are beyond the scope of this book. Refer to the IBM Optim pureQuery Runtime documentation for details about these situations and coding practices, and for information about how you can avoid them.

6.13 Advanced features

The Client Optimizer has many advanced features that are outside of the scope of this book. This list describes some of them. For details about using these features, refer to the IBM Optim pureQuery Runtime documentation.

- **Specifying pureQuery properties for applications, for data sources, and globally:** You can use pdq.appwide.properties files to specify pureQuery properties that are specific to applications, pdq.dataSourceName.properties files to specify pureQuery...
properties that apply to data sources, and `pdq.properties` files to specify pureQuery properties that apply globally.

- **Capturing in multiple pureQueryXml files:** You can use the pureQuery property `pdq.outputPureQueryXml` to specify the file in which pureQuery should capture new SQL statements. When you do this, the property `pdq.pureQueryXml` indicates the file that contains the SQL statements that have already been captured. When your application captures SQL statements into multiple pureQueryXml files, you then use the Merge utility to merge the files into a single file that you can use during execution.

- **Capturing SQL statements in clustered applications:** You can capture in applications that are vertically clustered, horizontally clustered, or both. This involves using the property `pdq.outputPureQueryXml` to capture multiple pureQueryXml files and then using the Merge utility to merge the files.

- **Finalizing statement sets:** You can use the `FINAL` value of the `configureStatus` attribute to direct the Configure utility not to modify a statement set.

- **Rowset cursors:** You can specify the option `-allowStaticRowsetCursors` to the Configure utility to have SQL statements use rowset cursors for multi-row fetch during static execution.

- **Managing stack traces:** pureQuery captures stack traces to enable you to determine which part of your application executed the SQL statements. You can set the pureQuery properties `pdq.maxStackTracesCaptured`, `pdq.packagePrefixExclusions`, and `pdq.stackTraceDepth` to configure how pureQuery captures stack traces.

- **Limits on statements that pureQuery captures:** If you have an application that executes SQL in large batches, or if you have an application that executes many SQL statements that it constructs at runtime, your capture file could become bloated with SQL statements that the application very rarely executes. You can use the pureQuery properties `pdq.maxNonParmSQL` and `pdq.captureStatementBatchSQL` to configure which of these statements pureQuery captures.

IBM Optim Development Studio also provides advanced features that you can use for the Client Optimizer. This list describes some of them. Refer to the IBM Optim Development Studio documentation for more information.

- **pureQueryXML editor:** You can use the pureQueryXML editor to modify the contents of pureQueryXml files and to merge pureQueryXml files together. You can easily perform a number of modifications on
pureQueryXml files, such as changing whether statements are candidates for static execution (that is, changing the value of the isBindable attribute). You can also modify statements, removing statements, and changing the groupings of statements. IBM recommends that you use the pureQueryXML editor, rather than editing pureQueryXml files manually. If you directly edit the contents of files, you could accidentally render the contents invalid.

- **Automatic capturing:** You can use menu options to direct IBM Optim Development Studio to create a pdq.properties file for a pureQuery-enabled project. The properties file that it creates causes pureQuery to capture.

### 6.14 Summary

The pureQuery Client Optimizer enables you to improve easily the security, performance, and maintainability of JDBC applications. You first direct pureQuery to capture SQL statements that your application executes into a pureQueryXml file. Next, you use the Configure utility to prepare the file so that you can bind the statements in it and so that pureQuery can refer to it while an application is executing. Then, if you want to use static execution, you specify the file to the StaticBinder utility to direct it to bind the SQL statements in the file. Finally, you direct pureQuery to use the file while your application executes. Using the Client Optimizer allows you to execute the SQL in your application statically. It also allows you to restrict the SQL statement that your application can execute, even without using static SQL. It provides several other benefits as well, such as a way to replace easily SQL statements in an application and the ability to improve the security and performance of your application by using SQL literal substitution.

### 6.15 Review questions

1. What is the name of the pureQuery feature that you can use to use static SQL with JDBC applications?

2. What is the name of the Client Optimizer stage in which pureQuery creates a pureQueryXml file that contains the SQL statements that your application executes?

3. What is the name of the utility that you use to process pureQueryXml files that you capture before you use them for a bind or for static or dynamic execution?
   - A. Generator
   - B. Configure
   - C. StaticBinder
4. In addition to specifying the pureQueryXml file to the Configure utility, which option must you always specify, and what does the utility do with the value of this option?

5. If you are using IBM Optim Development Studio, what is the name of the options file in which you must list the pureQueryXml files and the options that you want the Configure utility to use for them?
   A. Default.bindProps
   B. pdq.properties
   C. Default.genProps
   D. optionsFile.txt

6. What is the name of the utility that you must use to bind the SQL statements in a pureQueryXml file in a database?
   A. Generator
   B. Configure
   C. StaticBinder
   D. Merge

7. If you are using IBM Optim Development Studio, what is the name of the options file in which you can specify options for the StaticBinder utility to use when it binds pureQueryXml files?
   A. Default.bindProps
   B. pdq.properties
   C. Default.genProps
   D. optionsFile.txt

8. True or false: If you make any changes to a pureQueryXml file, such as by performing an incremental configure, you should process the file with the Configure utility before you use the file to bind or for static or dynamic execution.

9. Which of the following ways does IBM recommend that you use to modify pureQueryXml files?
   A. Create or add to the file by using pureQuery capture
   B. Process the file with the Configure utility
   C. Edit the file by using the IBM Optim Development Studio pureQueryXML editor
   D. Edit the file manually (that is, with any editor other than the pureQueryXML editor)
10. If you manually edit a statement set in a pureQueryXml file, what must you do before you process the files with the Configure utility?

11. For SQL statements that execute dynamically, what is the name of the pureQuery property that you can use to direct pureQuery to use the replacements SQL statements that you provide in `processedSql` elements in your pureQueryXml file?

12. If you want your application to use SQL literal substitution, what is the name of the pureQuery property that you must use?

13. What are some of the benefits of using the Client Optimizer?

### 6.16 Exercise

Temporarily replace the SQL statement in the `displayLogin()` method of `hrm.inProgress.jdbc.HumanResourcesManager` with another SQL statement that was not captured, such as "SELECT * FROM EMPLOYEE WHERE EMPNO = '000010' AND EMPNO <> ?". Run the application either with captured-only execution, or in static mode without allowing dynamic SQL and confirm that pureQuery throws an exception indicating that the SQL statement is not allowed to execute.
Chapter 7 - The StaticBinder Utility

In this chapter, you will learn how to use the StaticBinder utility to bind the SQL statements in interfaces and pureQueryXml files in your database. Once the StaticBinder utility has bound the SQL statements in your database, they can be executed statically. For a brief description of static SQL and its powerful benefits, see Appendix C

7.1 The StaticBinder utility: The big picture

If you want to use static SQL, you must use the StaticBinder utility to bind SQL statements in interfaces and pureQueryXml files in your database. If you are using IBM Optim Development Studio, you can select the interfaces and pureQueryXml files that contain the SQL statements that you want to bind. If you are running the StaticBinder utility from the command line, you can easily specify the interfaces on the command line or in an options file that you specify on the command line. From IBM Optim Development Studio and from the command line, you can also specify an archive file that contains the interfaces and pureQueryXml files that you want the utility to use for the bind.

Figure 7.1 illustrates how you use the StaticBinder utility to bind packages in the database.

Figure 7.1 - You can use the StaticBinder utility to bind packages in the database containing the SQL statements in pureQueryXml files or in interfaces of annotated methods
7.2 Binding from interfaces and pureQueryXml files

You must specify pureQueryXml files or interfaces that contain the SQL statements that
pureQuery should bind.

7.2.1 Binding from interfaces

7.2.1.1 Database packages that the StaticBinder utility binds

For static SQL, interfaces of annotated methods represent packages in the database. The
implementation class corresponding to an interface has a root package name. The
StaticBinder utility uses this to determine the names of the packages in the database. The
root package name was specified to the Generator utility by using the -rootPkgName
option, or the Generator utility determined it based on the name of the interface, if the
option was not specified. In the implementation class, the root package name is stored in
the field identifier.

By default, the StaticBinder utility binds four packages in the database for each interface,
one at each of four isolation levels. Also by default, the StaticBinder utility appends 1, 2, 3,
and 4 to the root package name of the interface so that the four packages have distinct
names. The number 1 indicates the isolation level uncommitted read (UR), 2 indicates
cursor stability (CS), 3 indicates read stability (RS), and 4 indicates repeatable read (RR).

If you want the StaticBinder utility to bind only one package for an interface, specify the
option -isolationLevel. If you only want it to bind one package, and you do not want
the 1, 2, 3, or 4 indicator appended to the root package name, specify the option -
forceSingleBindIsolation to the Generator utility when you generate the
implementation class for the interface. The Generator utility needs to store this information
in the implementation class so that during static execution, pureQuery knows the names of
the packages. When you bind the interface, specify -isolationLevel so that the
StaticBinder utility will know which isolation level to use.

As an example, suppose that you have an interface with the root package name MYPKG.
Table 7.1 shows some example of the packages that the utility would bind in the database
for certain values of -forceSingleBindIsolation and -isolationLevel. The table
only shows the packages when the option -isolationLevel is either not specified, or is
specified with the value UR. The results follow the same pattern when other values are
specified for -isolationLevel.

<table>
<thead>
<tr>
<th>Value specified for the -forceSingleBindIsolation option to the Generator utility</th>
<th>Value specified for the -isolationLevel option to the StaticBinder utility</th>
<th>Names of packages bound in the database</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7.1
Table 7.1 - For an interface, example of the packages that are bound when different options are specified

<table>
<thead>
<tr>
<th></th>
<th>Not specified or FALSE</th>
<th>Not specified</th>
<th>MYPKG1, MYPKG2, MYPKG3, MYPKG4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FS1</td>
<td>TRUE</td>
<td>UR</td>
<td>MYPKG1</td>
</tr>
<tr>
<td>FS2</td>
<td>TRUE</td>
<td>UR</td>
<td>MYPKG2</td>
</tr>
<tr>
<td>FS3</td>
<td>TRUE</td>
<td>UR</td>
<td>MYPKG3</td>
</tr>
<tr>
<td>FS4</td>
<td>TRUE</td>
<td>UR</td>
<td>MYPKG4</td>
</tr>
</tbody>
</table>

7.2.1.2 Specifying an interface

When you specify an interface to the StaticBinder utility, you must provide its fully-qualified name. Listing 7.1 shows an example of specifying an interface.

myPackage1.MyInterface1

Listing 7.1 - Example of specifying an interface

7.2.2 Binding from pureQueryXml files

7.2.2.1 Database packages that the StaticBinder utility binds

In a pureQueryXml file, SQL statements are grouped into statement sets. Each statement set is represented by a statementSet element in the file. When the StaticBinder utility binds a pureQueryXml file, packages in the database correspond to statement sets in the pureQueryXml file. Each statement set has a base package name. This is the value of the name attribute of the package element of the statement set, and the utility uses it to determine the names of the packages in the database.

The StaticBinder utility only binds packages for statement sets in which the statement-set level isBindable attribute is set to true. Within each statement set, the utility only binds the SQL statements for which the statement-level isBindable attribute is set to true. Refer to Section 9.6.2.1 for a description of the isBindable attributes.

By default, the StaticBinder utility binds four packages in the database for each statement set, one at each of four isolation levels. Also by default, the StaticBinder utility appends 1, 2, 3, and 4 to the base package names of the statement sets so that the four packages have distinct names. The number 1 indicates the isolation level uncommitted read (UR), 2 indicates cursor stability (CS), 3 indicates read stability (RS), and 4 indicates repeatable read (RR).

If you want the StaticBinder utility to bind packages at only one isolation level, specify the option -isolationLevel. If you only want one package to be bound, and you do not want the 1, 2, 3, or 4 indicator appended to the base package name, specify the option -forceSingleBindIsolation to the Configure utility when you configure the pureQueryXml file. The Configure utility needs to store this information in the pureQueryXml file so that during static execution, pureQuery knows the names of the
packages. When you bind the pureQueryXml file, specify the `-isolationLevel` so that the StaticBinder utility will know which isolation level to use. The Configure utility can process pureQueryXml files that pureQuery’s Client Optimizer functionality created, but it cannot process all pureQueryXml files.

As an example, suppose that you have a pureQueryXml file that contains two statement sets with the base package names `MYPKGA` and `MYPKGB`. Suppose that you first process this file with the Configure utility, and then you use the StaticBinder utility to bind the statements in your database. Table 7.2 shows an example of the packages that the utility would bind in the database for certain values of `-forceSingleBindIsolation` and `-isolationLevel`. The table only shows the packages when the option `-isolationLevel` is either not specified, or is specified with the value `UR`. The results follow the same pattern when other values are specified for `-isolationLevel`.

<table>
<thead>
<tr>
<th>Value specified for the <code>-forceSingleBindIsolation</code> option to the Configure utility</th>
<th>Value specified for the <code>-isolationLevel</code> option to the StaticBinder utility</th>
<th>Names of packages bound in the database</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not specified or <code>FALSE</code></td>
<td>Not specified</td>
<td><code>MYPKGA1</code>, <code>MYPKGA2</code>, <code>MYPKGA3</code>, <code>MYPKGA4</code>, <code>MYPKGB1</code>, <code>MYPKGB2</code>, <code>MYPKGB3</code>, <code>MYPKGB4</code></td>
</tr>
<tr>
<td>Not specified or <code>FALSE</code></td>
<td><code>UR</code></td>
<td><code>MYPKGA1</code>, <code>MYPKGB1</code></td>
</tr>
<tr>
<td><code>TRUE</code></td>
<td><code>UR</code></td>
<td><code>MYPKGA</code>, <code>MYPKGB</code></td>
</tr>
</tbody>
</table>

Table 7.2 - For a pureQueryXml file that contains two statement sets, example of the packages that are bound when different options are specified

### 7.2.2.2 Specifying a pureQueryXml file

Specify a pureQueryXml file by providing the path of the file. Listing 7.2 shows an example of specifying a pureQueryXml file.

```
C:\path\captureFile.pdqxml
```

Listing 7.2 - Example of specifying a pureQueryXml file

If you want to bind the database packages that are associated with a single statement set in a pureQueryXml file, specify the file name, followed by a colon, followed by the base package name of the statement set. Listing 7.3 shows an example of how you could specify the statement set with the base package name `MYPKGA`.

```
C:\path\captureFile.pdqxml:MYPKGA
```

Listing 7.3 - Example of specifying a statement set in a pureQueryXml file
7.2.2.3 Changes to pureQueryXml files

If you are using the StaticBinder to bind the statements in a pureQueryXml file that you created by using the capture functionality of the Client Optimizer, you must use the Configure utility to process the file before you use the file with the StaticBinder utility. You must do so anytime you alter the file, either by performing an incremental capture or by editing it manually. If you do not process the file with the Configure utility, your application could have unexpected results during static execution. If you are using a pureQuery-enabled file in IBM Optim Development Studio, the tool runs the Configure utility for the pureQueryXml files in the pureQueryFolder directory automatically.

Tools that integrate with pureQuery have pureQueryXml files that are created without using the pureQuery capture capability. These tools are .NET, CLI, JPA, and Hibernate. Chapter 17 discusses the integration with JPA. The integration with .NET and CLI is beyond the scope of this book. As with files created using pureQuery capture, you must use the Configure utility to process pureQueryXml files for .NET and CLI after the files are created or altered. The files that are created for integration with JPA and Hibernate do not need to be configured, and the Configure utility cannot process them.

IBM recommends that you only modify pureQueryXml files by using tools provided by IBM. These include capturing, configuring, and using the IBM Optim Development Studio pureQueryXML editor. If you modify pureQueryXml files manually, they could contain an unexpected or invalid combination of values. The Configure utility can correct or identify some of the problems that manual editing can cause. Therefore, IBM particularly recommends that you do not manually edit the pureQueryXml files that are created for JPA and Hibernate.

7.3 Specifying options for the StaticBinder utility

pureQuery provides multiple utilities. All of these utilities allow you to specify options on the command line. For the StaticBinder utility and some of the other utilities, you can specify the interfaces and pureQueryXml files that you want the utility to use, along with the options it should use for each interface or file, in a type of file that is known as an options file. Refer to Appendix D for a description of options files. pureQuery allows two types of options files: genProps and bindProps. The StaticBinder utility uses bindProps files.

An options file is a text file that lists interfaces and pureQueryXml files to be processed, and the options to apply to each interface or file. For pureQueryXml files, you can use a colon and the name (:MYPKG, for example) of a statement set to indicate that the options on the line apply to only the specified statement set. Each interface and pureQueryXml file must be listed on a separate line in the options file, followed by an equal sign (=), and then any options that pureQuery should use when it processes that item. Additionally, in each options file, you can provide default options on a line that starts with defaultOptions. pureQuery uses each option that you specify on the defaultOptions line for all the items, except for those items for which the option is specified with a different value. Listing
7.4 shows an example of how you would provide default options, interfaces, and pureQueryXml files in an options file.

```plaintext
defaultOptions = <options to use as defaults>
myPackage1.MyInterface1 = <options for myPackage1.MyInterface1>
myPackage2.MyInterface2 = <options for myPackage2.MyInterface2>
C:\path1\captureFile1.pdqxml = <options for captureFile1.pdqxml>
C:\path2\captureFile2.pdqxml = <options for captureFile2.pdqxml>
C:\path3\captureFile3.pdqxml:MYPKGA = <options for the statement set MYPKGA>
```

Listing 7.4 - Example of listing default options, interfaces, and pureQueryXml files in an options file

### 7.4 Specifying interfaces and pureQueryXml files in an archive file

You can specify an archive file that contains interfaces and pureQueryXml files that you want the StaticBinder utility to use for the bind. The archive file must be a JAR file with the extension \*.jar, an EAR file with the extension \*.ear, a WAR file with the extension \*.war, or ZIP file with the extension \*.zip.

If you specify an archive file, you must also provide an options file that lists the interfaces and pureQueryXml files to use. The options file can be separate from the archive file, or it can be inside the archive file. When you do not specify an options file explicitly, pureQuery looks for the file in the archive. It performs a breadth-first search of the files in the archive for files with the extension \*.bindProps. pureQuery uses the first such file that it finds as the options file.

### 7.5 Running the StaticBinder utility in IBM Optim Development Studio

pureQuery-enabled projects in IBM Optim Development Studio contain a folder named `pureQueryFolder`. The folder contains two options files: one named `Default.genProps`, and one named `Default.bindProps`. Notice the file `Default.bindProps` in Figure 7.2. When you bind interfaces and pureQueryXml files in IBM Optim Development Studio, the tool uses `Default.bindProps` as your options file.
Through menu options and content assist, IBM Optim Development Studio can help you to create options files. It can help you to add interfaces and pureQueryXml files to your options files, and it can help you add options. IBM recommends that you use Optim Development Studio assistance to add pureQueryXml files to your options files rather than adding them manually. In order for the utilities to determine which options to use, the paths of pureQueryXml files in the options files must exactly match the paths that IBM Optim Development Studio uses when it runs the utilities. Using ODS assistance will ensure the paths match.

If you are using pureQuery-enabled project in IBM Optim Development Studio, you can use the menu options to bind the interfaces and pureQueryXml files in your project. The following list describes how you can bind different portions of a project. In all cases, after you select Bind… or Bind pureQuery Application, select the connection that you want to use for the bind and click Finish, as shown in Figure 7.7. The tool uses the connection that you select to perform all of the binds that you requested.

- **Everything in a project**: Figure 7.3 shows how you can bind the statements in the interfaces in a project, as well as those in pureQueryXml files in the directory pureQueryFolder. Right-click on the project in the Package Explorer view and select pureQuery, and then select Bind pureQuery Application.

- **Everything in a bindProps file**: Figure 7.4 shows how you can bind all of the entries in a bind props file by right-clicking on the file in the Package Explorer view and selecting pureQuery and then selecting Bind….

- **A single interface, pureQueryXml file, or archive**: Figure 7.5 shows how you can bind the statements in an interface, pureQueryXml file, or archive file by right-clicking on the file in the Package Explorer view and selecting pureQuery and then selecting Bind….

- **One statement set in a pureQueryXml file**: Figure 7.6 shows how you can bind the packages associated with a statement set. In the Database Packages tab of the SQL Outline menu, right-click on a statement set and select Bind….
Getting Started with pureQuery

Figure 7.3 - Right-click on a project and select pureQuery, and then select **Bind pureQuery Application** to bind the statements for the project. This binds the statements in the interfaces in the project, as well as statements in pureQueryXml files in the directory pureQueryFolder.

Figure 7.4 - Right-click on a bindProps file and select pureQuery, and then select **Bind...** to bind all of the entries that are in the bindProps file.

Figure 7.5 - Right-click on an interface or a pureQueryXml file and select pureQuery, and then select **Bind...** to bind the statements it contains.
Figure 7.6 - In the Database packages tab of the SQL Outline menu, right-click on a statement set and select Bind… to bind the database packages for that statement set.

Figure 7.7 - After you select Bind..., choose the connection that you want to use for the bind and click Finish.
In bindProps files, you can have more than one entry for each interface or pureQueryXml file. When you bind an interface, a pureQueryXml file, or a single statement set from a pureQueryXml file, the utility binds using every entry in the options file that refers to the item that you selected. If you specify a pureQueryXml file, the utility also binds entries that represent statement sets in the pureQueryXml file. This allows different statement sets within a single pureQueryXml file to be bound using different options.

The StaticBinder utility performs binds for entries in the order of the entries in the file. When you specify a pureQueryXml file that is not in the bindProps file at all, or for which only certain statement sets are in the file, the utility first binds the file using the default options, and then every entry in the options file that contains a statement set from the file. Performing a bind in IBM Optim Development Studio is equivalent to running the StaticBinder utility on the command-line while specifying an options file and interfaces or pureQueryXml files.

When you add interfaces and pureQueryXml files to your project that you want to use for binding, it can be helpful to add entries for them to the bindProps file. This way, when you have options that you want to specify, it is easy to add them. In addition, adding everything that must be bound to the options file makes it easy to bind the application from the command line.

When you bind from an archive file in IBM Optim Development Studio, the archive file must contain an options file with the file extension .bindProps.

Refer to the documentation for more information about running the StaticBinder utility in IBM Optim Development Studio.

### 7.6 Running the StaticBinder utility from the command line

You can easily run the StaticBinder utility from the command line. With pdq.jar, pdqmgmt.jar, and the JAR files for the IBM Data Server Driver for JDBC and SQLJ in the classpath, use the command in Listing 7.5.

```java
java com.ibm.pdq.tools.StaticBinder
```

Listing 7.5 - The command for running the StaticBinder utility

#### 7.6.1 Specifying database connection information

Use the options -url, -username, and -password to specify the URL, username, and password for the database connection. You can specify these options either directly on the command line, or in an options file.

Listing 7.6 shows an example of specifying the connection information directly on the command line. A value that you specify for an option on the command line overrides any values that you specify for that option in an options file.
Listing 7.6 - An example of specifying the URL, username, and password on the command line

Listing 7.7 shows an example of specifying the connection information on the defaultOptions line in an options file.

defaultOptions = -url yourURL -username yourUsername -password yourPassword

Listing 7.7 - An example of specifying the URL, username, and password on the defaultOptions line in an options file

Listing 7.8 shows an example of specifying the connection information for a specific entry in an options file. You can provide different connection information for different entries in an options file if you want to bind the entries on different databases.

C:\path\captureFile.pdqxml = -url yourURL -username yourUsername -password yourPassword

Listing 7.8 - An example of specifying the URL, username, and password for an entry in an options file

7.6.2 Specifying the interfaces and pureQueryXml files

You must specify pureQueryXml files or interfaces that contain the SQL statements that pureQuery should bind. There are several ways to specify interfaces and pureQueryXml files.

7.6.2.1 Specifying interfaces or pureQueryXml files on the command line

You can specify interfaces by using the option -interface, as shown in Listing 7.9.

-interface myPackage1.MyInterface1

Listing 7.9 - Example of specifying an interface to the StaticBinder utility

You can specify pureQueryXml files by using the option -pureQueryXml, followed by the paths of one or more pureQueryXml files. Listing 7.10 shows an example of specifying a pureQueryXml file.

-pureQueryXml "C:\path\captureFile.pdqxml"

Listing 7.10 - Example of specifying a pureQueryXml file to the StaticBinder utility

If you want to bind the database packages that are associated with a single statement set in a pureQueryXml file, use the -pureQueryXml option to specify the file name, followed by a colon, followed by the base package name of the statement set. Listing 7.11 shows an example of how you could specify the statement set with the base package name MYPKGA.

-pureQueryXml "C:\path\captureFile.pdqxml:MYPKGA"

Listing 7.11 - Example of specifying a statement set in a pureQueryXml file to the StaticBinder utility
7.6.2.2 Specifying an options file on the command line
You can use the option \-optionsFile to specify an options file that contains the interfaces and pureQueryXml files that you want the StaticBinder utility to bind, along with the options that the utility should use for each of them. pureQuery performs a bind for each interface or pureQueryXml file in the options file, in the order in which they are listed. Listing 7.12 shows an example of specifying an options file.

\-optionsFile "C:\path\Default.bindProps"

Listing 7.12 - Example of specifying an options file that lists the interfaces and pureQueryXml files to bind

7.6.2.3 Specifying an options file and interfaces or pureQueryXml files on the command line
If you specify both an options file and interfaces or pureQueryXml files on the command line, pureQuery binds the interfaces and pureQueryXml files that you specify on the command line, using the options that you specify on the command line, combined with the options in the options file. Listing 7.13 shows an example of specifying an options file and a pureQueryXml file.

\-optionsFile "C:\path\Default.bindProps" \-pureQueryXml "C:\path\captureFile.pdqxml"

Listing 7.13 - Example of specifying an options file and a pureQueryXml file on the command line

7.6.2.4 Specifying an archive file on the command line
You can use the option \-archive to specify an archive file that contains interfaces and pureQueryXml files that you want the StaticBinder utility to use for the bind. If you specify an archive file, you must also provide an options file that lists the interfaces and pureQueryXml files to use. You can specify the file on the command line, by using the option \-optionsFile. When you provide an archive file without using the option \-optionsFile on the command line, pureQuery looks for the file in the archive.

Listing 7.14 shows an example of specifying an archive file and an options file on the command line. Listing 7.15 shows an example of specifying just an archive file on the command line.

\-archive "C:\path\myArchive.jar" \-optionsFile "C:\path\Default.bindProps"

Listing 7.14 - Example of specifying an archive file and an options file.

\-archive "C:\path\myArchive.jar"

Listing 7.15 - Example of specifying an archive file without an options file. The archive file must contain an options file.
7.6.3 Specifying additional options

You can customize the behavior of the StaticBinder utility by specifying additional options, either on the command line or in an options file. Specify \-help to see a list of the available options. Refer to the IBM Optim pureQuery Runtime documentation for details about the many options that are available for the StaticBinder utility.

You can specify the additional options either on the command line, or in the options file. When you specify an option both on the command line and in the options file, pureQuery will use the value that is on the command line.

7.6.4 Example commands for the StaticBinder utility

The following listings provide several sample commands that you could use to run the StaticBinder utility. Remember that anytime you run the StaticBinder utility from the command line, you must have pdq.jar, pdqmgt.jar, and the JAR files for the IBM Data Server Driver for JDBC and SQLJ in the classpath.

Listing 7.16 shows an example of specifying database connection information and two interfaces on the command line. The utility binds each of the interfaces using the connection information.

```
```

Listing 7.16 - Example of running the StaticBinder utility while specifying two interfaces on the command line

Listing 7.17 shows an example of specifying database connection information and two statement sets from a pureQueryXml file on the command line. The utility binds the two statement sets using the connection information.

```
java com.ibm.pdq.tools.StaticBinder -url yourURL -username yourUsername -password yourPassword -pureQueryXml "C:\path\captureFile.pdqxml:MYPKGA" "C:\path\captureFile.pdqxml:MYPKGB"
```

Listing 7.17 - Example of running the StaticBinder utility while specifying two statement sets from a pureQueryXml file on the command line

Listing 7.18 shows an example of specifying database connection information and an options file on the command line. The utility uses the connection information to bind all of the entries in the options file.

```
java com.ibm.pdq.tools.StaticBinder -url yourURL -username yourUsername -password yourPassword -optionsFile "C:\path\Default.bindProps"
```

Listing 7.18 - An example of running the StaticBinder utility while specifying database connection information and an options file on the command line

Listing 7.19 shows an example of specifying an options file and an interface on the command line. The options file must contain the database connection information. The
utility uses the connection information and other options in the options file to bind the two interfaces.

```java
```

Listing 7.19 - An example of running the StaticBinder utility while specifying an options file and two interfaces on the command line

Listing 7.20 shows an example of specifying database connection information and an archive file on the command line. The utility locates the first bindProps file in the archive and uses the connection information to bind all of the entries that it contains.

```java
java com.ibm.pdq.tools.StaticBinder -url yourURL -username yourUsername -password yourPassword -archive "C:\path\myArchive.jar"
```

Listing 7.20 - An example of running the StaticBinder utility while specifying database connection information and an archive file on the command line

Refer to the documentation for more information about running the StaticBinder utility from the command line.

### 7.7 Advanced features

The StaticBinder provides many advanced features that are outside of the scope of this book. This list describes some of them. For details about using these features, refer to the IBM Optim pureQuery Runtime documentation.

- **Verify packages**: You can use the option `-verifyPackages` to have the utility check for the existence of packages in the database and show information about them, rather than binding packages in the database.

- **Generate DBRM files**: You can use the option `-generateDBRM` to have the utility generate DBRM files instead of binding packages in the database. (DBRM files are specific to DB2. See DB2 product documentation for more information.)

- **Grant EXECUTE privileges**: You can use the option `-grant` to have the utility grant EXECUTE privileges to specified users.

If you are using IBM Optim Development Studio, it provides some additional features that you can use for binding. This list describes some of them.

- **Saved binding scripts**: You can direct IBM Optim Development Studio to save script files for the binds that you perform. Each time you perform a bind, the tooling generates a script file that you can use to perform an identical bind from the command line. To enable this feature, you can right-click on a pureQuery-enabled project, select Properties, and select the pureQuery page. In the Saved Scripts section, click the checkbox...
next to Save commands that the workbench sends to the StaticBinder utility.

- **Automatic binding:** You can instruct IBM Optim Development Studio to bind pureQueryXml files automatically when you save them in the pureQueryXML editor. To enable this feature, you can right-click on a pureQuery-enabled project, select Properties, and select the pureQuery page. In the pureQueryXML Editing Preferences section, click the checkbox next to Automatically bind file on save.

### 7.8 Summary

You can use the StaticBinder utility to bind the SQL statements in interfaces and pureQueryXml files. When binding from interfaces, each database package contains the SQL statements in an interface. When binding from pureQueryXml files, each database package contains the SQL statements in a statement set of a pureQueryXml file. You can specify the interfaces and pureQueryXml files directly, or you can specify an archive file that contains them. You can specify the interfaces and pureQueryXml files, along with the options that you want the utility to use during the bind, on the command line or in an options file. If you are using IBM Optim Development Studio, you can use menu options to select the files that you want the utility to use.

### 7.9 Review questions

1. You can use the StaticBinder utility to bind SQL statements. Where can you specify the SQL statements?
   - A. In a file with the extension .txt
   - B. In an interface of annotated methods
   - C. In any XML file
   - D. In a pureQueryXml file

2. Where can you specify the options that you want the StaticBinder utility to use for the bind?
   - A. On the command line
   - B. In an options file
   - C. In a Java properties file
   - D. None of the above

3. If you direct the StaticBinder utility to bind packages from an interface, and if you do not specify any options (besides -interface and the connection information), what packages does the utility bind?
4. How does the StaticBinder utility determine the names of the packages that it binds, when binding from an interface?

5. How does the StaticBinder utility determine the names of the packages that it binds, when binding from a pureQueryXml file?

6. Which option must you use to indicate to pureQuery that it must bind database packages at only one isolation level for interfaces and pureQueryXml files? This option also indicates to pureQuery that it must not append 1, 2, 3, or 4 to the names.
   A. `-forceSingleBindIsolation`, specified to the Generator utility or to the Configure utility
   B. `-forceSingleBindIsolation`, specified to the StaticBinder utility
   C. `-isolation`, specified to the Generator utility or to the Configure utility
   D. `-isolation`, specified to the StaticBinder utility

7. Which option must you use to direct pureQuery to bind packages at only one isolation level?
   A. `-forceSingleBindIsolation`, specified to the Generator utility or to the Configure utility
   B. `-forceSingleBindIsolation`, specified to the StaticBinder utility
   C. `-isolation`, specified to the Generator utility or to the Configure utility
   D. `-isolation`, specified to the StaticBinder utility

8. How do you direct the StaticBinder utility to bind only one statement set of a pureQueryXml file?

9. You can specify an archive file that contains the interfaces and pureQueryXml files that you want the utility to use for the bind. What are the extensions that the archive file can have?

10. If you specify an options file to the StaticBinder utility on the command line, and you do not specify `-interface` or `-pureQueryXml`, what does the utility do?
    A. It does not bind anything
    B. It binds everything that is listed in the options file, in the order in which the entries are listed
    C. It binds the interfaces that are listed in the options file
    D. It binds the pureQueryXml files that are listed in the options file
11. True or false: When you run the StaticBinder utility from the command line, and you specify an options file, the options file can direct the utility to bind packages in multiple databases.

7.10 Exercise

If you are using IBM Optim Development Studio, use menu options to bind everything in the GettingStartedWithPureQuery project. Refer to Section 11.5 for instructions on how to do this.
Chapter 8 - Extended Insight

8.1 Extended Insight: the big picture

pureQuery delivers extensive client monitoring capabilities via its Extended Insight feature in conjunction with IBM's Optim Performance Manager Extended Edition product. Extended Insight gives the DBA insight into the performance of previously obscured data access layers that affect data access application performance. For data access applications written using pureQuery API's, pureQuery Client Optimized applications, and JDBC or CLI applications using supported JDBC or CLI drivers, Extended Insight delivers database client, application server, and network statistics side by side with database statistics from traditional database monitoring tools. In addition to statistics, pureQuery also supplies application metadata that works in conjunction with Extended Insight data to provide quick identification of applications executing problem SQL.

In a multi-tiered application environment, DBA's may be asked to "solve the problem" of a slow response time for a running application, before it is even known whether the problem is in the database or somewhere in the many layers between the application and the database. In this situation, DBA's may have little information to help them identify which parts of the application are issuing problematic SQL. Extended Insight helps DBA's who are faced with problem determination in a multi-tiered environment. It increases visibility into the end-to-end stack so the DBA can quickly pinpoint the layer in the stack or identify the application that is causing problems in a running, monitored system. Extended Insight capabilities are presented seamlessly integrated in Optim Performance Manager along with the traditional tools DBA's use for database monitoring and troubleshooting.

8.2 Common Problems in Multi-Tiered Monitoring

Application-centric DBAs who support databases for dynamic data access applications in multi-tiered environments face a common set of problems resulting from the large number of layers between an executing application and the database.
Figure 8.1 - In a multi-tiered environment there is less information at the database server than the DBA needs to quickly identify problem sources.

In a typical environment shown in Figure 8.1, the database administrator sees SQL activity that is identified by a generic dynamic driver package (labeled “A” in this picture), used by the database driver executing application SQL. The identity of the application issuing the SQL is obscured by the generic driver package and several layers of the architecture. User id’s may not be visible at the database either, if they are masked by the connection pool. It’s also possible the application developer didn’t configure client property settings that would help the DBA identify different parts of the client application and workload.

Administrators solving performance problems observed at the application layer, who are using traditional database monitoring tools, can easily look at database performance and statistics to identify and solve problems in the database itself. Likewise, traditional application monitoring tools identify issues with the business logic or application server management layer. However, sometimes the problem is not found in either of these layers. Administrators historically lack needed information about network, driver, and persistence layer activity to quickly pinpoint the actual source of the problem.

Figure 8.2 shows the gap in useful monitoring information:
pureQuery client monitoring solves this problem by increasing application visibility at the monitor for SQL executing in a multi-tiered environment, and providing useful monitoring statistics from portions of the stack that are usually obscured by persistence frameworks.

8.3 Monitoring pureQuery Clients

8.3.1 More Monitored Layers

Extended Insight provides new monitoring information from the perspective of the application client, and integrates it with traditional database-centric and application-centric monitoring data, as shown in Figure 8.3:
Figure 8.3 – Extended Insight collects useful information from the middle tiers and makes it available to the DBA.

Optim Performance Manager (OPM) provides integrated interfaces to view Extended Insight data alongside traditional database monitoring statistics. These products work together to deliver client monitoring functionality, with Optim pureQuery Extended Insight installed on the application server, and OPM Extended Edition server installed on a network-reachable machine as shown in Figure 8.4:
Figure 8.4 - Deployment configuration for monitoring. Extended Insight is available for IBM Java or CLI clients, via Optim Performance Manager Extended Edition and OPM Extended Insight.

The OPM interface integrates Extended Insight data seamlessly with traditional database statistics. Workload summaries can be drilled down in the GUI to view details such as time spent in the server, time spent in the network, or time spent waiting for a connection from the Websphere Application Server connection pool (requires minimum WAS version). Figure 8.5 shows how client monitoring statistics are traversed as a DBA identifies a problem in the middle layer of the application.
Extended Insight gives DBA’s insight into the application stack layers above the database, to solve data access problems quickly. Even packaged and framework applications using the prerequisite IBM JDBC or CLI drivers can take advantage of the additional client statistics provided via client monitoring.

8.3.2 Identifying pureQuery Applications Using Extended Insight

pureQuery applications can take special advantage of certain Extended Insight features. In addition to providing monitoring statistics from previously obscured stack layers, pureQuery client monitoring also gives DBA’s important identifying information about pureQuery client applications.

In Figure 8.1, the identity of applications issuing SQL is obscured from the DBA monitoring the database. They are identified to the DBA only as generic, dynamic, JDBC applications. If there are thousands of dynamic JDBC applications in the enterprise, this does not give the DBA enough information to quickly identify the application executing a problematic SQL statement. Java database drivers use one set of common dynamic packages to execute dynamic SQL. The end result is that most Java workloads are grouped into one large, shapeless, and obscure category by database monitors and workload management tools.
It's possible to improve this situation by executing static applications (see Appendix C Static SQL to learn more), or if client applications are configured to use DB2 driver client properties (for example, by using one of the JDBC 4.0 API's `setClientInfo(String name, String value)` or `setClientInfo(Properties properties)` methods on the active connection). Otherwise, the DBA monitoring the workload will still have trouble identifying the dynamic application executing a given SQL statement. Yet this is often exactly what is required to resolve the problem. Extended Insight together with pureQuery application metadata provide a more reliable way to address this issue and result in quicker problem resolution times for pureQuery applications.

8.3.2.1 pureQueryXml Speeds Problem Resolution

pureQuery applications such as Client Optimized JDBC or CLI applications, pureQuery annotated method style applications, pureQuery Data Web Services, or applications created using the Websphere JPA / pureQuery integration solution, all produce a set of pureQuery application metadata called pureQueryXml. pureQuery application metadata is stored in an xml file, and describes a set of SQL statements and their source. It is created during the capture and configure phase for Client Optimized applications, and during the development phase for pureQuery applications developed using Optim Development Studio.

pureQuery tooling provides the capability to register pureQueryXml with Optim Performance Manager using the `ManageRepository` tool. To maximize the problem determination effectiveness of OPM EI, it is important to register the new pureQueryXml with OPM when a new version of a pureQueryXml-associated application is deployed in the enterprise. Registering pureQueryXml with OPM makes it easier to monitor and perform problem determination throughout the enterprise, and it is an important deployment step that ties together all stages of the application lifecycle. For example, registered metadata makes it easier for a DBA who is monitoring an SQL statement to identify the application type and application version executing that statement.

Figure 8.6 shows the lifecycle process that maximizes the capability of Optim tooling to improve application lifecycle management. The application developer/system administrator registers a set of pureQuery application metadata with OPM for each deployed pureQuery application in the enterprise. This improves the ability of the database administrator using OPM for real-time monitoring to quickly identify problem applications, and integrates key information throughout the lifecycle to improve communication and speed up problem resolution:
Figure 8.6 - Metadata registration and client monitoring make it easy for a DBA to identify applications causing database problems.

Improving the ease with which DBAs and application developers can communicate about problems using a common language and references is one of the biggest advantages to using Optim Performance Manager Extended Insight with pureQuery application metadata.

8.4 Extended Insight For A Variety of Application Types

8.4.1 Monitoring Plain JDBC or CLI applications

Extended Insight always collects client-side statement statistics, even for plain JDBC or CLI applications that use supported drivers but have never been client optimized. Each statistic is aggregated per statement and per client-user during a given time period. Client statistics can be grouped by a variety of characteristics such as workstation name, client application identifiers, or users. These characteristics are said to define the monitored workload. The following are the primary aggregated statistics reported via OPM and Extended Insight for each monitored statement:
- ResultSet processing time (aggregated application time from query execution through ResultSet close for a particular query and client user)
- Driver time (aggregated, accumulated time spent in the IBM driver for a particular statement and client user - includes open, fetch and close cursor times for queries, and execute time for non-queries)
- Network time (aggregated, accumulated time spent in the network per statement and client user)
- Server time (aggregated, accumulated execution time reported by the server for a particular statement and client user)

Additional detailed information is also available about each client, such as a breakdown of transaction times for the client, information about negative SQL codes, and certain Websphere Application Server statistics, such as connection pool wait times (minimum level of WAS required). See the OPM documentation for the complete set of information available for client monitored workloads.

### 8.4.2 Monitoring pureQuery Applications associated with pureQueryXml

Several types of applications are associated with application metadata in the form of pureQueryXml. These include pureQuery annotated method style applications, applications resulting from any pureQuery integrated tooling that generates pureQueryXml (such as Data Web Service (DWS) applications or WAS/JPA pureQuery applications), and Client-Optimized JDBC or CLI applications. See Chapter 11, Chapter 16, and Chapter 6 for more information about creating these types of applications.

Extended Insight collects the same statistics for pureQuery applications associated with pureQueryXML, as it does for plain JDBC or CLI applications. In addition, for most applications associated with pureQueryXml, Extended Insight also identifies the pureQueryXml element associated with the executing statement. This means that when the pureQueryXml is registered with OPM and Extended Insight monitoring is enabled, a DBA can use the monitoring tools to easily identify additional information about a pureQuery statement associated with pureQueryXml. For generated or tooling-produced pureQueryXml, this information includes:

- Source class
- Source timestamp (version)
- Source line of code containing the given SQL string

For client optimized applications executing or loading captured SQL from pureQueryXml (see Chapter 6 for the variety of configurations under which pureQuery Client Optimization executes captured SQL), this information includes:

- pureQueryXml file version
• pureQueryXml package name and statement entry id
• Information about the call stack of the statement at the time the metadata was created (when the SQL was captured)

This information, when provided to the application developer, can be used to jump-start problem resolution for monitored applications associated with pureQueryXml.

8.5 Summary
Extended Insight provides visibility into, and knowledge about, the application stack that was previously unavailable to DBAs and others trying to monitor and troubleshoot multi-tiered, data access applications. It provides the ability to pinpoint specific java applications as they are running, regardless of whether they are executing statically or dynamically. For each identified, client-monitored workload, result set processing time, driver time, network time, and server time, along with a variety of additional metrics, can be examined for any group of transactions or statements. This allows the application-centric DBA to identify, rule out, and pinpoint problems anywhere in the data access application stack, not only within the database. Importantly, pureQuery monitoring can increase visibility into the data access stack for existing applications, running in existing frameworks, without changing application code. pureQuery applications with pureQueryXml registered with Optim Performance Manager benefit from the deepest problem determination support, and plain JDBC or CLI applications also benefit from increased visibility through Extended Insight. When pureQuery application metadata is registered with Optim Performance Manager, the monitor can identify the issuing pureQuery application for many pureQuery-issued SQL statements, down to the specific application source line or the specific capture snapshot, reducing resolution and fix time. Extended Insight provides a previously missing link across the persistence layer and data access frameworks. It fits seamlessly between existing application monitor coverage zones and traditional application and database monitoring tools, to monitor currently obscured layers of the data access stack, reducing problem determination time and increasing monitoring effectiveness.

8.6 Review Questions
1. What are some common problems faced by DBA’s in a multi-tiered environment?
2. What is a native pureQuery application?
3. True or false: Extended Insight integrates seamlessly with database-centric monitor data?
4. What is the necessary step required before pureQuery application metadata is useful to the DBA performing monitoring activities?
5. Which stack layers does Extended Insight provide new monitoring statistics for?

6. Which of the following products is installed on the application server to deliver client monitoring capabilities?
   A. Data Studio
   B. Optim Development Studio
   C. Optim Performance Manager (OPM) Extended Insight

7. Which of the following pieces of monitoring information is available for native pureQuery applications whose metadata is registered with OPM?
   A. Source class name
   B. Source timestamp (version)
   C. Source code line number
   D. All of the above

8. Which of the following is NOT a benefit of client monitoring:
   A. Decrease problem determination time
   B. Correlate monitored SQL statements with originating line of Java source code
   C. .NET visual development
   D. Provide insight into previously obscured layers of the data access stack

9. Which types of applications can be monitored using pureQuery client monitoring?
   A. plain JDBC or CLI applications
   B. native pureQuery applications
   C. pureQuery client optimized applications
   D. All of the above

10. How does OPM correlate an SQL statement with its source application?
    A. Using registered pureQuery application metadata
    B. String matching
    C. Scanning the source code repository
    D. It doesn’t
PART IV – API, RUNTIME AND SERVICES
9.1 Persistence Frameworks and Data Access: the big picture

Almost every business application has a requirement to store data across sessions. This is known as persistence. Persistence frameworks support enterprise applications by providing a set of persistence services, such as transaction management and object caching. The goal of persistence frameworks is to shield business objects from persistence-related tasks. With an underlying persistence framework, business object implementations can focus on carrying out business logic, while the underlying persistence framework handles most of the work behind the scenes for update, persist, remove, and find operations against a data store.

A persistence framework can store data in any number of ways – in object databases, flat files, xml files or any other data storage solution. However, for reliable and scalable persistence (along with many other benefits), many persistence frameworks use a relational database management system (RDBMS), such as IBM’s DB2, for the underlying data store. Frameworks that use a relational database also provide support and implementation for object-relational mapping. The object-relational mapper handles the issues associated with moving data between an object-oriented application and a relational database.

pureQuery is not a persistence framework, but pureQuery features enhance the persistence framework data access layer. pureQuery is designed to enhance a variety of persistence frameworks and provides features that integrate with, and extend beyond, the persistence layer. Figure 9.1 depicts the variety of frameworks pureQuery enhances:
Object-relational mapping refers to the process of transferring data back and forth (mapping) between the object-oriented software layer and an underlying relational database. Object-relational mapping is one of the services provided by persistence frameworks that use a relational database backend.

The goal of the object-relational mapping layer is to transfer data efficiently and correctly between two different paradigms – object and relational – such that neither the performance nor data integrity of either is lost.

The object-relational mapping layer encapsulates knowledge of the relational database layer, so business logic can be designed independently from the database schema. The software engineering benefits of encapsulation include the ability for each layer to be optimized individually, and the reduction of the impact of maintenance, upgrades, or redesign of either layer to the other.

pureQuery’s object-relational mapping tools automate the development of an optimized object-relational mapping layer, and minimize the time, effort, and system resources required to develop, maintain, manage, and run the data access application.

### 9.1.1 pureQuery and Persistence Frameworks

pureQuery integrates with a variety of persistence frameworks to provide improved development time, manageability, security, and performance for framework applications.
9.1.1.1 JPA
The Java Persistence API (JPA) has become the current de-facto standard for Java persistence. JPA delivers persistence support for both JSE and JEE applications. Implementations of JPA are available in the open source community and from a variety of vendors, including IBM’s Websphere Application Server. See Chapter 16 for more information about features designed to make development, management, and monitoring of JPA applications quicker with pureQuery.

9.1.1.2 Hibernate and iBatis
Hibernate and iBatis are open-source projects that provide object-relational mapping tools, API’s, and object-relational persistence. They continue to release new versions to provide such features as stronger and simpler object relational mapping supporting complex joins, queries from multiple tables, and automated mapping for object hierarchies and n-cardinality relationships. pureQuery integration with Hibernate and iBatis accelerates data access from either of these frameworks. Acceleration of Hibernate and iBatis applications can be achieved using the pureQuery Client Optimizer, described in Chapter 6.

9.2 pureQuery APIs in Context
There is no “one size fits all” solution for a data access layer. The ideal mapping solution varies depending on the type of data store, the database schema if it’s a database, and what types of business objects and processes are represented in the object layer. As a result, pureQuery tooling supports a variety of access layer paradigms, from quick, generated, simple table-to-class maps, to tools that support advanced fine-tuning for complex query results.

pureQuery makes it easy to implement the core J2EE patterns Data Access Object, which abstracts and encapsulates persistent data access, and Transfer Object, which encapsulates the business data, in a variety of ways. For example, pureQuery can quickly create the simplest type of object-relational map, which defines one class for each relational table. The class contains a field for each column in a table, and each instance represents a row in the table, following the active record and rowset patterns.

However, a more complex mapping solution is required under many common data access circumstances and pureQuery provides intuitive support for these as well. If the same data is stored in several classes/tables (as often happens since classes represent interconnected real-world objects), a simple class-to-table map breaks during normalization. This is because normalization ensures the data is represented in only one table in the database. Object-relational mapping must therefore be fine-tuned after database normalization.

pureQuery provides a comprehensive and flexible set of approaches for creating a data access layer. It supports integration of relational data, object, and xml data. It supports a variety of data servers and supports mixed queries across data servers and in-memory cache objects using standard SQL.
Another common pain point is that the data access layer can become a “black hole” for problem determination efforts. Often the data access layer is an obscure and untraced layer performing critical database-to-object mapping, but presents an information void from the perspective of traditional monitoring and problem determination tools. While databases and application servers have individually developed robust monitoring and problem determination tools, problem determination systems historically do not bridge the gap across the data access layer. pureQuery addresses this by integrating with end-to-end monitoring that specifically links problem determination through and across the data access layer. See Chapter 8 to learn about tools that integrate with pureQuery to monitor across the data access layer.

9.3 pureQuery Data Access Overview

9.3.1 pureQuery Beans as Transfer Objects

The Transfer Object pattern encapsulates business data and transfers it among other business objects. A pureQuery data bean is an implementation of the Transfer Object pattern. It acts as a container for a logical set of information used and manipulated by the business application layer and the data access layer. If the data in the bean originates in or must be persisted to a relational database, the bean is usually the primary object operated on by the Data Access Object. Listing 9.1, shows an EmployeeBean, which acts as a container for information modeling a real-world company employee. Note that it holds information which needs to be persisted in the database, and also may be referenced by other business objects.

```java
public class EmployeeBean {
    public EmployeeBean ()
    {}

    private String empno;
    public String getEmpno ()
    public void setEmpno (String empno);
    private String firstName;
    public void setFirstName (String firstName);
    public String getFirstName ()
    ...
}
```

Listing 9.1- The EmployeeBean models a real-world company employee.

pureQuery associates a data bean with SQL statements, database tables, and/or columns within the tables. Listing 9.2 shows a sample SQL query that could be used to populate the EmployeeBean. Note that this query returns results whose column names don’t exactly
match the bean field names in the EmployeeBean above. pureQuery does the work to map the result columns to the correct bean fields.

```
SELECT EMPLOYEE.EMPNO, EMPLOYEE.FIRSTNME, EMPLOYEE.LASTNAME, DEPARTMENT.DEPTNO FROM EMPLOYEE LEFT JOIN DEPARTMENT ON EMPLOYEE.WORKDEPT = DEPARTMENT.DEPTNO
```

**Listing 9.2- An example SQL query to populate the EmployeeBean.**

A pureQuery bean is a Java class that defines a set of properties. Each property is defined either by a public field, or by a pair of public accessor methods. The following rules define how pureQuery determines the properties in a bean.

- **Public field:** The name of the field indicates the name of the property. For example, a field defined as `public String employeeNumber` would represent a property named "employeeNumber".

- **Public accessor methods:** Each property defined by accessor methods must have one **setter** accessor method and one **getter** accessor method. Any setter method in a pureQuery bean must have a corresponding getter method, and vice versa. Setter and getter methods are defined according to these rules:
  
  - **Setter methods:** The names of setter methods start with "set". These methods have exactly one parameter. A setter method named "setAbcDef" indicates a property named "abcDef." For example, a bean could contain a method that is defined as: `public void setEmployeeNumber(String)`. Such a method would indicate a property named "employeeNumber."
  
  - **Getter methods:** The names of getter methods start with "get" or "is". These methods have no parameters. A getter method named "getAbcDef" indicates a property named "abcDef," as does a getter method named "isAbcDef" when property "abcDef" is of Boolean type. For example, a bean could contain a method that is defined as: `public String getEmployeeNumber()`. Such a method would indicate a property named "employeeNumber."

The names of the properties in a pureQuery bean must be unique in a case-insensitive search. Therefore, for example, if you have a pair of public accessor methods named `setEmployeeNumber` and `getEmployeeNumber`, you could not also have a pair of public accessor methods named `setEmployeenumber` and `getEmployeenumber`. You also could not have a public field named `employeenumber`.

Every pureQuery bean must have a no-argument constructor, either explicit or implicit. Additionally, the type of each property must be supported by JDBC.

By default, pureQuery maps a pureQuery bean property to a database column with the same name or alias. You can use Java annotations to change these mappings. The annotations are as follows:

- `@Column`
Getting Started with pureQuery

- Provides a single column to property mapping at the bean field level, and is applied to either the public getter method or the public setter method or the public field

  - @ColumnOverride

    - Provides a single override for a column-to-property mapping at the bean class level that will override a mapping defined using @Column. It will override the default mapping or any column mapping specified elsewhere for the bean, even in one of the bean’s superclasses.

  - @ColumnOverrides

    - Provides a list of column-to-property mappings at the bean class level that override mappings set using @Column. It will override either the default mapping, or any mappings specified elsewhere for the bean, even in one of the bean’s superclasses. @ColumnOverrides makes it easy to define a bean subclass which needs to map to and from the relational layer differently than its superclass. This annotation is not used with @ColumnOverride. They are mutually exclusive.

  - @GeneratedKey

    - Indicates a property which will be populated by a database generated value, and is applied to either the public getter method or the public setter method or the public field. The indicated property is populated prior to returning from an annotated method in which the bean is used as a parameter for an SQL insert statement that returns a generated key value. (The annotation is only supported when the bean is used for non-batched inserts).

When using annotations to override pureQuery’s default mapping mechanism, it is a best programming practice to be consistent when choosing which bean members to apply the annotations to, such as always applying @Column annotations to the public getter methods (this prevents the mistake of accidentally applying two mapping annotations for the same field, for example, one on the getter and one on the setter).

If you use a pureQuery bean to set the SQL parameters of a DML statement, you must use parameter markers to indicate the name of the bean properties that correspond to the SQL parameters.

Table 3.1 shows some simple examples using the default pureQuery data access layer mapping between bean fields and SQL results or parameters. For a complete description of all the options for bean mapping, see the online pureQuery documentation:

<table>
<thead>
<tr>
<th>sql property of annotated method</th>
<th>pureQuery bean Annotation</th>
<th>Bean field or method</th>
<th>pureQuery mapping rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT EMPNO FROM EMPLOYEE</td>
<td>none</td>
<td>setEmpno()</td>
<td>Result column name is a case insensitive match to a field with a public accessor method</td>
</tr>
</tbody>
</table>
### Chapter 9 – Persistence Frameworks and Data Access: Context for pureQuery

#### 9.1.1 Data Access with Annotations

<table>
<thead>
<tr>
<th>sql property of annotated method</th>
<th>pureQuery bean Annotation</th>
<th>Bean field or method</th>
<th>pureQuery mapping rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT FIRSTNM FROM EMPLOYEE</td>
<td>@Column (name=&quot;firstnm&quot;)</td>
<td>setFirstName()</td>
<td>@Column annotation specifies non default mapping between a result column name and a property's public accessor method using a case insensitive match with the result column name</td>
</tr>
<tr>
<td>SELECT GOESBYNAME FROM EMPLOYEE</td>
<td>@ColumnOverride (column=&quot;goesbyname&quot;, propertyName=&quot;firstName&quot;)</td>
<td>setFirstName()</td>
<td>@ColumnOverride annotation at the bean class level overrides any other mappings</td>
</tr>
<tr>
<td>INSERT INTO EMPLOYEE (LASTNAME) VALUES (:lastname)</td>
<td>@GeneratedKey</td>
<td>getEmpno()</td>
<td>Before an annotated method with a bean parameter returns, the bean field corresponding to the @GeneratedKey annotation will be populated using the value returned from the database</td>
</tr>
</tbody>
</table>

#### Table 9.1 – Examples of default pureQuery bean mapping without annotations

Even though the pureQuery data access layer covers a wide variety of basic mappings by default, there are cases when the user might want to override the default mapping mechanism. If so, the user will specify the mapping by using the Annotations described above in the bean definition. Table 3.2 gives examples for using Annotations to override the default pureQuery mapping mechanism:
IBM Optim Development Studio provides features that automate many of the steps involved in creating pureQuery bean specifications, and to verify correct mapping to and from database tables or SQL results. pureQuery tooling supports live checking against a database schema so developers can ensure the object-relational mappings they have specified function as they expect.

See Chapter 3 to learn how Optim Development Studio support simplifies the following bean-related tasks to help quickly and accurately implement the data access layer:

- Automatically generating data beans from a database schema definition
- Testing SQL statements for correctness and performance against the database
- Manually modifying generated beans and SQL statements

9.3.2 Advanced pureQuery bean assembly using pureQuery Handlers

pureQuery provides entry points for advanced, user-defined, object-relational mapping by supporting customized row, result, stored procedure result, and parameter handlers.

A pureQuery RowHandler can be implemented by the data access layer developer to process one row of a query result at a time, using any implementation the developer defines. If the RowHandler is specified for an annotated or inline method, the pureQuery object-relational mapping layer will then automatically use the provided custom handler to process each row of the ResultSet.

In contrast, a pureQuery ResultHandler is designed to process an entire set of query results with one call to handle(), instead of only one row of the results with each call. A custom pureQuery ResultHandler can also be implemented by the user and specified for an annotated or inline method. Again, the pureQuery object-relational mapping layer will use the user-supplied handler instead of its own internal implementation to perform the mapping.

Implementing the CallHandlerWithParameters interface allows the user to define custom mappings for stored procedure results and parameters. These types of handlers are provided to the pureQuery annotated and inline methods which call stored procedures.

The above handlers support mapping from database results to the object layer. The mapping direction is reversed when parameters are provided from bean objects as input parameters to SQL statements. Mapping from bean fields to SQL statement parameters can be customized using a pureQuery ParameterHandler provided to pureQuery annotated methods using the @Handler annotation. If a ParameterHandler is provided in this manner, the pureQuery object-relational layer will use it to perform mapping instead of using its default mapping implementation.

pureQuery delivers example implementations of non-default ResultHandler interfaces to demonstrate the object-relational mapping flexibility that is possible with user-defined handlers. With a user-defined handler, the pureQuery object-relational mapping layer still handles the database access and much of the rote processing. However the designer is free to implement any type of mapping required for the specific application.
See Chapter 13 for more information about implementing custom handlers.

### 9.3.3 Query over Collections with pureQuery

pureQuery supports transparent, mixed, in-memory vs. relational queries using standard SQL. This feature is called Query over Collections, and allows the object-relational layer to be implemented using a hybrid approach dividing query processing between a traditional relational database backend and in-memory objects.

Depending on the kind of processing that is required, there may be systems which can be optimized by pulling a large, initial collection of data from the database using a traditional database query, then closing the database connection and performing several further, filtering queries on the initial collection in-memory. The advantage of this scheme is that the initial database connection resources can be freed up for other clients. Likewise, if the same set of initial data will have a number of filtering queries performed on it, the ability to query over the initial collection in-memory cuts the cost of additional trips to the database for each subsequent sub-query.

To query over collections, pureQuery accepts a parameter marker in the place of the table name in any SQL query using SQL-92 standard syntax. The application then provides a java collection for the table parameter to the annotated or inline pureQuery method which performs the query.

In Listing 3.4, a java List containing the contents of a previous database query is provided as a parameter to a pureQuery query-over-collection annotated method. In this example, the pureQuery numbered parameter marker ("?1") represents a table in the SQL query syntax. pureQuery’s implementation for this method will treat the List of Customers as the table over which to perform the query.

```java
import com.company.Customer;
import com.company.Campaign.CitySize;
public interface RegionReport {
    @Select(sql="SELECT COUNT(DISTINCT city) FROM ?1")
    public Integer countCities (List<Customer> curRegion);
}
```

**Listing 9.4 - An example of a Query Over Collection, in this case a List of Customers.**

### 9.3.4 XML entity mapping with pureQuery

One common way to increase the flexibility of the object-relational mapping layer is to separate the object-relational mapping definition from the application code. pureQuery supports XML entity-mapping files for annotated methods defined for pureQuery data access objects. See Chapter 11 to learn how to create pureQuery data access objects with user-defined, annotated methods using a pureQuery development project. To use XML entity mapping with pureQuery, an XML object-relational mapping file is provided to the pureQuery tooling along with the pureQuery annotated interface used to generate the class
definition. If the entity-mapping XML file is present, and it contains entity-mappings entries for bean classes used by methods in the interface, then the pureQuery generator will use the mappings defined in the XML file to generate the object-relational mapping code for the pureQuery annotated method.

The advantage of isolating the mapping directives outside of the java source is that it further compartmentalizes the design. It also provides a way to override old hard-coded or default mappings when some other part of the design changes, without having to change the java source. The changes can be isolated to the XML file.

Listing 3.5 shows an example snippet from an XML entity-mapping file defining the field-to-column mappings for an example pureQuery bean class, Customer:

```xml
<entity class="Customer">
  <attributes>
    <basic name="cid">
      <column name="CUSTID" />
    </basic>
    ...
    <basic name="province">
      <column name="PROV" />
    </basic>
    ...
  </attributes>
</entity>
```

Listing 9.5 - The “entity” element specifies attribute mapping for the Customer class.

### 9.3.5 Overriding SQL Statements in Applications

To support flexibility, pureQuery provides several ways to override SQL statements executed by the object-relational mapping layer. Annotated methods and inline methods both support substitution of SQL. In addition, the pureQuery client optimizer feature supports SQL substitution for standard JDBC applications that are not implemented using the pureQuery API.

For pureQuery annotated methods, the preferred way to override SQL is to provide an XML ORM definition file to the pureQuery Generator tool. To override the SQL for a given annotated method, the XML file will contain a named-native-query/query element whose name matches the fully qualified method name (in the form `<package>.<class>#<method>`). This element will contain the override SQL for the method. The pureQuery object-relational layer that is subsequently generated will be implemented using the override SQL from the XML file. This feature is especially beneficial
if the same application is being generated to run against multiple database products, and the SQL must be slightly different for each database. A database-specific XML file can be provided with database-specific SQL overrides whenever the application is generated by the pureQuery tooling against a new database.

In the following example, the named-native-query/query element specifies override SQL for a pureQuery annotated method:

```
<named-native-query name="com.ibm.db2.pureQuery.CustomerData#getCustomers()">
  <![CDATA[select CUSTID, NAME, COUNTRY, STREET, CITY, PROV, ZIP, PHONE, INFO
                from PDQ.SC.CUSTOMER]]>
</query>
```

Listing 9.6 - The “named-native-query” element contains the SQL string for the CustomerData.getCustomers() method.

The SQL executed by pureQuery inline methods can also be overridden. To override SQL for inline methods, the application passes a `Map<Object, Object>` to the pureQuery DataFactory when it retrieves its Data object. The map keys are SQL Strings which, if executed on the Data, should be overridden. The map values are the SQL strings which should override them. The pureQuery framework will then automatically override SQL matching one of the keys whenever it is executed via an inline method on the Data object.

Finally pureQuery’s client optimizer feature provides a powerful tool to override SQL in plain, legacy JDBC applications. After using the client optimizer to capture dynamic JDBC statements executed by a JDBC application, and configuring the resulting pureQuery XML file, the application can be configured to perform a variety of different runtime substitutions, without altering the legacy application code. For example, the application can be configured in such a way that some of the previously dynamic SQL statements can be bound at the database server and executed statically. Likewise, it’s also possible to configure basic SQL string substitution for any of the captured statements. For detailed information on the pureQuery Client Optimizer, see Chapter 6.

### 9.4 Summary

Persistence and object-relational mapping are multifaceted problems too complex for a one-size-fits-all solution. pureQuery provides a platform of integrated tooling, automation, and features to solve the most common problems associated with object-relational mapping. These tools can be used in a standalone mode or in integration with existing persistence frameworks.

pureQuery provides designers with an object-relational mapping implementation that is easy to use, reduces development errors, and reduces the amount of tedious code which must be written by hand. At the same time, it provides the flexibility to customize the
object-relational mapping layer to respond to increasingly complex system requirements, while still supporting the necessary performance behavior in the relational database and application layers. pureQuery is flexible enough to support complex mapping design. At the same time, its default object-relational mapping implementation is quick to produce using pureQuery tooling and a live database schema.

For systems with complex mapping requirements, pureQuery provides support for common and standardized object-relational mapping specifications such as XML entity-mapping definitions, and fully customizable object-relational mapping handlers which are pluggable into the default implementation when needed.

Finally, once the pureQuery object-relational mapping layer is launched into production, the pureQuery framework supports effective monitoring, maintenance, and problem determination across the object-relational layer. In doing so, pureQuery opens a much-needed window in the wall between the object and relational paradigms, supporting more effective design and problem-solving across the object-relational divide.

9.5 Review Questions

1. Explain the advantage of using a persistence framework?
2. How are object-relational mappers related to persistence frameworks?
3. True or false: pureQuery is a persistence framework?
4. What are two factors which complicate object-relational mapping design?
5. True or false: pureQuery supports only simple object to table mapping?
6. Which of the following persistence frameworks does pureQuery NOT specifically integrate with?
   A. JDO
   B. JPA
   C. Hibernate
7. Which of the following are advanced object-relational mapping tools supported by pureQuery?
   A. XML entity mapping files
   B. ResultHandlers and RowHandlers
   C. SQL Replacement
   D. All of the above
8. Which pureQuery annotation is used on a bean field?
   A. @ColumnOverrides
Chapter 9 – Persistence Frameworks and Data Access: Context for pureQuery 181

B. @Sql
C. @Column
D. All of the above

9. What is the meaning of the @GeneratedKey pureQuery annotation?
   A. The value will be generated by the database for INSERT statements and populated in the bean prior to return from the pureQuery method
   B. The field will be generated automatically by the Java framework
   C. The field should be populated by the user prior to using the bean for an INSERT statement
   D. None of the above

10. What is the name of the development tool that includes features to ensure SQL in the object-relational mapping layer is written quickly and correctly from the start?
    A. Query Over Collections
    B. Code Generation
    C. Extended Insight
    D. Optim Development Studio
Chapter 10 - Inline Programming Style

In this chapter, you will learn how to use the pureQuery inline style.

10.1 Inline programming style: The big picture

The pureQuery inline-style is an API for easily and quickly specifying and executing SQL statements in your application. The API is designed to allow you to quickly write code that specifies SQL statements, executes them, and gets the results in a useful format. 

pureQuery provides an interface named com.ibm.pdg.runtime.Data that has methods you can use to execute a wide variety of SQL statements. The methods allow you to specify the values of the parameters in your SQL statements. They return the results of SQL statements in a variety of formats, making it simple to start using the results as soon as the method calls return. Listing 10.1 shows an example of inline-style code that executes two SQL statements.

```java
Data data = DataFactory.getData (connection);
List<DepartmentBean> allDepartments = data.queryList ("SELECT * FROM DEPARTMENT", DepartmentBean.class);

// Iterate through the departments...
...
DepartmentBean department = data.queryFirst ("SELECT * FROM DEPARTMENT WHERE DEPTNO = ?", DepartmentBean.class, departmentNumber);

// Do something with the department...
...
```

Listing 10.1 - A simple example of using the inline method style to execute two SQL statements

Figure 10.1 shows how you can create an instance of the Data interface.
Figure 10.1 - Creating an instance of Data for the pureQuery inline style

Figure 10.2 shows how you can use the inline methods that are available on an instance of Data to run SQL statements and get results in a wide variety of formats.
Figure 10.2 - The inline methods available through an instance of the Data interface

In Figure 10.2, RowType represents the format in which each row of a query result is returned. There are a number of ways to specify this. ResultType represents the format that is specified by the parameterized type RES of a com.ibm.pdq.runtime.handlers.ResultHandler<RES> object. CallResultType represents the format that is specified by the parameterized type CAL of a com.ibm.pdq.runtime.handlers.CallHandlerWithParameters<CAL> object. Chapter 13 discusses pureQuery handlers.

There are several benefits to using the inline programming style, including:

- You can speed-up your application development. The methods in the Data interface do most of the work for accessing the data. All you have to do is call methods with your choice of parameters and return types, and
write the SQL statements for the methods to execute. pureQuery takes care of the rest of the work.

- You can choose from a wide variety of return types for the query results. A few of the possible formats are a single pureQuery bean, a single `java.util.Map`, a collection of pureQuery beans, and a collection of `Maps`.
- Complex queries can be prototyped and tuned directly within the context of the Java editor, and pureQuery will create the appropriate data bean to match the resultset of your query.
- For many developers this supports a more iterative approach to development, allowing them to refine their data access logic as they flesh out the other aspects of their application.

### 10.2 Instances of the `com.ibm.pdq.runtime.Data` Interface

In the pureQuery inline style, you execute SQL statements by invoking methods in the `com.ibm.pdq.runtime.Data` interface. Therefore, in order to use the inline-programming style, you must first create an instance of `Data`. You can use a single instance of `Data` to execute multiple SQL statements.

To create an instance of `Data`, use one of the methods provided in the class `com.ibm.pdq.runtime.factory.DataFactory`. The `DataFactory` class has many methods named `getData`. As the name would suggest, each of these methods returns an instance of `Data`. Different `getData` methods are useful in different circumstances. For the inline style, choose a method that does not have a `Class<T>` `interfaceClass` parameter. To create an instance of `Data` that you can use to run SQL statements against a database, choose a method that has either a `javax.sql.DataSource` parameter or a `java.sql.Connection` parameter.

Instances of the `Data` interface generally contain an instance of `java.sql.Connection`. The `Data` interface provides the methods `commit()`, `rollback()`, `setAutoCommit(boolean)`, `getAutoCommit()`, and `close()`, all of which call the similarly-named method on the `Data` instance contained `Connection` object. Use these methods on the instance of `Data` to manage the `Connection` object and to perform transaction management for the SQL executed that the instance executes.

### 10.2.1 In practice: Overview of the “In practice” sections in this chapter

In this chapter, the “In practice” sections modify the class `hrm.inProgress.pdqInlineStyle.HumanResourcesManager` for the Human Resources Manager application. This file is in the `GettingStartedWithPureQuery` project that is distributed with this book. Appendix B provides an overview of this
application. Most of the “In practice” sections ask you to add a line or two of pureQuery code to either the class or the interface. Look for the comments that say TODO: Add pureQuery code here. to know where to add the code.

10.2.2 In practice: Create a Data object in the displayLogin() method

The Human Resources Manager application creates a single instance of Data in the displayLogin() method and passes that instance to other methods, as required. Finally, before the application exits, it closes the instance of Data. Modify the displayLogin() method so that it creates and closes an instance of Data.

10.2.2.1 Add code to the displayLogin() method to create the Data object

Since we are using the inline programming style, we need a getData method that does not take a Class<T> interfaceClass parameter. We will be using a database, and we want pureQuery to create a connection by using the DataSource object that is in hrm.HRMDatasourceFactory, so we need a method that has a javax.sql.DataSource parameter. We choose the method shown in Listing 10.2.

```java
DataFactory.getData (DataSource ds)
```

**Listing 10.2 - The getData method that we will use to create an instance of Data**

The displayLogin() method contains a declaration for a variable named data. Add code that creates an instance of Data and assigns the result to data. Look for a comment that says TODO: Add pureQuery code here. to know where you need to add the new code. Listing 10.3 shows the updated version of the code.

```java
// Get the DataSource, and use it to create an instance of Data.
DataSource dataSource = HRMDatasourceFactory.getDatasourceForApplication ();
data = DataFactory.getData (dataSource);
```

**Listing 10.3 - Add code to the displayLogin() method that creates an instance of Data**

10.2.2.2 Close the Data object when your application no longer needs it

Modify the finally block in displayLogin() to close the instance of Data. This closes the java.sql.Connection object that pureQuery created for the instance of Data. Listing 10.4 shows the code that you need to add.

```java
// Close the instance of Data. This closes the
// database connection that it contains.
if (null != data) {
    data.close ();
}
```

**Listing 10.4 - Add code to the displayLogin() method that closes the instance of Data**
10.2.2.3 Test the application
Now, run `hrm.inProgress.pdqInlineStyle.HumanResourcesManager`. When the application prompts you for an employee number, use 000030. When the main menu displays, immediately select the menu option 0 to exit the application. We have not finished implementing the other screens yet, so they will not work.

10.3 Executing SQL SELECT statements
The Data interface provides many `queryXXX` methods for executing SQL SELECT statements. Choose the one you want to use based on the format in which you want to retrieve your data and on any other parameters that you want to be able to specify.

10.3.1 Choosing the return type
Use the following questions as a guide to help you select the right return type.

What format do you want to use for each row?

- A pureQuery bean or a scalar object: `queryXXX` methods that have a parameter `Class<ROW> returnClass` store the contents of each row in an instance of `returnClass`.
  - If `returnClass` is a pureQuery bean, pureQuery creates an instance of `returnClass` and sets the values of its properties according to the values of the columns returned from the database. pureQuery uses a consistent set of mapping rules to determine the bean property that represents each column that is selected from the database. When pureQuery cannot find a property that represents a column in the SQL statement, pureQuery does not store the value of that column in the bean.
  - If `returnClass` is not a pureQuery bean, pureQuery constructs an object of type `returnClass` and attempts to store the contents of the first column of the query result in that object.

- An object of a specific type, constructed by a `com.ibm.pdq.runtime.handlers.RowHandler<ROW>` object: `queryXXX` methods that have a parameter `RowHandler<ROW> rowHandler` use `rowHandler` to create the object that contains each row. Chapter 13 discusses handlers.

- A `java.util.Map<String, Object>` object: `queryXXX` methods that do not have a parameter `Class<ROW> returnClass` or a parameter `RowHandler<ROW> rowHandler` return the contents of each row in a `Map<String, Object>` object, in which the keys are the column labels, and the values are the contents of the columns.
What object do you want to use to contain all of the rows?

- An array: `queryArray` methods return an array in which each entry represents a row.
- A `java.util.List<E>` object: `queryList` methods return a `java.util.List<E>` object in which each element represents a row.
- A `java.util.Iterator<E>` object: `queryIterator` methods return a `java.util.Iterator<E>` object for which each call to the `Iterator`'s `next()` method returns a row. These methods do not fully materialize the information from the database during the initial method call. Instead, pureQuery retrieves one row each time the application calls the `next` method. This is particularly useful if your application retrieves large amounts of information from the database, and you do not want to store all of the information in memory at one time. When the application calls `next()` to retrieve all of the rows from the `Iterator`, pureQuery closes it automatically, freeing the associated database resources. However, if you have an application that does not retrieve all the rows, you must free the database resources by closing the `Iterator<E>` explicitly. Do so by casting the `Iterator<E>` object as a `com.ibm.pdq.runtime.ResultIterator<T>` object and calling its `close()` method, like this: `((com.ibm.pdq.runtime.ResultIterator<T>) myIterator).close()`.
- Only the first row, in the object used to contain the row: `queryFirst` methods return only the first row. If the query result does not contain any rows, these methods return `null`.
- An object of a specific type, constructed by a `com.ibm.pdq.runtime.handlers.ResultHandler<RES>` object: `query` methods use the parameter `ResultHandler<RES> resultHandler` to construct the object that is returned. For this method, there is not a parameter for specifying the format for each row. Instead, `resultHandler` determines the format that is used for each row.
- A `java.sql.ResultSet` object: `queryResults` methods return an instance of `java.sql.ResultSet`.

### 10.3.2 Overriding the default cursor attributes

By default, inline methods use the cursor type “forward only,” the cursor concurrency mode “read only,” and the cursor holdability “close cursors at commit.” You can change the values of these attributes by using an inline method that has these parameters: `int type, int concurrency, int holdability`
10.3.3 In practice: Execute a simple SELECT statement that uses cursor attributes -- Complete the “Company Employees” screen

The Company Employees screen displays all of the employees in the company in pages of 10 employees each. It allows the user to request additional information about any particular employee on the screen. The displayAllEmployees (Data data) method creates this screen. Add the pureQuery code to it to get the employee information from the database.

10.3.3.1 Use pureQuery to create an Iterator<EmployeeBean> object that contains the information about the employees

In order to get information about all the employees in the company, we need to execute a query against the database. We will use the query shown in Listing 10.5, which will select all the employees and sort them by their names.

```
SELECT * FROM EMPLOYEE ORDER BY LASTNAME, FIRSTNAME, MIDINIT
```

Listing 10.5 - An SQL query for selecting all the employees in the company

Next, we need to determine which Data method to use. Since we need several pieces of information about each employee, it would be convenient to retrieve each row as an instance of EmployeeBean. We therefore need to use a queryXXX method that has a Class<ROW> returnClass parameter. We will be selecting all the employees in the database, but we would prefer not to have to store all of the company's employees in the application's memory at once, since that could be quite a lot of memory. As a result, an Iterator<E> is the most logical object to use for returning the rows. Therefore, we will use a queryIterator method that has a Class<ROW> returnClass parameter.

While users are viewing the list of employees, they can request to view additional information about an employee and then return to the list. Therefore, we need to set the cursor holdability to “hold cursors over commit.” As a result, we use a queryIterator method that has parameters for cursor attributes.

The code to execute our SQL query against the database and create the Iterator is simple. All we have to do is invoke the queryIterator method, specifying the SQL statement and EmployeeBean.class. This returns an Iterator<EmployeeBean> object, where each item is an instance of EmployeeBean. Modify the application to contain this code. Remember to look for a comment that says TODO: Add pureQuery code here. so that you will know where you need to add the new code. Listing 10.6 shows the updated code.

```java
// Search the database for all of the employees in the company
// and create an Iterator<EmployeeBean> object to retrieve them.
// Use "hold cursors over commit" so that if the user goes to another
// screen and comes back, we can continue displaying employees.
employees = data.queryIterator (ResultSet.TYPE_FORWARD_ONLY,
        ResultSet.CONCUR_READ_ONLY, ResultSet.HOLD_CURSORS_OVER_COMMIT,
```
Chapter 10 – Inline Programming Style

"SELECT * FROM EMPLOYEE ORDER BY LASTNAME, FIRSTNAME, MIDINIT",
EmployeeBean.class); 

Listing 10.6 - Modify the displayAllEmployees method so that it creates an
Iterator<EmployeeBean> for all the employees

10.3.3.2 Close the Iterator<EmployeeBean> object to free the associated database
resources

The user of Human Resources Manager might not view all of the employees. Therefore,
we need to close employees explicitly to free the database resources. Listing 10.7 shows
the code that you should add to close employees.

finally {
  // Close employees. This frees the associated database resources.
  if (null != employees)
    ((ResultIterator<EmployeeBean>) employees).close();
}

Listing 10.7 - Add code to the displayAllEmployees method that closes the
Iterator<EmployeeBean> when the application no longer needs it

10.3.3.3 Test the "Company Employees" screen

Save and test your application. As before, supply 000030 when the Human Resources
Manager asks for your employee number. On the main menu, select option 4 to display all
the employees in the company. You will then be able to see the employees, one page at a
time.

10.4 Executing SQL statement that have parameters

pureQuery makes it easy to use parameters in SQL statements. pureQuery provides four
types of parameter markers, and with all four of these types, it automatically populates the
parameters with the information that you provide. When you use multiple parameter
markers in an SQL statement, all of the parameter markers must be of the same type.

In the inline method style, the Data methods have varargs parameters object…
parameters. You specify the objects that pureQuery should use to populate the
parameters of your SQL statement as the value of parameters.

Table 10.1 shows the four types of parameter markers. References to Map indicate the
class java.util.Map. Listing 10.8 shows the definitions of the objects that the “Sample
usage” column references.

If the functionality of pureQuery’s parameter markers does not meet the needs of your
application, you can write an implementation of the
com.ibm.pdq.runtime.handlers.ParameterHandler interface to set the
parameters appropriately.

// Scalars
String deptNo = "C01";
String managerEmpNo = "000030";

// pureQuery bean
DepartmentBean dept = new DepartmentBean();
department.setDepartmentNumber ("C01");
department.setManagerEmployeeNumber ("000030");

// Map
Map<String, String> map = new HashMap<String, String> ();
map.put ("departmentNumber", "C01");
map.put ("managerEmployeeNumber", "000030");

Listing 10.8 - Definitions of the objects referenced in the "Sample usage" column of Table 10.1

<table>
<thead>
<tr>
<th>Marker</th>
<th>Types of objects in parameters</th>
<th>Meaning</th>
<th>Sample usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>Scalars</td>
<td>Directs pureQuery to use the value of one of the objects in parameters as the value of this SQL parameter. The first ? represents the first object, the second ? represents the second object, and so forth.</td>
<td>List&lt;EmployeeBean&gt; employees = data.queryList (&quot;SELECT * FROM EMPLOYEE WHERE WORKDEPT = ? AND EMPNO != ?&quot;, EmployeeBean.class, deptNo, managerEmpNo);</td>
</tr>
<tr>
<td>?n</td>
<td>Scalars</td>
<td>Directs pureQuery to use the value of the n-th object in parameters as the value of the SQL parameter. ?1 represents the first object, ?2 represents the second object, and so forth.</td>
<td>List&lt;EmployeeBean&gt; employees = data.queryList (&quot;SELECT * FROM EMPLOYEE WHERE WORKDEPT = ?1 AND EMPNO != ?2&quot;, EmployeeBean.class, deptNo, managerEmpNo);</td>
</tr>
<tr>
<td>?n.name</td>
<td>pureQuery beans and Map objects</td>
<td>Directs pureQuery to use the value of the property name in the nth object in parameters as the value of the SQL parameter. For example, ?1.lastName represents the lastName property of first object. The property name is case sensitive.</td>
<td></td>
</tr>
<tr>
<td>:name</td>
<td>pureQuery beans and Map objects</td>
<td>Directs pureQuery to use the value of the property name in the first object in parameters as the value of the SQL parameter. This is a shorter way of specifying ?1.name that is useful when only one pureQuery bean or Map object is specified. The property name is case sensitive.</td>
<td></td>
</tr>
</tbody>
</table>

Table 10.1 - Parameter markers in the inline-method programming style

10.4.1 In practice: Specify a scalar as a parameter to an SQL statement--Complete the “Login” screen

When Human Resources Manager starts, it asks the user for his or her employee number. The application queries the database for information about the employee with that employee number. If the database contains an employee with that employee number, it displays the main menu. If it does not contain such an employee, the application exits.

The original code provided in the displayLogin method of hrm.inProgress.pdqInlineStyle.HumanResourcesManager does not query the database for the employee; instead, it just creates an empty EmployeeBean object. This is shown in Listing 10.9.
// Search the database for an employee with the employee number that the
// user entered.
// TODO: Add pureQuery code here.
employeeUsedForLogin = new EmployeeBean();
Listing 10.9 - Original assignment of employeeUsedForLogin in the
displayLogin() method of
hrm.inProgress.pdqInlineStyle.HumanResourcesManager

Modify the code so that it queries the database for the employee information.

10.4.1.1 Use pureQuery to create an EmployeeBean object that contains the
information about the employee

We will use the SQL statement in Listing 10.10 to get the information about the employee
with the specified employee number. The text in square brackets indicates an SQL
parameter that pureQuery needs to populate with the employee number that the user
enters.

SELECT * FROM EMPLOYEE WHERE EMPNO = [employee number]
Listing 10.10 - An SQL query for selecting the employee with the specified employee
number

Because we have the employee number in a String, we can use the ? or the ?n style of
parameter marker. We choose to use the ? style. We will specify the employee number as
the first (and only) object in the varargs parameters to the method.

We need to use a method in the Data interface to execute our SQL statement. We want
the employee to be returned in an EmployeeBean object, so we want to use a queryXXX
method in which we can specify the EmployeeBean.class. The EMPNO column is the
primary key of the EMPLOYEE table, so there will be at most one row that matches
the employee number that the user enters. Therefore, we choose a queryFirst method.
Modify the application code as shown in Listing 10.11 to use this method to create the
EmployeeBean object.

// Search the database for an employee with the employee number that the
// user entered.
employeeUsedForLogin = data.queryFirst {
"SELECT * FROM EMPLOYEE WHERE EMPNO = ?", EmployeeBean.class,
employeeNumber);
Listing 10.11 - Modify the displayLogin() to create an EmployeeBean object for
the employee that logged in
10.4.2 In practice: Specify a pureQuery bean as a parameter to an SQL statement-- Complete the “Employee Information” screen

The “Employee Information” screen displays detailed information for a single employee. The displayEmployeeInformation method has an EmployeeBean parameter that provides details about the employee. The screen also displays summary information about the manager of the employee. Information about an employee’s manager is not included in an EmployeeBean object. To get information about the employee’s manager, the displayEmployeeInformation method calls a utility method named selectEmployeeManager. That method has an EmployeeBean parameter. It returns another EmployeeBean object representing the employee’s manager, or null if the employee does not have a manager. Add pureQuery code to the selectEmployeeManager method to query the database for information about the employee’s manager.

10.4.2.1 Use pureQuery to create an EmployeeBean object that contains the information about an employee’s manager

We next need an SQL query to determine the manager of an employee. Note that the EMPLOYEE table contains a WORKDEPT column, which contains the department number of the department in which the employee works. The DEPARTMENT table contains a MGRNO column, which contains the employee number of the manager of the department. An ADMRDEPT column contains the department number of the department that administers it. Therefore, a non-manager employee reports to the manager of his or her department. An employee who is a manager reports to the manager of the department that administers his or her department. Listing 10.12 shows the SQL query that we will use to select the information about the employee who manages the specified employee. The text in square brackets indicates parameters that will need to be set from the user’s information.

"SELECT e.EMPNO, e.FIRSTNAME, e.MIDINIT, e.LASTNAME, e.WORKDEPT, " + "e.PHONENO, e.HIREDATE, e.JOB, e.EDLEVEL, e.SEX, e.BIRTHDATE, " + "e.SALARY, e.BONUS, e.COMM FROM " + "(EMPLOYEE e INNER JOIN DEPARTMENT d ON (e.WORKDEPT = d.DEPTNO)) WHERE " + "// The employee is not his or her own manager. " + "(e.EMPNO <> [employee’s employee number]) AND " + "// If the employee is not the manager of his or her department, find the manager of his or her department. " + "((0 = (SELECT COUNT(*) FROM DEPARTMENT WHERE " + "MGRNO = [employee’s employee number] " + "AND DEPTNO = [employee’s department number])) " + "AND (e.WORKDEPT = [employee’s department number]) " + "AND (e.EMPNO = d.MGRNO))") AND " + "// If the employee is the manager of his or her department, find the manager of the department that administers it. " + "OR ((1 = (SELECT COUNT(*) FROM DEPARTMENT WHERE " + "MGRNO = [employee’s employee number] " + "AND DEPTNO = [employee’s department number])) " + "AND (e.WORKDEPT = [employee’s department number]) " + "AND (e.EMPNO = d.MGRNO))") AND "
+ "MGRNO = [employee’s employee number] "
+ "AND DEPTNO = [employee’s department number] "
+ "AND e.WORKDEPT = (SELECT ADMRDEPT FROM DEPARTMENT "
+ "WHERE DEPTNO = [employee’s department number] AND e.EMPNO = d.MGRNO))"

Listing 10.12 - An SQL statement that determines the manager of the employee that
is specified

We want the method to return the manager's information in an EmployeeBean object, and
there will be at most one row, so we choose to use the queryFirst method (which
executes faster when we only need one row) in which we can specify the class
EmployeeBean.class.

We will use pureQuery parameter markers for the parameters to the SQL. We need the
employee number and the department number for the employee. Both of these pieces of
information are available in the EmployeeBean object that is a parameter to the method.
Refer to Table 10.1, which lists the various types of parameter markers. Both the ?n.name
type of parameter marker and the :name type indicate that the information is a property in
a pureQuery bean. We choose to use the ?n.name type. The names of the appropriate
properties in EmployeeBean are employeeNumber and departmentNumber.
Therefore, the two parameter markers are ?1.employeeNumber and
?1.departmentNumber.

Modify the selectEmployeeManager method as shown in Listing 10.13 so that it
executes the query and return the result in an EmployeeBean object. Note how easy it is
to specify multiple parameters to an SQL statement!

```java
private EmployeeBean selectEmployeeManager (Data data, EmployeeBean employee) {
    return data.queryFirst("SELECT e.EMPNO, e.FIRSTNME, e.MIDINIT, 
        + "e.LASTNAME, e.WORKDEPT, e.PHONENO, e.HIREDATE, e.JOB, e.EDLEVEL, 
        + "e.SEX, e.BIRTHDATE, e.SALARY, e.BONUS, e.COMM FROM 
        + "(EMPLOYEE e INNER JOIN DEPARTMENT d ON (e.WORKDEPT = d.DEPTNO)) 
        + " WHERE " 
        // The employee is not his or her own manager. 
        + "(e.EMPNO <> ?1.employeeNumber) AND " 
        // If the employee is not the manager of his or her department, 
        // find the manager of his or her department. 
        + "((0 = (SELECT COUNT(*) FROM DEPARTMENT WHERE " 
        + "MGRNO = ?1.employeeNumber AND DEPTNO = ?1.departmentNumber)) " 
        + "AND (e.WORKDEPT = ?1.departmentNumber) AND (e.EMPNO = d.MGRNO)) " 
        // If the employee is the manager of his or her department, find the 
        // manager of the department that administers it. 
        + "OR ((1 = (SELECT COUNT(*) FROM DEPARTMENT WHERE " 
```

196 Getting Started with pureQuery
Chapter 10 – Inline Programming Style

Listing 10.13 - Add code to the `selectEmployeeManager` method to determine the manager of an employee

10.4.2.2 Test the "Company Employees" screen
Save and test your application. Supply 000030 when the Human Resources Manager asks for your employee number. On the main menu, select option 1 to display information about Sally Kwan.

10.5 Executing SQL statements that modify information in a database
The pureQuery inline-method style provides methods named `update` and `updateMany` for executing SQL statements that modify the information stored in database tables. These methods can execute SQL statements that do not return query results, such as `INSERT`, `UPDATE`, and `DELETE`.

The `Data` interface contains two `update` methods for executing a single SQL statement one time, as well as three methods for updating an SQL statement multiple times.

10.5.1 Executing an SQL statement one time by using the method: `int update (String sql, Object... parameters)`
The method `int update (String sql, Object... parameters)` executes the specified SQL statement and returns an `int` indicating how many rows were updated.

Listing 10.14 shows an example of using this method.

```
EmployeeBean newEmployee = new EmployeeBean();
newEmployee.setEmployeeNumber("010010");
newEmployee.setFirstName("FirstName");
newEmployee.setLastName("LastName");
newEmployee.setEducationLevel(18);
int updateCount = data.update("INSERT INTO EMPLOYEE (EMPNO, FIRSTNME, LASTNAME, EDLEVEL)
VALUES (:employeeNumber, :firstName, :lastName, :educationLevel)",
newEmployee);
System.out.println("Update count: " + updateCount);
```

Listing 10.14 - An example of executing the update method
10.5.2 Executing an SQL statement with generated keys one time by using the method: \texttt{ROW update (String sql, Class<ROW> returnClass, String[] columnNames, Object... parameters)}

The method \texttt{ROW update (String sql, Class<ROW> returnClass, String[] columnNames, Object... parameters)} is intended for SQL statements that have generated keys. It executes the specified SQL statement and, in some cases, returns the contents of the columns whose names are specified. If the SQL statement is an \texttt{UPDATE} or \texttt{DELETE} statement, the method returns \texttt{null}. If the SQL statement is an \texttt{INSERT} statement, it returns an object of type \texttt{returnClass} with the contents of the specified columns from the inserted row.

When a method returns an object of the type \texttt{returnClass}, the object's contents depend on the class. If the class is \texttt{Object[].class}, then all but the last item in the array contain the value of one of the specified columns. If the class is a scalar, pureQuery attempts to return the contents of the first specified column in an object of that type.

You can specify column names in one of two ways. If \texttt{columnNames} is not \texttt{null}, the column names that are used are those listed in the \texttt{columnNames} array. If \texttt{columnNames} is \texttt{null}, and the first parameter of \texttt{Object... parameters} is a pureQuery bean, the column names that are used are the names corresponding to the properties in the bean that have the \texttt{com.ibm.pdq.annotation.GeneratedKey} annotation.

For examples of using this method, assume that we have a table \texttt{EMPLOYEE_GEN_KEY} that has a generated column, defined as in Listing 10.15. We will use the class \texttt{EmployeeBean} to represent this table. We add the \texttt{@GeneratedKey} annotation to the \texttt{employeeNumber} property, as shown in Listing 10.16. There are four examples of using this method. All of them use the \texttt{EmployeeBean} object defined in Listing 10.17.

\begin{verbatim}
CREATE TABLE EMPLOYEE_GEN_KEY (EMPNO INT GENERATED ALWAYS AS IDENTITY (START WITH 10, INCREMENT BY 10), FIRSTNME VARCHAR(12), LASTNAME VARCHAR(15))

Listing 10.15 - The DB2 definition of the EMPLOYEE_GEN_KEY table
\end{verbatim}

\begin{verbatim}
@GeneratedKey
@Column(name = "EMPNO")
public String getEmployeeNumber ()
{
    return employeeNumber;
}

public void setEmployeeNumber (String employeeNumber)
{
    this.employeeNumber = employeeNumber;
}
\end{verbatim}
Listing 10.16 - The @GeneratedKey annotation on the employeeNumber property in EmployeeBean

EmployeeBean newEmployee = new EmployeeBean();
newEmployee.setFirstName("FirstName");
newEmployee.setLastName("LastName");

Listing 10.17 - The instance of EmployeeBean that is used by the examples

The following four examples show different ways of using this method. In all of the examples, the first parameter of Object... parameters is an instance of the pureQuery bean EmployeeBean named newEmployee.

The examples in Listing 10.18 and Listing 10.19 provide the column names by using the columnNames method parameter. The examples in Listing 10.20 and Listing 10.21 do not provide columnNames, so the properties in EmployeeBean provide the column names.

The examples in Listing 10.19 and Listing 10.21 specify Object.class as the value of returnClass, so they return an Object array in which the first item contains the value of the EMPNO column, and the second item contains the update count. The examples in Listing 10.18 and Listing 10.20 do not specify Object.class as the value of returnClass, so they return the contents of the first column from the column names.

Integer newEmployeeNumber = data.update(
    "INSERT INTO EMPLOYEE_GEN_KEY (FIRSTNAME, LASTNAME) VALUES (:firstName, :lastName)",
    Object[].class, new EmployeeBean()
);

System.out.println("New employee number: " + newEmployeeNumber);

Listing 10.18 - An example of executing the update method with Integer.class as the value of returnClass and with the column names specified by the String[] columnNames

Object[] updateOutput = data.update(
    "INSERT INTO EMPLOYEE_GEN_KEY (FIRSTNAME, LASTNAME) VALUES (:firstName, :lastName)",
    Object[].class, new EmployeeBean()
);

System.out.println("New employee number: " + updateOutput[0]);
System.out.println("Update count: " + updateOutput[1]);

Listing 10.19 - An example of executing the update method with Object[].class as the value of returnClass and with the column names specified by the String[] columnNames

Integer newEmployeeNumber = data.update(
    "INSERT INTO EMPLOYEE_GEN_KEY (FIRSTNAME, LASTNAME) VALUES (:firstName, :lastName)",
    Integer.class, null, newEmployeeBean()
);

System.out.println("New employee number: " + newEmployeeNumber);
Listing 10.20 - An example of executing the `update` method with `Integer.class` as the value of `returnClass` and with the column names specified by the properties of `EmployeeBean`

```
Object[] updateOutput = data.update {
   "INSERT INTO EMPLOYEE_GEN_KEY (FIRSTNME, LASTNAME) VALUES (:firstName, :lastName)"
   Object[].class, null, newEmployee);
System.out.println ("New employee number: " + updateOutput[0]);
System.out.println ("Update count: " + updateOutput[1]);
```

Listing 10.21 - An example of executing the `update` method with `Object[].class` as the value of `returnClass` and with the column names specified by the properties of `EmployeeBean`

### 10.5.3 Executing an SQL statement multiple times

The `Data` interface contains three `updateMany` methods. Each of these methods has two parameters: an SQL statement and a collection. Each entry in the collection must contain all the parameters for the SQL statement. `pureQuery` executes the SQL statement one time for each entry in the collection, each time using that entry to set the values of the SQL parameters. Each method returns an `int[]`, in which each item contains the update count for the corresponding set of parameters in the collection.

- `int[] updateMany (String sql, Iterable<T> parameters)`
- `int[] updateMany (String sql, Iterator<T> parameters)`
- `int[] updateMany (String sql, T[] parameters)`

Listing 10.22 shows an example of using an `updateMany` method.

```
EmployeeBean[] newEmployees = new EmployeeBean[2];
// First employee
newEmployees[0] = new EmployeeBean();
newEmployees[0].setEmployeeNumber ("010010");
newEmployees[0].setFirstName ("FirstName1");
newEmployees[0].setLastName ("LastName1");
newEmployees[0].setEducationLevel (18);
// Second employee
newEmployees[1] = new EmployeeBean();
newEmployees[1].setEmployeeNumber ("010020");
newEmployees[1].setFirstName ("FirstName2");
newEmployees[1].setLastName ("LastName2");
newEmployees[1].setEducationLevel (18);
// Run the update
int[] updateCounts = data2.updateMany (};
"INSERT INTO EMPLOYEE (EMPNO, FIRSTNME, LASTNAME, EDLEVEL) " +
"VALUES (:employeeNumber, :firstName, :lastName, :educationLevel)",
newEmployees);

// Review the update counts
for (int i = 0; i < updateCounts.length; i++)
    System.out.println ("Update count for " + newEmployees[i].getFirstName
        + newEmployees[i].getLastName () + ": " + updateCounts[i]);

Listing 10.22 - An example of using an updateMany method

10.5.4 In practice: Execute an SQL statement that modifies the database -- Complete the "Change Employee Information" screen

The "Change Employee Information" screen allows the information in certain fields to be changed. The screen displays the editable fields and asks the user which one he or she would like to modify. The changeEmployeeInformation method controls the "Change Employee Information" screen. When a user asks to change a particular column, it updates the EmployeeBean object that contains the user's information with the new information, and then it uses the updated object to update the database. Add code to that method that updates the information in the database. The code should make sure that exactly one row was updated.

10.5.4.1 Use pureQuery to update the database

We will use a separate SQL statement for each column that a user can update. Two of the SQL statements are shown in Listing 10.23 and Listing 10.24. The remaining statements follow the same pattern.

```
UPDATE EMPLOYEE SET LASTNAME = [new last name] WHERE EMPNO = [original employee number]

Listing 10.23 - An SQL statement for updating an employee's last name

UPDATE EMPLOYEE SET FIRSTNME = [new first name] WHERE EMPNO = [original employee number]

Listing 10.24 - An SQL statement for updating an employee’s first name
```

Next, we need to choose the Data method that we will use to execute the statement. Since we only have one SQL statement to execute at a time, there is no reason to use one of the updateMany methods. We want the method to return an update count, so we use the method shown in Listing 10.25.

```
int update (String sql, Object... parameters)

Listing 10.25 - The inline method that we will use to update an employee's information
```

We will use pureQuery parameter markers for the parameters to the SQL. All of the information about the employee will be available in the EmployeeBean object that describes the employee. Refer to Table 6.1, which lists the various types of parameter
markers. Both the \texttt{?n.name} type of parameter marker and the \texttt{:name} type indicate that the information is a property in a pureQuery bean. We choose to use the \texttt{:name} type.

Modify \texttt{changeEmployeeInformation} to update the database with the employee’s new information. You need methods that update each of the following columns: \texttt{LASTNAME}, \texttt{FIRSTNME}, \texttt{MIDINIT}, \texttt{EDLEVEL}, \texttt{BIRTHDATE}, \texttt{JOB}, \texttt{SALARY}, \texttt{BONUS}, and \texttt{COMM}. The corresponding properties in \texttt{EmployeeBean} are, respectively: \texttt{lastName}, \texttt{firstName}, \texttt{middleInitial}, \texttt{educationLevel}, \texttt{birthDate}, \texttt{job}, \texttt{salary}, \texttt{bonus}, and \texttt{commission}. Add a call to an inline method for each SQL statement. Two are shown in Listing 10.26 and in Listing 10.27; the remaining methods are similar. You can refer to the completed code in \texttt{hrm.completed.pdqInlineStyle.HumanResourcesManager} if you want help.

// Update the last name in the database.
updateCount = data.update ("UPDATE EMPLOYEE SET LASTNAME = :lastName WHERE EMPNO = :employeeNumber", employee);

\textbf{Listing 10.26 - An inline method that updates an employee's last name}

// Update the first name in the database.
updateCount = data.update ("UPDATE EMPLOYEE SET FIRSTNME = :firstName WHERE EMPNO = :employeeNumber", employee);

\textbf{Listing 10.27 - An inline method that updates an employee's first name}

Note how easy it is to modify information in the database when using pureQuery!

10.5.4.2 Test the "Change Employee Information" screen

Save and test your application. Supply \texttt{000030} to login as Sally Kwan. On the main menu, select option 2 to change information about Sally. Then, return to the main menu and select option 4 to go to the “Company Employees” screen and find Heather A. Nicholls, one of the employees that Sally manages. View information about Heather, and then change information for her.

10.6 Advanced features

The inline style provides many advanced features that are outside of the scope of this book. This list describes some of them. For details about using these features, refer to the IBM Optim pureQuery Runtime documentation.

- \textbf{Exception handling}: The inline style APIs and data access object annotated-methods throw exceptions of the type \texttt{com.ibm.pdq.runtime.exception.DataRuntimeException}. Utility methods on these exceptions provide helpful information about the causing exception.
Chapter 10 – Inline Programming Style

- **Hooks:** You can use implementations of the interface `com.ibm.pdq.runtime.statement.Hook` to provide methods that bracket the execution of the inline and annotated methods.

- **Batching SQL statements:** You can use heterogeneous batching to execute multiple different SQL statements in a single trip to the database.

- **Queries over collections:** You can use the SQL syntax to perform queries over in-memory collections of data.

- **Replacing SQL statements:** You can provide SQL statements to replace other SQL statements in your application. This is particularly useful if you have an application you execute against different database, and the syntax for some of the SQL statements varies between the databases. This feature is also useful if you want to replace an SQL statement with an optimized one without otherwise modifying your application. You can do this by specifying the statements in a `java.util.Map<Object,Object> sqlOverrides` parameter that you provide to a `DataFactory.getData` method.

10.7 Summary

The inline style is an API for accessing databases that is easy and quick to use. Under the inline style, you use the methods provided in the `com.ibm.pdq.runtime.Data` interface to execute SQL statements. The different methods execute different types of SQL statements and return query results in different formats. You specify the objects that you want to set the values of the parameters of the SQL statements as the parameters of these methods.

10.8 Review questions

1. In the inline method style, where do you specify an SQL statement that you want to execute?
2. How do you create an instance of the `com.ibm.pdq.runtime.Data` interface?
3. In an inline method, how do you specify the values of the parameters of SQL statement?
4. Which of the following parameter markers reference properties in beans?
   A. `?n`
   B. `:name`
   C. `?`
   D. `?n.name`
5. True or false: You can use more than one type of parameter marker in an SQL statement.

6. If you are executing an SQL `SELECT` statement, which of the following aspects should you consider when choosing a method in the `Data` interface to execute your statement?
   A. The object that you want to contain each row
   B. How many of the rows you want to return
   C. The object that you want to contain all of the rows
   D. All of the above

7. If you are going to execute an SQL query that is going to return a very large number of rows, which format should you probably use to return all the rows, and why?

8. True or false: If your application does not iterate through all of the rows returned through a `java.util.Iterator<E>` object, the application should close the Iterator to free the associated database resources?

9. If none of the objects that pureQuery provides for returning query results meet the needs of your application, what can you do?
   A. Use a `queryResults` method, which returns an instance of `java.sql.ResultSet`.
   B. Use a `com.ibm.pdq.runtime.handlers.RowHandler<ROW>` object to define the return type
   C. Use a `com.ibm.pdq.runtime.handlers.ResultHandler<RES>` object to define the return type
   D. Any of the above.

10. If you want to use a cursor type, holdability, or concurrency that is different from pureQuery’s default, how do you specify the attributes that you want to use?
    A. Specify the cursor attributes as parameters to the `getData` method.
    B. Specify the cursor attributes as parameters to the `queryXXX` method.
    C. Set the cursor attributes on the `Data` object.
    D. None of the above.

11. What is one way in which you can close the `java.sql.Connection` object associated with an instance of an instance of `Data`?

12. What are some of the benefits of using the inline style?
10.9 Exercise

As described in Appendix B, the “Employee Report Chain” screen displays the employees that report to a specified employee, as well as the manager of the specified employee. Add the necessary pureQuery code to hrm.inProgress.pdqInlineStyle.HumanResourcesManager to finish developing this screen.

- The method displayEmployeeReportChain creates the “Employee Report Chain” screen.
- Locate the code in Listing 10.28 in the displayEmployeeReportChain method. You need to add an inline method that queries the database for the employees that the specified employee directly manages. You can use the SQL statement in Listing 10.29. It selects all of the employees in his or her department, along with the managers of the departments that his or her department administers.

```java
// Search the database for all the employees that this employee
// directly manages and create a list of the employees managed by
// this employee.
// TODO: Add pureQuery code here.
List<EmployeeBean> managedEmployees = null;
Listing 10.28 - You need to add a call to an inline method that gets the employees managed by the specified employee

"SELECT e.EMPNO, e.FIRSTNAME, e.MIDINIT, e.LASTNAME, e.WORKDEPT, "
+ "e.PHONENO, e.HIREDATE, e.JOB, e.EDLEVEL, e.SEX, e.BIRTHDATE, "
+ "e.SALARY, e.BONUS, e.COMM "
// Join such that each row contains an employee and his or her
// department.
+ "FROM ((EMPLOYEE e INNER JOIN DEPARTMENT d ON e.WORKDEPT = d.DEPTNO) "
// Join such that each row also contains the department that
// administers the employee's department.
+ "INNER JOIN DEPARTMENT a ON d.ADMRDEPT = a.DEPTNO) "
// Do not find the specified employee.
+ "WHERE (e.EMPNO <> [employee's employee number]) "
// Find people in the employee's department.
+ "AND ((d.MGRNO = [employee's employee number]) "
// Find the managers of the departments that the employee's
department
// administers.
+ "OR (a.MGRNO = [employee's employee number] AND e.EMPNO =
```
d.MGRNO)) "
+ "ORDER BY e.LASTNAME"

Listing 10.29 - An SQL statement that selects the employees that a specified employee manages

- For your reference, the completed version of the code is available in hrm.completed.pdqInlineStyle.HumanResourcesManager.
Chapter 11 - Annotated-Method Programming Style

In this chapter, you will learn how to use the pureQuery annotated-method programming style.

11.1 Annotated-method Programming Style: The big picture

pureQuery provides an annotated-method programming style for executing SQL statements. Like the inline style, the annotated-method style makes it easy write applications that access databases. In addition, it allows you to use static SQL.

In the annotated-method style, rather than invoking the methods in the Data interface to execute SQL statements, you define interfaces in which each method executes a single SQL statement. Each of these methods is called an annotated method. Each annotated method must have an annotation to indicate what SQL statement the method should execute. The return type of the method indicates how pureQuery should return the query result. Listing 11.1 shows an example of an interface that defines two annotated methods.

```java
import hrm.completed.DepartmentBean;
import java.util.List;
import com.ibm.pdq.annotation.Select;

public interface DepartmentData
{
    @Select(sql="SELECT * FROM DEPARTMENT")
    List<DepartmentBean> selectAllDepartments();

    @Select(sql="SELECT * FROM DEPARTMENT WHERE DEPTNO = ?")
    DepartmentBean selectDepartmentByNumber(String departmentNumber);
}
```

Listing 11.1 - An interface that defines annotated methods

The pureQuery Generator utility generates implementation classes for interfaces of annotated methods. You must have IBM Optim Development Studio installed to use the pureQuery Generator. When developing pureQuery applications in Optim Development Studio, the tool generates implementation classes automatically when you build your
application. After an implementation class has been generated for an interface, you can create an instance of the interface using DataFactory (which actually creates an instance of the generated implementation class whose name ends in "...Impl") and run its methods, as shown in Listing 11.2.

```java
departmentData data = DataFactory.getData(DepartmentData.class, connection);
List<DepartmentBean> allDepartments = data.selectAllDepartments());
// Iterate through the departments
DepartmentBean department = data.selectDepartmentByNumber (departmentNumber);
// Do something with the department
```

Listing 11.2 - A simple example of using the annotated-method style to execute two SQL statements

Figure 11.1 shows how you can create an instance of a user-defined interface.

![Diagram](image-url)
Figure 11.2 shows the return types that user-defined annotated methods can have.

In Figure 11.2, `RowType` represents the format in which each row of a query result is returned. There are a number of ways to specify this. `ResultType` represents the format that is specified by the parameterized type `RES` of a `com.ibm.pdq.runtime.handlers.ResultHandler<RES>` object. `CallResultType` represents the format that is specified by the parameterized type `CAL` of a `com.ibm.pdq.runtime.handlers.CallHandlerWithParameters<CAL>` object. Chapter 8 discusses pureQuery handlers.

There are several benefits to using the annotated-method programming style. These are some of them:

- You can speed-up your application development. pureQuery generates the method implementations. All you have to do is declare methods with your choice of parameters and return types, and write the SQL statements for the methods to execute. The tools take care of the rest of the work.

- You can choose from a wide variety of return types for the query results. A few of the possible formats are a single pureQuery bean, a single `java.util.Map`, a collection of pureQuery beans, and a collection of `Maps`.

- If you are using the IBM® DB2® Database, you can easily switch between dynamic and static execution of the SQL. Static SQL has many performance, reliability, and security advantages. For a brief description of static SQL see Appendix C.
The interfaces containing annotated methods isolate the SQL statements from the rest of the application code. This makes it easy to analyze the SQL that the application is executing, and it allows developers to find and use the data access methods without having to know any specifics about the database. It also makes it easy to alter the SQL statements with no or minimal impacts to the application code.

### 11.2 Annotated-method interfaces

To execute SQL statements using the annotated-method programming style, you specify the SQL statements and annotated methods in an interface. First, declare an interface, and then for each SQL statement that you want to execute, declare a method in the interface. You can then create instances of the interface in your application and invoke the annotated methods.

To get an instance of an interface of annotated methods, you must invoke a `getData` method in the class `com.ibm.pdq.runtime.factory.DataFactory`. The `DataFactory` class has many methods named `getData`. For the annotated-method style, choose a method that has a `Class<T>` `interfaceClass` parameter. These methods return instances of the interfaces that you specify. To create an instance that you can use to run SQL statements against a database, choose a method that has either a `javax.sql.DataSource` parameter or a `java.sql.Connection` parameter.

Each implementation class that the Generator utility generates implements the `com.ibm.pdq.runtime.Data` interface. The `Data` interface provides the methods `commit()`, `rollback()`, `setAutoCommit(boolean)`, `getAutoCommit()`, and `close()`, all of which call the similarly-named method on the `Data` instance’s contained `Connection` object. Use these methods on the instance of your interface to manage the `Connection` object and to perform transaction management for the SQL statements executed by your annotated methods.

### 11.2.1 In practice: Overview of the “In practice” sections in this chapter

In this chapter, the “In practice” sections modify the class `hrm.inProgress.pdqMethodStyle.HumanResourcesManager` and the interface `hrm.inProgress.pdqMethodStyle.HumanResourcesData` for the Human Resources Manager application. These files are in the `GettingStartedWithPureQuery` project that provides an overview of this application. Most of the “In practice” sections ask you to add a line or two of pureQuery code to either the class or the interface. Look for the comments that say `TODO: Add pureQuery code here` to know where to add the code.

You must have Optim Development Studio installed to be able to use the pureQuery Generator to generate implementation classes for instances of annotated methods. Consequently, the “In practice” sections in this chapter assume that you are using Optim...
Development Studio for development, and that you have added pureQuery support to the GettingStartedWithPureQuery project. If you have not already added pureQuery support to your project, refer to Section 2.4.1 for instructions.

The Generator utility uses metadata about your database to know how to generate the code that processes the tables in your database. Therefore, the Generator utility relies on Optim Development Studio having a live database connection, or on using the “work offline” feature in which it stores metadata locally. The “In practice” sections in this chapter assume that you have a live database connection in Optim Development Studio, or that you are using the “work offline” feature.

11.2.2 In practice: Observe the HumanResourcesData interface

Human Resources Manager uses only one interface of annotated methods: HumanResourcesData. Locate the declaration of this interface in the hrm.inProgress.pdqMethodStyle package. hrm.inProgress.pdqMethodStyle.HumanResourcesData does not yet contain any annotated methods. You can add them in the “In practice” sections in this chapter.

11.2.3 In practice: Create an instance of an interface of annotated methods in the displayLogin() method

The Human Resources Manager application creates a single instance of hrm.inProgress.pdqMethodStyle.HumanResourcesData in the displayLogin() method, passes that instance to other methods as required, and then closes the instance before the application exits. Modify the displayLogin() method in the class hrm.inProgress.pdqMethodStyle.HumanResourcesManager so that it creates and closes an instance of hrm.inProgress.pdqMethodStyle.HumanResourcesData.

11.2.3.1 Add code to the displayLogin() method to create the HumanResourcesData object

Since we are using the annotated-method programming style, we need a getData method that takes a Class<T> interfaceClass parameter. We will be using a database, so we choose a method that has a javax.sql.DataSource parameter. We choose the method shown in Listing 11.3.

DataFactory.getData (Class<T> interfaceClass, DataSource ds)

Listing 11.3 - The getData method that we will use to create an instance of HumanResourcesData

The method displayLogin() in the class hrm.inProgress.pdqMethodStyle.HumanResourcesManager contains a declaration for a variable named data. Add code that creates an instance of hrm.inProgress.pdqMethodStyle.HumanResourcesData and assigns the result to data. Listing 11.4 shows the updated version of the code.
// Get the DataSource, and use it to create an instance of
// HumanResourcesData.
DataSource dataSource = HRMDataSourceFactory.getDataSourceForApplication()
; data = DataFactory.getData(HumanResourcesData.class, dataSource);

Listing 11.4 - Add code to the displayLogin() method that creates an instance of
HumanResourcesData

11.2.3.2 Close the Data object when your application no longer needs it
Modify the finally block in displayLogin() to close the instance of
hrm.inProgress.pdqMethodStyle.HumanResourcesData. This closes the
java.sql.Connection object that pureQuery created for the instance. Listing 11.5
shows the code that you need to add.

Listing 11.5 - Add code to the displayLogin() method that closes the instance of
HumanResourcesData

11.3 Generating an implementation class for an interface of annotated
methods
In order to use the methods that you declare in the annotated method style, you must use
the pureQuery Generator to generate an implementation class for the interfaces. There
are two ways to run the pureQuery Generator. If you are developing your Java project in
Optim Development Studio, and you have added pureQuery support to your project, then
the tool generates the implementation classes automatically when you build your project.
Additionally, if you have Optim Development Studio installed, you can run the pureQuery
Generator from the command line.

This book only discusses how to use the Generator within Optim Development Studio. For
instructions on running the Generator from the command line, see the documentation for
IBM Optim pureQuery Runtime.

11.3.1 Specifying options to the pureQuery Generator
pureQuery provides multiple utilities. All of these utilities allow you to specify options on
the command line. Refer to the documentation for IBM Optim pureQuery Runtime to see
the options that are available for the Generator utility. For the Generator utility and for
some of the other utilities, you can also specify the interfaces that the utility should use,
along with the options it should use for each file, in a type of file that is known as an
options file. Refer to Appendix D for a description of options files. pureQuery allows two
types of options files: `genProps` and `bindProps`. The Generator utility uses `genProps` files.

An options file is a text file that lists interfaces and `pureQuery` xml files to be processed, and the options to apply to each interface or file. Each interface and `pureQuery` xml file must be listed on a separate line in the options file, followed by an equal sign (=), and then any options that `pureQuery` should use when it processes that item. Additionally, in each options file, you can provide default options on a line that starts with `defaultOptions`. `pureQuery` uses each option that you specify on the `defaultOptions` line for all the items, except for those items where the option is specified with a different value. Listing 11.6 shows an example of how you would provide default options and interfaces in an options file.

```
defaultOptions = <options to use as defaults>
myPackage1.MyInterface1 = <options for myPackage1.MyInterface1>
myPackage2.MyInterface2 = <options for myPackage2.MyInterface2>
```

Listing 11.6 - Example of listing default options and interfaces in an options file

### 11.3.1.1 Specifying option files in Optim Development Studio

IBM Optim Development Studio automatically runs the Generator utility for the interfaces in `pureQuery`-enabled projects when you build them. `pureQuery`-enabled projects in IBM Optim Development Studio contain a folder named `pureQueryFolder`. The folder contains two options files: one named `Default.genProps`, and one named `Default.bindProps`. Notice the file `Default.genProps` in Figure 11.3. If you want to specify options for any of the interfaces in your project, list those interfaces in `Default.genProps`, along with the options that you want. When there are interfaces in your project that are not in `Default.genProps`, Optim Development Studio still generates implementation classes for them, using default settings.

![Figure 11.3 - The options file Default.genProps in a pureQuery-enabled project](image)

Through menu options and content assist, Optim Development Studio can help you to create options files. It can help you to add your interfaces and `pureQuery` xml files to your
options files, and it can help you to add options. Refer to Chapter 14 for information about these features.

When you save changes to Default.genProps, a dialogue like the one in Figure 11.4 will pop up with a warning message that indicates that the options you are specifying for interfaces might have changed, and that you might need to rebuild your project for its implementation classes to reflect your changes. The dialogue will ask if you want to rebuild the project so that any changes will take affect. Select Yes. This generates the implementation classes for your project. Chapter 13 describes generating implementation classes.

![Warning](image)

*Figure 11.4 - When you save changes to Default.bindProps, a dialogue asks if you want to rebuild your project to reflect any changes.*

### 11.3.1.2 In practice: Specify options for the interface

`hrm.inProgress.pdqMethodStyle.HumanResourcesData`

Add an entry to `Default.genProps` for the interface `hrm.inProgress.pdqMethodStyle.HumanResourcesData`. Specify a *root package name* by using the option `-rootPkgName`. Provide the value `HRMIM` (for "Human Resources Manager," "InProgress," and "Method style"), as shown in Listing 11.7. For applications that execute SQL statically, pureQuery uses the root package name to determine the names of the packages in the database. Refer Chapter 11 for more information about how pureQuery uses the root package name. If you do not specify the root package name, the Generator utility determines a name based on the name of the interface. In our application, we should specify the root package name because there are multiple interfaces in the GettingStartedWithPureQuery project with the name `HumanResourcesData`.

```plaintext
hrm.inProgress.pdqMethodStyle.HumanResourcesData = -rootPkgName HRMIM
```

*Listing 11.7 - Specifying an option for the HumanResourcesData interface in an options file*

Save your changes to `Default.genProps`, and select Yes on the dialogue like the one in Figure 11.4.
11.3.2 Generating an implementation class in Optim Development Studio

When pureQuery generates an implementation class, it appends "Impl" to the name of interface to create the name of the implementation class.

When your build your project, IBM Optim Development Studio automatically generates implementation classes for the interfaces that contain annotated methods. You can have Optim Development Studio build your project automatically, or you can tell it to build manually.

If anything goes wrong during the generation, IBM Optim Development Studio places a red exception marker on the interface, as in Figure 11.5. Similarly, if there are warnings during the generation, IBM Optim Development Studio places a yellow warning marker on the interface. You can open the interface and point your mouse to the marker in the interface to get more information. If there is an exception marker on your implementation class because the class does not compile, you probably incorrectly declared at least one of the annotated methods.

Figure 11.5 - An error that IBM Optim Development Studio puts on an interface because it cannot generate its implementation class

11.4 Annotated methods

Follow these three steps to add an annotated method to an interface:

1. Determine the SQL statement that you want to execute.

2. Add an annotation in which you specify the SQL statement. Use one of these three annotations:
   A. `com.ibm.pdq.annotation.Select`: Use this annotation for SQL SELECT statements.
   B. `com.ibm.pdq.annotation.Update`: Use this annotation for SQL INSERT, UPDATE, and DELETE statements and for SQL statements that do not return any results, such as data definition language (DDL) statements.
   C. `com.ibm.pdq.annotation.Call`: Use this annotation for SQL statements that run stored procedure calls.

3. Declare the method.
A. Specify the modifiers that you want the method to use. For example, you can specify the modifier public.

B. Specify the return type of the method. The type of annotation that you use determines which return types you can use.
   1. @Select annotation: The return type specifies the format of the query result.
   2. @Update annotation: The return type specifies if the method returns an update count.
   3. @Call annotation: The return type specifies the format of the query results.

C. Specify a name for the method.

D. Specify the method parameters that pureQuery will use to populate parameters in the SQL statement. Section 6.5 discusses how to use parameter markers to indicate how pureQuery should set the values of parameter from the method parameters.

The following two steps are optional:

4. If you want the annotated method to use one or more handlers, specify the handlers. You might want to use a com.ibm.pdq.runtime.handlers.ResultHandler<RES>, a com.ibm.pdq.runtime.handlers.RowHandler<ROW>, or a com.ibm.pdq.runtime.handlers.CallHandlerWithParameters<CAL> to customize the return type of your method. You also might want to use a com.ibm.pdq.runtime.handlers.ParameterHandler to customize how pureQuery sets the values of the SQL parameters. Specify handlers either in the com.ibm.pdq.annotation.Handler annotation, or as method parameters.

5. If you want to override any of the default cursor attributes for the SQL statement, specify the attributes that you want to use in the com.ibm.pdq.annotation.Cursor annotation. Without this annotation, annotated methods use the cursor type “forward only,” the cursor concurrency mode “read only,” and the cursor holdability “close cursors at commit.” You can change the values of these attributes by using the @Cursor annotation. You can use this to set the type, concurrency, and holdability by using the attributes type, concurrency, and holdability, respectively. You can also change the cursor name by using the attribute cursorName, and you can indicate if pureQuery should use a rowset cursor when it executes the SQL statement statically by using the attribute allowStaticRowsetCursors.
11.4.1 Declaring the return type

Use the following questions as a guide to help you select the right return type for your annotated methods.

11.4.1.1 Methods with the @Select annotation

- **What format do you want to use for each row?**
  - A pureQuery bean or a scalar: pureQuery creates and populates the object automatically. For pureQuery beans, pureQuery uses the rules defined in Section 3.31 to determine the bean property that represents each column that is selected from the database. When pureQuery cannot find a property that represents a column in the SQL statement, pureQuery does not store the value of that column in the bean. For scalars, pureQuery constructs an object and attempts to store the contents of the first column of the query result in that object.
  - An object of a specific type, constructed by a `com.ibm.pdq.runtime.handlers.RowHandler<ROW>` object: You will have to specify the `RowHandler<ROW>` that will construct the object.
  - A `java.util.Map<String, Object>` object: pureQuery constructs a `Map` in which the keys contain the labels of the columns and the values contain the contents of the columns.

- **What object do you want to contain all of the rows?**
  - An array in which each entry represents a row: Specify an array as the return type, such as `MyPureQueryBean[]`.
  - A `java.util.List` object in which each element represents a row: Specify the parameterized List as the return type, such as `java.util.List<MyPureQueryBean>`.
  - A `java.util.Iterator<E>` object for which each call to the `Iterator.next()` method returns a row: Specify the parameterized `Iterator` as the return type, such as `java.util.Iterator<MyPureQueryBean>`.

Methods with this return type do not fully materialize the information from the database during the initial method call. Instead, pureQuery retrieves one row each time the application calls the `next` method. This is particularly useful if your application retrieves large amounts of information from the database, and you do not want to store all of the information in memory at one time. When the application calls `next()` to retrieve all of the rows from the
Iterator, pureQuery closes it automatically, freeing the associated database resources. However, if you have an application that does not retrieve all the rows, you must free the database resources by closing the `Iterator<E>` explicitly. Do so by casting the `Iterator<E>` object as a `com.ibm.pdq.runtime.ResultIterator<T>` object and calling its `close()` method, like this:

```java
((com.ibm.pdq.runtime.ResultIterator) myIterator).close()
```

- Only the first row, in the object used to contain the row: Specify the object used to contain each row as the return type, such as `MyPureQueryBean`. The method returns `null` if the query result does not contain any rows.

- An object of a specific type, constructed by a `com.ibm.pdq.runtime.handlers.ResultHandler<RES>` object: Specify the parameterized type (represented by `RES`) as the return type of the method. You will have to specify the `ResultHandler<RES>` that will construct the object.

- A `java.sql.ResultSet` object: Specify `java.sql.ResultSet` as the return type of the method.

### 11.4.1.2 Methods with the @Update annotation

- **What do you want the method to return?**

  - **An update count**: Specify `int` as the return type if you want the method to return an update count indicating how many rows were modified.

  - **Update counts**: Specify `int[]` as the return type if you want the method to execute the SQL statement multiple times with different parameters each time. In this case, the parameter to the method must be a `java.util.List`, `java.util.Iterator`, or `array`. Each entry in the `List`, `Iterator`, or array must provide the parameters for one execution of the SQL statement. pureQuery executes the SQL statement once for each entry and returns the update count from that statement in the corresponding location in the `int[]`.

  - **void**: Specify `void` as the return type if the SQL statement will not produce any results that you want the method to return.
11.4.1.3 Methods with the @Call annotation

- What do you want the method to return?
  - `void`: Specify `void` as the return type if the stored procedure will not produce any results that you want the method to return.
  - The first query result: If you want the method to return only the first query result, using one of the return types allowed with the @Select annotation, specify the return type that you want to use. You can use any of the types that are allowed for @Select, except for those that involve either a ResultHandler or a RowHandler and those that return only a single row of the query result.
  - An object of a specific type, constructed by a `com.ibm.pdq.runtime.handlers.CallHandlerWithParameters<CAL>` object: Specify the parameterized type (represented by `CAL`) as the return type of the method. You will have to specify the `CallHandlerWithParameters<CAL>` that will construct the object.

11.4.2 In practice: Execute a simple SELECT statement that uses cursor attributes -- Complete the “Company Employees” screen

The Company Employees screen displays all of the employees in the company in pages of 10 employees each. It allows the user to request additional information about any particular employee on the screen. The `displayAllEmployees (Data data)` method creates this screen. Add the `pureQuery` code to it to get the employee information from the database using the annotated-method style.

11.4.2.1 Add an annotated method to `HumanResourcesData` that executes the SQL

First, we need to determine an SQL statement to use. We will use the query shown in Listing 11.8, which will select all the employees and sort them by their names.

```
SELECT * FROM EMPLOYEE ORDER BY LASTNAME, FIRSTNME, MIDINIT
```

Listing 11.8 - An SQL query for selecting all the employees in the company

Next, we need to declare an annotated method in `hrm.inProgress.pdqMethodStyle.HumanResourcesData` that will execute this query.
We are executing a `SELECT` statement, so we use the `com.ibm.pdq.annotation.Select` annotation. Since we need several pieces of information about each employee, it would be convenient to retrieve each row as an instance of `EmployeeBean`. We will be selecting all the employees in the database, but we would prefer not to have to store all of the company's employees in the application's memory at once, since that could be quite a lot of memory, so we will return the rows in a `java.util.Iterator<E>` object. Therefore, we will use `java.util.Iterator<EmployeeBean>` as the return type of our method.

Our SQL statement does not have any parameters, so the method does not need any parameters.

While users are viewing the list of employees, they can request to view additional information about an employee and then return to the list. Therefore, we need to set the cursor holdability to "hold cursors over commit." We use the `com.ibm.pdq.annotation.Cursor` annotation to specify the holdability cursor attribute `java.sql.ResultSet.HOLD_CURSORS_OVER_COMMIT`. Listing 11.9 shows the definition of the new annotated method.

```java
@Cursor(holdability = ResultSet.HOLD_CURSORS_OVER_COMMIT)
@Select(sql = "SELECT * FROM EMPLOYEE ORDER BY LASTNAME, FIRSTNAME, MIDINIT")
Iterator<EmployeeBean> selectAllEmployees();
```

**Listing 11.9 - Add an annotated method to the HumanResourcesData interface that creates an Iterator<EmployeeBean> of all the employees**

After you have added the method to `hrm.inProgress.pdqMethodStyle.HumanResourcesData`, build your project so that Optim Development Studio will generate an implementation class. Confirm that `hrm.inProgress.pdqMethodStyle.HumanResourcesDataImpl` was generated without any problems. Figure 11.6 shows the location of the new file. If IBM Optim Development studio has placed a red exception marker on the interface, read the text of the exception and determine what you need to do to solve the problem.

![Figure 11.6 - The generated implementation class HumanResourcesDataImpl](image)

### 11.4.2.2 Modify the displayAllEmployees method to use the new annotated method

In `hrm.inProgress.pdqMethodStyle.HumanResourcesManager`, modify the `displayAllEmployees` method to use the new annotated method to query the database...
and create the Iterator<EmployeeBean> object of the employees. This is shown in Listing 11.10.

```java
// Search the database for all of the employees in the company
// and create an Iterator<EmployeeBean> object to retrieve them.
employees = data.selectAllEmployees();
```

**Listing 11.10 - Modify the displayAllEmployees method to use the annotated method**

11.4.2.3 Close the Iterator<EmployeeBean> object to free the associated database resources

The user of Human Resources Manager might not view all of the employees. Therefore, we need to close employees explicitly to free the database resources. Add code to close employees.

```java
finally {
    // Close employees. This frees the associated database resources.
    if (null != employees)
        ((ResultIterator<EmployeeBean>) employees).close();
}
```

**Listing 11.11 - Add code to the displayAllEmployees method that closes the Iterator<EmployeeBean> when the application no longer needs it**

11.4.2.3.1 Test the "Company Employees" screen

Save and test your application. As before, supply 000030 when the Human Resources Manager asks for your employee number. On the main menu, select option 4 to display all the employees in the company. You will then be able to see the employees, one page at a time.

11.5 Executing SQL statements that have parameters

It is very easy to use parameters in SQL statements when you use the pureQuery annotated method style. The four types of parameter markers that are available in the inline style are also available in the annotated method style. With each of these types, pureQuery automatically populates the SQL parameters with the information that you provide. When you use multiple parameter markers in an SQL statement, all of the parameter markers must be of the same type.

In the pureQuery method style, you specify the objects that pureQuery should use to populate the parameters of your SQL statement as the parameters of the annotated methods when you declare the methods.

Table 11.1 shows the four types of parameter markers. References to Map indicate the class java.util.Map. Listing 11.12 shows the definitions of the objects that the “Sample usage” column references.
If the functionality of pureQuery’s parameter markers does not meet the needs of your application, you can write an implementation of the 
\texttt{com.ibm.pdq.runtime.handlers.ParameterHandler} interface to set the parameters appropriately.

// Scalars
String deptNo = "C01";
String managerEmpNo = "000030";

// pureQuery bean
DepartmentBean dept = new DepartmentBean();
department.setDepartmentNumber ("C01");
department.setManagerEmployeeNumber ("000030");

// Map
Map<String, String> map = new HashMap<String, String> ();
map.put ("departmentNumber", "C01");
map.put ("managerEmployeeNumber", "000030");

\textbf{Listing 11.12 - Definitions of the objects referenced in the "Sample usage" column of Table 11.1}

<table>
<thead>
<tr>
<th>Marker</th>
<th>Types of objects in method parameters</th>
<th>Meaning</th>
<th>Sample usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>Scalars</td>
<td>Directs pureQuery to use the value of one of the method parameters as the value of this SQL parameter. The first ? represents the first method parameter, the second ? represents the second method parameter, and so forth.</td>
<td>@Select(sql = &quot;SELECT * FROM EMPLOYEE WHERE WORKDEPT = ? AND EMPNO != ?&quot;) List&lt;EmployeeBean&gt; selectDeptEmployees (String deptNo, String managerEmpNo);</td>
</tr>
<tr>
<td>?\text{n}</td>
<td>Scalars</td>
<td>Directs pureQuery to use the value of the \text{n}th method parameter as the value of the SQL parameter. ?\text{1} represents the first</td>
<td>@Select(sql = &quot;SELECT * FROM EMPLOYEE WHERE WORKDEPT = ?1 AND EMPNO != ?2&quot;) List&lt;EmployeeBean&gt; selectDeptEmployees (String deptNo, String managerEmpNo);</td>
</tr>
</tbody>
</table>
method parameter, \(?2\) represents the second method parameter, and so forth.

\[ ?n.name \]

**pureQuery beans and Map objects**

Directs pureQuery to use the value of the property **name** in the \(n\)th method parameter as the value of the SQL parameter. For example, \(?1.lastName\) represents the **lastName** property of first method parameter. The property name is case sensitive.

```java
// pureQuery bean
@Select(sql = "SELECT * FROM EMPLOYEE WHERE WORKDEPT = ?1.departmentNumber AND EMPNO != ?1.managerEmployeeNumber")
List<EmployeeBean> selectDeptEmployees (DepartmentBean dept);

// Map
@Select(sql = "SELECT * FROM EMPLOYEE WHERE WORKDEPT = :departmentNumber AND EMPNO != :managerEmployeeNumber")
List<EmployeeBean> selectDeptEmployees (Map<String, String> map);
```

\[ :name \]

**pureQuery beans and Map objects**

Directs pureQuery to use the value of the property **name** in the first method parameter as the value of the SQL parameter. This is a shorter way of specifying \(?1.name\) that is useful when only one pureQuery bean or Map object is specified. The property name is case sensitive.

```java
// pureQuery bean
@Select(sql = "SELECT * FROM EMPLOYEE WHERE WORKDEPT = ?1.departmentNumber AND EMPNO != ?1.managerEmployeeNumber")
List<EmployeeBean> selectDeptEmployees (DepartmentBean dept);

// Map
@Select(sql = "SELECT * FROM EMPLOYEE WHERE WORKDEPT = :departmentNumber AND EMPNO != :managerEmployeeNumber")
List<EmployeeBean> selectDeptEmployees (Map<String, String> map);
```

### Table 11.1 - Parameter markers in the annotated-method programming style

**11.5.1 In practice: Specify a scalar as a parameter to an SQL statement -- Complete the “Login” screen**

When Human Resources Manager starts, it asks the user for his or her employee number. The application queries the database for information about the employee with that employee number. If the database contains an employee with that employee number, it displays the main menu. If it does not contain such an employee, the application exits.
The original code provided in the displayLogin method of hrm.inProgress.pdqMethodStyle.HumanResourcesManager does not query the database for the employee; instead, it just creates an empty EmployeeBean object. This is shown in Listing 11.13.

// Search the database for an employee with the employee number that the // user entered.  // TODO: Add pureQuery code here.  employeeUsedForLogin = new EmployeeBean();
Listing 11.13- Original assignment of employeeUsedForLogin in the displayLogin() method of hrm.inProgress.pdqMethodStyle.HumanResourcesManager

Modify the code so that it queries the database for the employee information.

11.5.1.1 Add an annotated method to HumanResourcesData that executes the SQL

We will use the SQL statement in Listing 11.14 to get the information about the employee with the specified employee number. The text in square brackets indicates an SQL parameter pureQuery needs to populate with the employee number that the user enters.

SELECT * FROM EMPLOYEE WHERE EMPNO = [employee number]
Listing 11.14 - An SQL query for selecting the employee with the specified employee number

Next, we need to declare an annotated method in hrm.inProgress.pdqMethodStyle.HumanResourcesData that will execute this query. Since we are executing a SELECT statement, we use the com.ibm.pdq.annotation.Select annotation. We want the row to be returned in an EmployeeBean object. Since the column EMPNO is the primary key of the EMPLOYEE table, at most one row in the table will match the employee number that the user enters. Therefore, we choose to have EmployeeBean as the return type of the method.

The SQL statement has one parameter. Refer to Table 11.1 to determine what type of parameter marker to use. Because we have the employee number in a string, we can use the ? or the ?n style of parameter marker. We choose to use the ? style. We will specify the employee number as the first (and only) parameter in the annotated method.

Add the method shown in Listing 11.15 to hrm.inProgress.pdqMethodStyle.HumanResourcesData. After you have added the method, build your project so that Optim Development Studio will generate an updated implementation class.

@Select(sql = "SELECT * FROM EMPLOYEE WHERE EMPNO = ?")
EmployeeBean selectEmployee (String employeeNumber);
Chapter 11 – Annotated-Method Programming Style 225

Listing 11.15 - An annotated method that creates an EmployeeBean object for the employee with the specified employee number

11.5.1.2 Modify the displayLogin() method to use the new annotated method

In hrm.inProgress.pdqMethodStyle.HumanResourcesManager, modify the displayLogin method to use the new annotated method to query the database and create the EmployeeBean object. This is shown in Listing 11.16.

// Search the database for an employee with the employee number that the // user entered.
employeeUsedForLogin = data.selectEmployee (employeeNumber);

Listing 11.16 - Modify the displayLogin() method to use the annotated method

11.5.2 In practice: Specify a pureQuery bean as a parameter to an SQL statement-- Complete the “Employee Information” screen

The “Employee Information” screen displays detailed information for a single employee. The displayEmployeeInformation method has an EmployeeBean parameter that provides details about the employee. The screen also displays summary information about the manager of the employee. Information about an employee’s manager is not included in an EmployeeBean object. Add an annotated method that gets the information about an employee’s manager.

11.5.2.1 Add an annotated method to HumanResourcesData that executes the SQL

We next need an SQL statement to determine the manager of an employee... Listing 10.12 shows the SQL query that we will use to select the information about the employee who manages the specified employee. This SQL statement is described in Section 6.4.2.1.

Next, we need to declare an annotated method in hrm.inProgress.pdqMethodStyle.HumanResourcesData that will execute this query. Since we are executing a SELECT statement, we use the com.ibm.pdq.annotation.Select annotation. We want the method to return the manager’s information in an EmployeeBean object, and there will be at most one row, so we choose to have EmployeeBean as the return type of the method.

We will use pureQuery parameter markers for the parameters to the SQL. We need the employee number and the department number for the employee. Both of these pieces of information are available in the EmployeeBean object that is a parameter to the method. Refer to Table 11.1, which lists the various types of parameter markers. Both the ?n.name type of parameter marker and the :name type expect that the information is a property in a pureQuery bean. We choose to use the ?n.name type. The names of the appropriate properties in EmployeeBean are employeeNumber and departmentNumber. Therefore, the two parameter markers are ?1.employeeNumber and
?1.departmentNumber. Note how easy it is to specify multiple parameters to an SQL statement!

Add the method shown in Listing 11.17 to hrm.inProgress.pdgMethodStyle.HumanResourcesData. After you have added the method, build your project so that Optim Development Studio will generate an updated implementation class.

```java
@Select(sql = "SELECT e.EMPNO, e.FIRSTNME, e.MIDINIT, e.LASTNAME, "
+ "e.WORKDEPT, e.PHONENO, e.HIREDATE, e.JOB, e.EDLEVEL, e.SEX, "
+ "e.BIRTHDATE, e.SALARY, e.BONUS, e.COMM FROM "
+ "(EMPLOYEE e INNER JOIN DEPARTMENT d ON (e.WORKDEPT = d.DEPTNO)) WHERE "
+ "(e.EMPNO <> ?1.employeeNumber) AND "
+ "((0 = (SELECT COUNT(*) FROM DEPARTMENT WHERE "
+ "MGRNO = ?1.employeeNumber AND DEPTNO = ?1.departmentNumber)) "
+ "AND (e.WORKDEPT = ?1.departmentNumber) AND (e.EMPNO = d.MGRNO)) "
+ "(WHERE DEPTNO = ?1.departmentNumber) AND e.EMPNO = d.MGRNO))"
public EmployeeBean selectEmployeeManager (EmployeeBean employee);
```

Listing 11.17 - Add an annotated method to the HumanResourcesData interface that creates an EmployeeBean object for the manager of the specified employee

11.5.2.2 Modify the displayEmployeeInformation() method to use the new annotated method

In hrm.inProgress.pdgMethodStyle.HumanResourcesManager, modify the displayEmployeeInformation method in as shown in Listing 11.18 so that it uses the new annotated method to query the database and create the EmployeeBean object representing the employee's manager.

```java
// Add the manager of the employee.
EmployeeBean manager = data.selectEmployeeManager (employee);
```

Listing 11.18 - Modify the displayEmployeeInformation method to use the annotated method

The displayEmployeeInformation method calls the utility method isEmployeeManagedByLoggedInUser. In that method as well, add code as in Listing
11.19 so that it uses the new annotated method to search the database for a employee’s manager.

Listing 11.19 - Modify the `isEmployeeManagedByLoggedInUser` method to use the annotated method

11.5.3 Special case scenarios for parameter types

Some scenarios call for specific parameter types. When you declare an annotated method, see if your application falls under one of these special cases.

11.5.3.1 Methods with the `@Update` annotation

- **Executing an SQL statement multiple times with different parameters each time:** If you want `pureQuery` to execute the SQL statement multiple times, using different parameters each time, specify a `java.util.List<E>`, `java.util.Iterator<E>`, or array in which entry provides objects for setting the parameters. `pureQuery` executes the SQL statement once for each entry. If you use an array for this purpose, you must not use an array of type `Object[]`.

- **Creating generated keys:** If your method is executing an `INSERT` statement, `pureQuery` can update the values of keys generated by the statement. To use this feature, provide the parameters for the SQL statement using one `pureQuery` bean, and provide the `com.ibm.pdq.annotation.GeneratedKey` annotation on the properties you want `pureQuery` to update.

11.5.3.2 Methods with the `@Call` annotation

- **Updating the values of OUT and INOUT parameters:** If the method does not specify a `CallHandlerWithParameters<CAL>`, and if the method executes a stored procedure that has OUT or INOUT parameters, `pureQuery` updates the corresponding properties in maps or `pureQuery` beans provided as method parameters after the execution of the stored
procedure call. pureQuery can do this only when the parameters to the annotated method are maps and pureQuery beans. The parameter markers in the SQL statement must indicate the properties in the method parameters. For annotated methods that specify a CallHandlerWithParameters<CAL>, the handler’s handleCall method must update the parameters, if you want them to be updated.

11.6 Executing SQL statements that modify information in a database

In the annotated method style, creating methods that execute statements that update the data in database is very similar to creating methods that query the data. Observe that for SQL statements that update information in a database, and for other SQL statements that do not return query results, we use the com.ibm.pdq.annotation.Update annotation. INSERT, UPDATE, and DELETE are three types of SQL statements that use this annotation.

11.6.1 Retrieving generated keys

If your method executes an SQL INSERT statement into a table that has generated keys, pureQuery can update pureQuery beans with the generated values. To use this functionality, you must specify your parameters using only one pureQuery bean. In that bean, annotate one or more properties that you want pureQuery to update with the com.ibm.pdq.annotation.GeneratedKey annotation. If you want to change the way that pureQuery updates the bean, you can specify an implementation of com.ibm.pdq.runtime.handlers.RowHandler<ROW>.

For an example of using generated keys, assume that we have a table EMPLOYEE_GEN_KEY that has a generated column, defined in Listing 11.20. We will use the class EmployeeBean to represent this table. We add the @GeneratedKey annotation to the employeeNumber property, as shown in Listing 11.21. Then, we define an annotated method that executes an INSERT statement, as shown in Listing 11.22. Note that the annotated method has only one parameter. Finally, we execute the method as shown in Listing 11.23. The output, shown in Listing 11.24, indicates that pureQuery updated the employeeNumber property with the generated employee number.

```
CREATE TABLE EMPLOYEE_GEN_KEY (EMPNO INT GENERATED ALWAYS AS IDENTITY (START WITH 10, INCREMENT BY 10), FIRSTNME VARCHAR(12), LASTNAME VARCHAR(15))
Listing 11.20 - The DB2 definition of the EMPLOYEE_GEN_KEY table

@GeneratedKey
@Column(name = "EMPNO")
public String getEmployeeNumber ()
{
    return employeeNumber;
}
```
public void setEmployeeNumber (String employeeNumber) 
{
    this.employeeNumber = employeeNumber;
}

Listing 11.21 - The @GeneratedKey annotation on the employeeNumber property in EmployeeBean

@Update(sql="INSERT INTO EMPLOYEE_GEN_KEY (FIRSTNME, LASTNAME) VALUES (:firstName, :lastName)")
int insertEmployee(EmployeeBean employee);

Listing 11.22 - An annotated method that executes an INSERT statement and produces a generated key

EmployeeBean newEmployee = new EmployeeBean ();
newEmployee.setFirstName("FirstName");
newEmployee.setLastName("LastName");
int updateCount = data.insertEmployee(newEmployee);
System.out.println("Update count: " + updateCount);
System.out.println("New employee number: " + newEmployee.getEmployeeNumber());

Listing 11.23 - Code that executes the annotated method in Listing 11.22

Update count: 1
New employee number: 10

Listing 11.24 - Sample output of the code in Listing 11.23

11.6.2 In practice: Execute an SQL statement that modifies the database -- Complete the “Change Employee Information” screen

The “Change Employee Information” screen allows the information in certain fields to be changed. The screen displays the editable fields and asks the user which one he or she would like to modify. The changeEmployeeInformation method controls the “Change Employee Information” screen. When a user asks to change a particular column, it updates the EmployeeBean object that contains the user’s information with the new information, and then it uses the updated object to update the database. Add annotated methods that update the information in the database. The application should make sure that exactly one row was updated.

11.6.2.1 Add annotated methods to HumanResourcesData that executes the SQL

We will use a separate SQL statement for each column that a user can update. Two of the SQL statements are shown in Listing 11.25 and Listing 11.26. The remaining statements follow the same pattern.

UPDATE EMPLOYEE SET LASTNAME = [new last name] WHERE EMPNO = [original employee number]
Listing 11.25 - An SQL statement for updating an employee’s last name

```sql
UPDATE EMPLOYEE SET FIRSTNME = [new first name] WHERE EMPNO = [original employee number]
```

Listing 11.26 - An SQL statement for updating an employee’s first name

Since we are executing `UPDATE` statements, we use the `com.ibm.pdq.annotation.Update` annotation. We want the methods to return an update count, so we use `int` as the return type.

We will use pureQuery parameter markers for the parameters to the SQL. All of the information about the employee will be available in the `EmployeeBean` object that describes the employee. Refer to Table 11.1, which lists the various types of parameter markers. Both the `?n.name` type of parameter marker and the `:name` type that the information is a property in a pureQuery bean. We choose to use the `:name` type.

Add annotated methods for these SQL statements. You need one method for each of the following columns: `LASTNAME`, `FIRSTNME`, `MIDINIT`, `EDLEVEL`, `BIRTHDATE`, `JOB`, `SALARY`, `BONUS`, and `COMM`. The corresponding properties in `EmployeeBean` are, respectively: `lastName`, `firstName`, `middleInitial`, `educationLevel`, `birthDate`, `job`, `salary`, `bonus`, and `commission`. Listing 11.27 and Listing 11.28 show two of the annotated methods; the remaining methods are similar. You can refer to the completed methods in `hrm.completed.pdqMethodStyle.HumanResourcesData` if you want help. After you have added the methods, build your project so that Optim Development Studio will generate an updated implementation class.

```java
@Update(sql = "UPDATE EMPLOYEE SET LASTNAME = :lastName WHERE EMPNO = :employeeNumber")
int updateEmployeeLastName (EmployeeBean employee);
```

Listing 11.27 - An annotated method that updates an employee’s last name

```java
@Update(sql = "UPDATE EMPLOYEE SET FIRSTNME = :firstName WHERE EMPNO = :employeeNumber")
int updateEmployeeFirstName (EmployeeBean employee);
```

Listing 11.28 - An annotated method that updates an employee’s first name

11.6.2.2 Modify the `changeEmployeeInformation()` method to use the new annotated methods

Modify the `changeEmployeeInformation` method in `hrm.inProgress.pdqMethodStyle.HumanResourcesManager` to use the new annotated methods to update the database. Listing 11.29 and Listing 11.30 show two of the changes; the remaining changes are similar.

```java
// Update the last name in the database
updateCount = data.updateEmployeeLastName (employee);
```
Listing 11.29 - Changes to the displayEmployeeInformation method to update a last name

// Update the first name in the database
updateCount = data.updateEmployeeFirstName(employee);

Listing 11.30 - Changes to the displayEmployeeInformation method to update a first name

11.6.2.3 Test the "Change Employee Information" screen

Save and test your application. Supply 000030 to login as Sally Kwan. On the main menu, select option 2 to change information about Sally. Then, return to the main menu and select option 4 to go to the "Company Employees" screen and find Heather A. Nicholls, one of the employees that Sally manages. View information about Heather, and then change information for her.

11.7 Tooling Support for Annotated Methods and Generated Code

General pureQuery tooling support was introduced starting in Chapter 3. The following features provide support that is specifically tailored to code generated using the pureQuery annotated method API.

11.7.1 Generating XML file to override the SQL statements in interfaces

When the implementation class is generated by the pureQuery Generator for an interface that defines annotated methods, you can use an XML file to specify or override the SQL statements that those methods use. You can also use the same XML file to override how the columns in a database object, such as a table or view, map to the properties of a bean. To specify an XML configuration file, use the xmlFile option when you invoke the pureQuery Generator.

You might want to override SQL statements in an XML file when you want to use the same interface in another application that runs against a different database. For example, assume that your interface contains the following method:

@Select(sql="SELECT CUSTID, NAME FROM Customer WHERE STOREREGION=?1")
Iterator<Customer> getCustomersInRegion(int r)

In the XML configuration file for the other application, you can override the SQL like this:

<named-native-query
name="myPackage.CustomerInterface#getCustomersInRegion(int)">
  <query>
    <![CDATA[SELECT ID, NAME FROM CUST WHERE REGION=?1]]>
  </query>
</named-native-query>
To use the pureQuery tools to generate the XML file, a subset of the XML JPA specification, Open the pureQuery interface class, right-click on the interface and select pureQuery → Generate XML.

11.7.2 EMFT JET template customization for code generation

To generate code, the tooling uses Eclipse Modeling Framework Technologies(EMFT) Java Emitter Templates (JET) templates. In order to generate your own customized code, you must override the default JET transformation by creating an EMFT JET project, creating the templates, and telling ODS to use your own transformations. The many different JET templates used by the pureQuery tooling are located in the com.ibm.datatools.javatool.transform.codegen plugin within ODS. For detailed information on creating the project, templates and overriding the default transformation refer to the ODS documentation for Creating Eclipse Modeling Framework Technologies (EMFT) Java Emitter Templates (JET) to generate customized code.

The supported databases for code generation from templates are DB2 LUW, DB2 for z/OS, IDS, DB2 iSeries and Oracle.

The following table lists some of the templates for code generation that are in the plugin:

<table>
<thead>
<tr>
<th>Name of template</th>
<th>Supported DBMS</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GenTableBean.jet</td>
<td>DB2 LUW, DB2 for z/OS, IDS, Oracle</td>
<td>Generates a bean that contains properties of columns of a table or view.</td>
</tr>
<tr>
<td>GenTableInlineJUnit</td>
<td>DB2 LUW, DB2 for z/OS, IDS, Oracle</td>
<td>Generates a JUnit test case that contains inline methods for accessing the corresponding database object.</td>
</tr>
<tr>
<td>GenTableInlineSample.jet</td>
<td>DB2 LUW, DB2 for z/OS, IDS, Oracle</td>
<td>Generates a Java test class that contains inline methods for accessing the corresponding database object.</td>
</tr>
<tr>
<td>GenTableInterface.jet</td>
<td>DB2 LUW, DB2 for z/OS, IDS, Oracle</td>
<td>Generates a Java interface that contains annotates methods for accessing the corresponding database object.</td>
</tr>
<tr>
<td>GenTableInterfaceJUnit.jet</td>
<td>DB2 LUW, DB2 for z/OS, IDS, Oracle</td>
<td>Generates a JUnit test case for</td>
</tr>
</tbody>
</table>


Chapter 11 – Annotated-Method Programming Style

Table 11.2 - JET templates used for code generation from a database Table or View

<table>
<thead>
<tr>
<th>JET Template</th>
<th>DBMS Platform</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>GenTableInterfaceTest.jet</td>
<td>DB2 LUW, DB2 for z/OS, IDS, Oracle</td>
<td>Generates a test class for testing the implementation of the corresponding interface.</td>
</tr>
</tbody>
</table>

Once you have created your EMFT JET Transformation project and created a template to override one of the default templates, go to the pureQuery Transforms preference page and specify which transformation the workbench should use. An example is illustrated in Figure 11.7:

Figure 11.7 - Overriding the default JET templates to generate your own customized code

11.8 Using static SQL with annotated-method style applications

One of the benefits of the annotated method style is that you can use it to execute SQL statements statically. Refer to Appendix C for a brief overview of static SQL.
11.8.1 Binding the SQL statements in the interfaces

If you want to use static SQL, you must use the pureQuery StaticBinder utility to bind the SQL statements in your interfaces. You can run the StaticBinder utility either from the command line or by selecting menu options in IBM Optim Development Studio. Refer to Chapter 7 for a description of the StaticBinder utility and how to specify what packages it should bind in the database.

11.8.1.1 In practice: Bind the SQL statements in your interface

If you are using IBM DB2 Database for your database, use the StaticBinder to bind the statements in your interface in the database.

11.8.1.1.1 Specify options for the interface

As for the Generator utility, you can use an options file to specify options for the StaticBinder utility. Notice the options file named Default.bindProps in Figure 11.8. It is located in the pureQueryFolder directory in the GettingStartedWithPureQuery project.

Add a line to Default.bindProps for the interface. The StaticBinder’s defaults are fine for our purposes, so you do not specify any options on the line. The StaticBinder can bind the interface even without the line, but it is a good practice to add it. Adding everything that must be bound to the options file makes it easy to bind the application from the command line. Add a line as shown in Listing 11.31. If you were to add options for the interface, you would specify them after the equal sign (=).

hrm.inProgress.pdqMethodStyle.HumanResourcesData =

Listing 11.31 - An entry for our interface in Default.bindProps

11.8.1.1.2 Run the StaticBinder utility.

Bind the interface by right-clicking on it in the Package Explorer view and selecting pureQuery, and then selecting Bind… This is shown in Figure 11.9. Then, choose the connection that you want to use for the bind and click Finish, as shown in Figure 11.10.
Figure 11.9 - Right-click on the interface and select `pureQuery`, and then select `Bind...`
11.8.2 Running an annotated-method style application in STATIC execution mode

pureQuery has a number of properties, called pureQuery properties, that control its behavior. Once you have bound your SQL statements in the database, you execute an annotated-method style application statically by simply setting the property pdq.executionMode to the value STATIC. The default value of this property is DYNAMIC. If your application attempts to execute an SQL statement that is not bound in the database while the property pdq.executionMode is set to the value STATIC, pureQuery throws an exception. Note that if you are using the STATIC execution mode,
and you add or change something in your interface, you need to rebind your interface before you can run methods in it again.

You can specify pureQuery properties either in a properties file, in a java.util.Properties object that you pass to a DataFactory.getData method, or as the value of the IBM Data Server Driver for JDBC and SQLJ property pdqProperties. Refer to the IBM Optim pureQuery Runtime documentation for details about specifying pureQuery properties.

One common way to specify a properties file is to provide a file named pdq.properties. If your application creates a connection to the database by using a javax.sql.DataSource object, this file must be in the classpath that indicates the location of the JDBC driver. If your application creates a connection by using the java.sql.DriverManager class, this file must be in the classpath. If your application creates a connection by using a javax.sql.DataSource object, another way to specify a properties file is to use a file named pdq.dataSourceName.properties in your application classpath, where dataSourceName represents the name of the DataSource object.

If you use a properties file to specify pureQuery properties, each line of a properties file should contain the name of a property, then an equal sign (=), and finally the value for that property.

11.8.2.1 In practice: Run the application statically
If you are using IBM® DB2® Database for your database, run your application statically.

We will specify our pureQuery properties in a file named pdq.dataSourceName.properties. The class hrm.HRMDatasourceFactory defines the javax.sql.DataSource object that Human Resources Manager uses. The name of the DataSource that the application uses is DataSourceHumanResourcesManager. Therefore, the name of our properties file is pdq.DataSourceHumanResourcesManager.properties. Locate this file in the src directory of the project, and set the pdq.executionMode property in it. In the project, a single classpath indicates the location of the source code and the location of the JDBC driver, so this file is in the classpath that indicates the location of the JDBC driver. Listing 11.32 shows what the file should contain.

```
pdq.executionMode=STATIC
```

**Listing 11.32 - The property to set in your properties file**

After you have set the property, run and test your application.
11.9 Advanced features

The annotated-method style provides many advanced features that are outside of the scope of this book. This list describes some of them. For details about using these features, refer to the IBM Optim pureQuery Runtime documentation.

- **Positioned updates and deletes:** You can associate \@Select annotated methods with \@Update methods to allow you to perform positioned updates and deletes on the query result created by an SQL \texttt{SELECT} statement.

- **Exception handling:** The annotated-method style and inline style APIs throw exceptions of the type \texttt{com.ibm.pdq.runtime.exception.DataRuntimeException}. Utility methods on these exceptions provide helpful information about the causing exception.

- **Hooks:** You can use implementations of the interface \texttt{com.ibm.pdq.runtime.statement.Hook} to provide methods that bracket the execution of the annotated and inline methods.

- **Batching SQL statements:** If you are using the IBM Data Server Driver for JDBC and SQLJ, you can use heterogeneous batching to execute multiple different SQL statements that do not return query results in a single trip to the database.

- **Queries over collections:** You can use the SQL syntax to perform queries over in-memory collections of data.

- **Combining the contents of two interfaces into a single package in the database:** You can create interfaces that extend multiple interfaces of annotated methods. When you bind the extending interfaces in the database, the StaticBinder utility puts the SQL statements from the different interfaces into the same package.

- **Specifying SQL statements in XML files:** You can specify an XML file to the Generator utility to provide the SQL statements for the annotated methods. This allows you to use a single interface for SQL statements that you need to execute against multiple databases, even if the SQL syntax varies between the databases. Use the option \texttt{-xmlFile} to specify this XML file.

11.10 Summary

The annotated-method style is an easy-to-use API for accessing databases. Under the annotated-method style, you define annotated methods in an interface that specify the SQL statements that you want processed, the input parameters that you will use to specify the
parameters to the SQL statement, and the object in which you want query results returned. Then, the pureQuery Generator utility generates implementation classes for your interface. Once an implementation class has been generated, you can use the `com.ibm.pdq.runtime.factory.DataFactory` class to create an instance of your interface. If you want to execute the SQL in your interface statically, use the StaticBinder utility to bind the SQL statements in your database, and then set the value of the `pdq.executionMode` property to STATIC (it is set to DYNAMIC by default).

### 11.11 Review questions

1. Which of the following must every annotated method have?
   - A. One of the following annotations: `@Cursor`, `@Handler`
   - B. One of the following annotations: `@Select`, `@Update`, `@Call`
   - C. A `String` parameter for specifying the SQL statement
   - D. A clear, descriptive comment

2. In an annotated method, where do you specify an SQL statements that you want to execute?

3. In an annotated method, how do you specify the values of the parameters of SQL statement?

4. For an annotated method that executes a `SELECT` statement, which of the following aspects should you consider when choosing the return type of your annotated methods?
   - A. The object that you want to contain each row
   - B. How many of the rows you want to return
   - C. The object that you want to contain all of the rows
   - D. All of the above

5. After you have created an interface of annotated methods, how do you create an implementation class for the interface?
   - A. Specify the interface as the `Class<T> interfaceClass` variable to a `get` method in the `com.ibm.pdq.runtime.factory.DataFactory` class.
   - B. Build your application. IBM Optim Development Studio runs the pureQuery Generator utility automatically to generate the class.
   - C. Write it yourself.
D. Specify the interface to a method named `generateImplementationClass` in the `com.ibm.pdq.runtime.Data` interface.

6. If you want to customize how pureQuery generates the implementation classes for your interfaces, you can specify the options that you want to use in an options file. In IBM Optim Development Studio, what is the name of that file?
   A. `Default.bindProps`
   B. `pdq.properties`
   C. `Default.genProps`
   D. `generator.properties`

7. When you run an annotated-method style application with the property `pdq.executionMode` set to the value `STATIC`, what happens if your application executes an SQL statement that is not bound in the database?
   A. The SQL statement executes statically
   B. The SQL statement executes dynamically
   C. pureQuery throws an exception
   D. None of the above

8. For an interface of annotated methods, the implementation classes that pureQuery generates implements another interface. What is the other interface? Why is this helpful?

9. What is one way in which you can close the `java.sql.Connection` object associated with an instance of an interface?

10. What are some of the benefits of using the annotated method style?

11. What would a user need to do in order to generate their own customized data-access code using the pureQuery tooling?

### 11.12 Exercise

In `hrm.inProgress.pdqMethodStyle.HumanResourcesManager`, the method `displayEmployeeReportChain` creates the "Employee Report Chain" screen. This screen needs to display the employees that report to the specified employee, as well as the specified employee’s manager. Look for the comments `// TODO: Add pureQuery code here.` in the method and add the necessary pureQuery code. There are two places in `displayEmployeeReportChain` that you need to update; one needs a new annotated method, and the other can use one of the existing annotated methods. Comments in the code show the SQL statements that you can use for the operations. The completed
version of the code is available in
hrm.completed.pdqMethodStyle.HumanResourcesManager and
hrm.completed.pdqMethodStyle.HumanResourcesData.

As described in Section 4.1.5, the “Employee Report Chain” screen displays the
employees that report to a specified employee, as well as the manager of the specified
employee. Add the necessary pureQuery code to
hrm.inProgress.pdqMethodStyle.HumanResourcesManager and
hrm.inProgress.pdqMethodStyle.HumanResourcesData to finish developing this
screen.

- The method displayEmployeeReportChain creates the “Employee Report Chain” screen.
- For your reference, the completed version of the code is available in
  hrm.completed.pdqMethodStyle.HumanResourcesManager and
  hrm.completed.pdqMethodStyle.HumanResourcesData.
- Locate the code in Listing 11.33 in the displayEmployeeReportChain
  method. You will need to add code that calls the method
  selectEmployeeManager that you added in Section 6.5.2.1, shown in
  Listing 11.17.

```java
// Get the manager of this employee.
// TODO: Add pureQuery code here.
EmployeeBean manager = null;
Listing 11.33 - You need to add a call to the annotated method shown
in Listing 11.17.
```

- Locate the code in Listing 11.34 in the displayEmployeeReportChain
  method. You need to add a call to a new annotated method that you add.
  The new annotated method needs to query the database for the
  employees that the specified employee directly manages. You can use
  the SQL statement in Listing 10.29, which is described in the Exercise
  10.9. Since you are adding a new annotated method, you need to use the
  DYNAMIC execution mode, or you need to rebind the interface.

```java
// Search the database for all the employees that this
employee
// directly manages and create a list of the employees
managed by
// this employee.
// TODO: Add pureQuery code here.
List<EmployeeBean> managedEmployees = null;
Listing 11.34 - You need to add a call to an annotated method that gets
the employees managed by the specified employee
```
Chapter 12 - Stored Procedures

This chapter describes how to use the pureQuery API to execute stored procedure.

12.1 Stored procedures: The big picture

pureQuery provides methods in the inline style and an annotation in the annotated-method style for executing SQL stored procedure calls. Through these methods, pureQuery provides a variety of ways in which it can return the results of your SQL stored procedure call. pureQuery returns the query results of stored procedures in a variety of formats. It also returns the values of OUT and INOUT parameters, and it updates java.util.Map objects and pureQuery beans that represent OUT and INOUT parameters with their changed values.

Figure 12.1 shows the possible return types of methods that execute stored procedures.
12.2 The StoredProcedureResult interface

Inline and annotated methods are both capable of returning an object of the type `com.ibm.pdq.runtime.StoredProcedureResult`. The interface `StoredProcedureResult` provides several methods that allow you to retrieve the results of your stored procedure in a variety of formats.

12.2.1 Getting the updated values of OUT and INOUT parameters

One of the methods in `StoredProcedureResult` is `getOutputParms`, which you can use to get the updated values of the OUT and INOUT parameters to your SQL stored procedure call.

The method `getOutputParms` returns an object array (Object[]) whose length is the number of parameters to the SQL stored procedure call. Each item in the array...
corresponds to one of the parameters to the stored procedure call, and the order of the parameters in the array is the same as the order of the parameters in the stored procedure call. pureQuery populates the items in the array that correspond to OUT and INOUT parameters with the updated values of those parameters. pureQuery does not return the IN parameters; instead, it populates the items that correspond to IN parameters with Java null as a placeholder.

If you want your application to call `getOutputParms()`, you must have it do so before the `StoredProcedureResult` object has been closed. pureQuery automatically closes the object when you retrieve the last query result. Therefore, if you want to get updated values of OUT and INOUT parameters, have your application call `getOutputParms()` before it retrieves any of the query results.

As an example, suppose that the SAMPLE database contains a procedure `EXAMPLE_PROCEDURE`, as defined in Listing 12.1:

```sql
CREATE PROCEDURE SAMPLE.EXAMPLE_PROCEDURE
    (IN parameter1 CHAR(8),
     OUT parameter2 SMALLINT,
     OUT parameter3 BIGINT)
LANGUAGE SQL
BEGIN
    /* do something here */
END
```

Listing 12.1 - The example stored procedure `EXAMPLE_PROCEDURE`

Assume that you use pureQuery to execute the procedure and create a `StoredProcedureResult` object named `storedProcedureResult` containing the results. You could store the output parameters in an array named `outputParameters` like this:

```java
Object[] outputParameters = storedProcedureResult.getOutputParms();
```

Observe that this stored procedure contains three parameters. Therefore, the array `outputParameters` would have a length of three. The first parameter to the stored procedure, `parameter1` is an IN parameter, so the value of `outputParameters[0]` would be null. The second parameter is an OUT parameters, so `outputParameters[1]` would contain the updated value of `parameter2`. Similarly, the third parameter is an OUT parameter, so `outputParameters[2]` would contain the updated value of `parameter3`.

12.2.2 Getting the query results of the stored procedure

Stored procedures can have one or more query results. When pureQuery creates a `StoredProcedureResult` object, it does not retrieve any query results from the database. The `StoredProcedureResult` interface contains several `getXXX` methods, each of which retrieves the next (or first) query result from the database and returns it in a
format specified by the name and parameters of the method. When you call a `getXXX` method and there is no next (or first) query result, the method closes the `StoredProcedureResult` object and returns `null`.

The query result formats that the `getXXX` methods in the `StoredProcedureResult` interface provide are very similar to the query result formats that the `queryXXX` methods of the `com.ibm.pdq.runtime.Data` interface provide. Use the following questions as a guide to help you select the right return type:

- **What format do you want to use for each row?**
  - A pureQuery bean or a scalar object: `getXXX` methods that have a parameter `Class<ROW> returnClass` store the contents of each row in an instance of `returnClass`.
    - If `returnClass` is a pureQuery bean, pureQuery creates an instance of `returnClass` and sets the values of its properties according to the values of the columns returned from the database. pureQuery uses the rules defined in Section 3.3.1 to determine the bean property that represents each column that is selected from the database. When pureQuery cannot find a property that represents a column in the SQL statement, pureQuery does not store the value of that column in the bean.
    - If `returnClass` is not a pureQuery bean, pureQuery constructs an object of type `returnClass` and attempts to store the contents of the first column of the query result in that object.
  - An object of a specific type, constructed by a `RowHandler<ROW>` object: `getXXX` methods that have a parameter `RowHandler<ROW> rowHandler` use `rowHandler` to create the object that contains each row.
  - A `java.util.Map<String, Object>` object: `queryXXX` methods that do not have a parameter `Class<ROW>` `returnClass` or a parameter `RowHandler<ROW>` `rowHandler` return the contents of each row in a `Map<String, Object>` object, in which the keys are the column labels, and the values are the contents of the columns.

- **What object do you want to use to contain all of the rows?**
  - An array: `getArray` methods return an array in which each entry represents a row.
A `java.util.List<E>` object: `getList` methods return a `java.util.List<E>` object in which each element represents a row.

A `java.util.Iterator<E>` object: `getIterator` methods return a `java.util.Iterator<E>` object for which each call to the Iterator's `next()` method returns a row. These methods do not fully materialize the information from the database during the initial method call. Instead, pureQuery retrieves one row each time the application calls the `next` method. This is particularly useful if your application retrieves large amounts of information from the database, and you do not want to store all of the information in memory at one time. When the application calls `next()` to retrieve all of the rows from the `Iterator`, pureQuery closes it automatically, freeing the associated database resources. However, if you have an application that does not retrieve all the rows, you must free the database resources by closing the `Iterator<E>` explicitly. Do so by casting the `Iterator<E>` object as a `com.ibm.pdq.runtime.ResultIterator<T>` object and calling its `close()` method, like this:

```java
((com.ibm.pdq.runtime.ResultIterator)
myIterator).close()
```

An object of a specific type, constructed by a `ResultHandler<RES>` object: `getQuery` methods use the parameter `ResultHandler<RES> handler` to construct the object that is returned. Chapter 8 discusses handlers. For this method, there is not a parameter for specifying the format for each row. Instead, `resultHandler` determines the format that is used for each row.

A `java.sql.ResultSet` object: `getResults` methods return an instance of `java.sql.ResultSet`.

### 12.2.3 Closing the StoredProcedureResult object

When you are finished using a `StoredProcedureResult` object, if you have not retrieved all of the query results, you should close it by calling its `close()` method. Doing so closes the underlying `java.sql.Statement` object, freeing the associated database resources.

### 12.3 Executing SQL stored procedure calls in the inline style

Recall that in the inline style, you execute SQL statements by using methods in the `com.ibm.pdq.runtime.Data` interface. The `Data` interface provides the following three
methods named call for executing SQL stored procedure calls. Each of the three
methods has a String sql parameter for specifying the SQL statement. Additionally,
each method has an Object... parameters varargs parameter for specifying the objects
that pureQuery needs to use to set the values of the parameters in the SQL statement.
The method shown in Listing 12.2 is the simplest call method to use, and is probably the
method that you will use the most frequently. It uses pureQuery's default rules for setting
the values of the parameter markers in the SQL statement and it constructs a
StoredProcedureResult object to describe the results of the SQL stored procedure call.

StoredProcedureResult call (String sql, Object... parameters)
Listing 12.2 - A call method that returns a StoredProcedureResult object

The method shown in Listing 12.3 uses pureQuery's default rules for setting the values of
the parameter markers in the SQL statement. It allows you to specify a
calchandlerWithParameters object that
pureQuery will use to create the object that the method returns.

<CAL> CAL call (String sql, CallHandlerWithParameters,CAL>
callHandlerWithParamters, Object... parameters)
Listing 12.3 - A call method that uses a CallHandlerWithParameters object

The method shown in Listing 12.4 allows you to specify a
com.ibm.pdq.runtime.handlers.ParameterHandler object, which pureQuery will
use to set the values of the parameters. It also allows you to specify a
calchandlerWithParameters object that
pureQuery will use to create the object that the method returns

<CAL> CAL call (String sql, ParameterHandler parameterHandler,
CallHandlerWithParameters,CAL> callHandlerWithParameters, Object... parameters)
Listing 12.4 - A call method that uses ParameterHandler and
CallHandlerWithParameters objects

pureQuery's default behaviors will probably meet your needs in most cases. However,
there may be cases in which you want some other behavior, and in these cases, you can
use handlers to achieve the behavior that you want.

After the execution of the stored procedure call, if the method does not specify a
CallHandlerWithParameters object, and if the SQL stored procedure has OUT or
INOUT parameters, pureQuery updates the corresponding properties in java.util.Map
objects or pureQuery beans provided as method parameters. pureQuery can do this only
when the varargs parameters are Maps and pureQuery beans. The parameter markers in
the SQL statement must indicate the properties in the method parameters. For inline
methods that specify a handler, the handler's handleCall method must update the
parameters, if you want them to be updated.
If you are using an Oracle database to execute a stored procedure, you will need to use the method shown in Listing 12.4 when you execute SQL stored procedure calls because the metadata that pureQuery needs to set SQL parameters and to create the StoredProcedureResult is not available for the stored procedures in Oracle® databases.

12.3.1 In practice: Use the inline style to execute a stored procedure -- Complete the “Increase Employee Bonuses” screen

This “Increase Employee Bonuses” screen uses the stored procedure bonus_increase in the SAMPLE database to increase the bonuses of all of the employees in the company. This operation is only available to the company president. Christine Haas is the president of the company, and her employee number is 000010.

When Christine Haas logs into Human Resources Manager, the fifth item in the main menu causes the application to call the increaseEmployeeBonuses method, which displays the “Increase Employee Bonuses” screen.

The first two parameters to the bonus_increase stored procedure are IN parameters. They are as follows:

1. p_bonusFactor DECIMAL(3,2) -- This is a multiplier between 1 and 2 by which all bonuses will be multiplied.

2. p_bonusMaxSumForDept DECIMAL(9,2) -- This is the maximum sum of bonuses in a single department. If the stored procedure determines that after the bonuses in any department were increased, the total of bonuses in that department would succeed the maximum sum, it makes no changes to the bonuses of the employees in that department.

The next four parameters are OUT parameters. They are as follows:

3. p_deptsWithoutNewBonuses VARCHAR(255) -- This contains a list of the departments in which the procedure did not increase bonuses because the total of bonuses exceeded the maximum sum.

4. p_countDeptsViewed INTEGER -- This indicates how many departments had their bonuses analyzed.

5. p_countDeptsBonusChanged INTEGER -- This indicates how many departments had their bonuses changed.

6. p_errorMsg VARCHAR(255) -- This contains an error message, if an error occurred.

The stored procedure also returns one query result in which each row contains the department number of an employee, his or her employee number, and his or her bonus.

The “Increase Employee Bonuses” screen asks the user for the multiplier and for the max sum. It then runs the stored procedure. Finally, it displays how many departments were
analyzed, how many were modified, and a table indicating for each department, the total of bonuses and whether the bonuses were modified.

Let's now write the inline style implementation of this method. Edit the `increaseEmployeeBonuses` method in `hrm.inProgress.pdqInlineStyle.HumanResourcesManager`. We do not need to use any handlers, so we use this method in Listing 12.2.

In determining how to specify the SQL stored procedure call, we need to choose how we will specify the parameter markers. Refer to Section 5.4, which describes parameter markers. We use variables of the type `double` to store the values of `p_bonusFactor` and `p_bonusMaxSumForDept`, so we must use either the `?` or the `?n` type. We choose the `?` type. Because the SQL statement contains parameter markers for the OUT parameters, we must specify corresponding parameters in the varargs parameters. They are OUT parameters, so there is no input value. Therefore, we specify `null`.

Add the code in Listing 12.5 to execute the stored procedure in the inline method style:

```java
// Execute the bonus_increase stored procedure.
storedProcedureResult = data.call("CALL bonus_increase (?, ?, ?, ?, ?, ?)",
   bonusFactor, bonusMaxSumForDepartment, null, null, null, null);
```

**Listing 12.5 - Use an inline method to execute the stored procedure**

The first thing that the application needs to do after executing the stored procedure call is retrieve the OUT parameters from `storedProcedureResult`, since pureQuery will automatically close `storedProcedureResult` when the last query result is retrieved. Add the code in Listing 12.6, which retrieves the array of OUT and INOUT parameters, and assigns the new value of the parameters to variables.

```java
// Get the values of the OUT parameters from bonus_increase.
// Do this before getting the query results, since getting the last // query result closes storedProcedureResult.
Object[] outputParameters = storedProcedureResult.getOutputParms ();
String departmentsWithoutNewBonuses = (String) outputParameters[2];
int countDepartmentsViewed = (Integer) outputParameters[3];
int countDepartmentsBonusChanged = (Integer) outputParameters[4];
String errorMessage = (String) outputParameters[5];
```

**Listing 12.6 - Get the updated values of the OUT parameters from the stored procedure**

After we have retrieved the OUT parameters, we access the query result. The query result contains a row for each employee in the company. At a large company, this could be more data than we would want to have in memory at any point in time, so we use a `java.util.Iterator&lt;E&gt;` object to retrieve one row at a time from the database. Add
the code in Listing 12.7, which creates an `Iterator<EmployeeBean>` object containing
the query result.

```java
// Get the query result, which contains the bonuses for all the
// employees.
// Do this after getting the value of the OUT parameters from
// bonus_increase, since getting the last query result closes
// storedProcedureResult.
employees = storedProcedureResult.getIterator (EmployeeBean.class);
```

Listing 12.7 - Get the query result from the stored procedure call

`pureQuery` automatically closes the underlying database resources when the application
gets the last query result and when the application gets the last row of the
`Iterator<EmployeeBean>` object, so in most cases our application will not need to
close `employees` and `storedProcedureResult`. However, an exception could occur
before the application retrieves the query result or processes all of the rows, so it would be
good for our application to close `employees` and `storedProcedureResult` explicitly.

Add the code in Listing 12.8 to the `finally` block.

```java
finally {
    // Close employees. This frees the associated database resources.
    if (null != employees) ((ResultIterator<EmployeeBean>) employees).close ();
    // Close storedProcedureResult. This frees the associated database
    // resources.
    if (null != storedProcedureResult) storedProcedureResult.close ();
}
```

Listing 12.8 - Close `employees` and `storedProcedureResult`

Finally, test the application by logging in with the employee number 000010 and selecting
the option 5 on the main menu.

### 12.4 Executing SQL stored procedure calls in the annotated-method style

Annotated methods that execute SQL stored procedure calls must have the `@Call`
annotation. The return type of the method can be one of the following:

- A `com.ibm.pdq.runtime.StoredProcedureResult` object
- void
- One of several of the return types allowed with the `@Select`
  annotation
- An object of a specific type, constructed by a
  `com.ibm.pdq.runtime.handlers.CallHandlerWithParameters <CAL>` object
Sections 6.4 and 6.5 provide detailed instructions on how to write an annotated method, including a method that executes an SQL stored procedure call.

After the execution of the stored procedure call, if the annotated method does not specify a `CallHandlerWithParameters<CAL>`, and if the SQL stored procedure has OUT or INOUT parameters, pureQuery updates the corresponding properties in `java.util.Map` objects or pureQuery beans provided as method parameters. pureQuery can do this only when the parameters to the annotated method are `Map` objects and pureQuery beans. The parameter markers in the SQL statement must indicate the properties in the method parameters. For annotated methods that specify a `CallHandlerWithParameters<CAL>` object, the handler’s `handleCall` method must update the parameters, if you want them to be updated.

If you are using an Oracle database, you will need to specify a `ParameterHandler` object and a `CallHandlerWithParameters<CAL>` object when you execute SQL stored procedure calls because the metadata that pureQuery needs to set SQL parameters and to create the `StoredProcedureResult` is not available for the stored procedures in Oracle® databases.

12.4.1 In practice: Use the annotated-method style to execute a stored procedure -- Complete the “Increase Employee Bonuses” screen

Now let’s complete the “Increase Employee Bonuses” screen for the method-style version of our application. Section 12.3.1 describes the `bonus_increase` stored procedure. Edit `hrm.inProgress.pdqMethodStyle.HumanResourcesData`, along with the `increaseEmployeeBonuses` method in `hrm.inProgress.pdqMethodStyle.HumanResourcesManager`.

To develop the “Increase Employee Bonuses” screen, we will need updated values of the OUT parameters, as well as the stored procedure’s query result. The standard pureQuery behavior provides this functionality, so we do not need to use a `CallHandlerWithParameters<CAL>`. Instead, we can specify the annotated method in one of these two ways:

- We can have the query result be a `StoredProcedureResult` object. We can use simple objects and primitives for the method parameters. The `StoredProcedureResult` object allows us to get the value of OUT and INOUT parameters, so we do not need to update the values of the method parameters.

- There is only one query result, so we can have the return type of the annotated method be the format in which we want the query result. In this case, we would need to rely on pureQuery’s updating of method parameters to get the updated values of the OUT parameters. In order for
this to happen, we need to specify the parameters in java.util.Map objects or pureQuery beans.

We choose to use the first approach. You can implement the second approach as an exercise if you wish.

As for the inline style, we will specify the values of the IN parameters by using variables of the type double, and we will use the ? type of parameter markers. We specify parameters for all of the stored procedure parameters, but what we specify for the OUT parameters does not matter, as long as some parameter is referenced by the parameter marker. Add the annotated method in Listing 12.9 to the hrm.inProgress.pdqMethodStyle.HumanResourcesData interface and build the project so that the pureQuery Generator regenerates the implementation class.

```java
// Execute the bonus_increase stored procedure.
@Call(sql = "CALL bonus_increase (?, ?, ?, ?, ?, ?)"")
StoredProcedureResult callBonusIncrease (double bonusFactor,
    double bonusMaxSumForDepartment, String departmentsWithoutNewBonuses,
    int countDepartmentsViewed, int countDepartmentsBonusChanged, String errorMessage);
Listing 12.9 - An annotated method that executes the stored procedure
```

Add code to the increaseEmployeeBonuses method in hrm.inProgress.pdqMethodStyle.HumanResourcesManager that calls the annotated method. The values for the parameters that represent the OUT parameters do not matter, so specify null and 0. Listing 12.10 shows the completed code.

```java
// Execute the bonus_increase stored procedure.
storedProcedureResult = data.callBonusIncrease (bonusFactor,
    bonusMaxSumForDepartment,
    null, 0, 0,
    null);
Listing 12.10 - Code that calls the annotated method callBonusIncrease
```

After the StoredProcedureResult object is created, the application must get the OUT parameters first, and then the query results, just as in inline style. Finally, the application should close the Iterator<EmployeeBean> object and storedProcedureResult. The procedure for doing all of these things is identical to that in the inline method style, and the code is the same, too. Refer to Section 12.3.1 to add the necessary code. Finally, test the application by logging in with the employee number 000010 and selecting the option 5 on the main menu.

### 12.5 Summary

Both the inline style and the annotated-method style provide ways to execute SQL stored procedure calls. The inline style provides methods named call in the com.ibm.pdq.runtime.Data interface for this purpose. The annotated-method style provides the com.ibm.pdq.annotation.Call annotation for placing on methods to
have them execute stored procedures. When the parameters to the method are
`java.util.Map` and pureQuery bean objects, and no
`com.ibm.pdq.runtime.handlers.CallHandlerWithParameters<CAL>` object is
specified, pureQuery updates the properties of the `Map` or bean objects that represent OUT
and INOUT parameters to the stored procedure. pureQuery can return the results of a
stored procedure call as a `com.ibm.pdq.runtime.StoredProcedureResult` object,
which contains methods for retrieving the updated values of OUT and INOUT parameters,
as well methods for retrieving the query results in various formats. Both inline methods
and annotated methods can return a `.StoredProcedureResult` object. Additionally,
annotated methods can have most of the return types available to methods with the
`@Select` annotation, if they only need to return one query result from the stored
procedure. When none of pureQuery built-in return types provide what your application
needs, a `com.ibm.pdq.runtime.handlers.CallHandlerWithParameters<CAL>`
opt can specify the return type.

### 12.6 Questions

1. How do you execute a stored procedure call in the inline style?
2. How do you declare an annotated method that executes a stored procedure call?
3. When you use certain formats to represent your OUT and INOUT parameters,
pureQuery can update the objects with the new values. Which of the following
formats can pureQuery update?
   - Properties of `java.util.Map` objects
   - Scalars
   - Properties of pureQuery beans
   - Elements of `java.util.List` objects
4. Which of the following information do
   `com.ibm.pdq.runtime.StoredProcedureResult` objects provide?
   - How long the stored procedure took to run
   - The updated values of OUT and INOUT parameters in an `Object` array
     (`Object[]`)
   - Query results
   - The names of the tables that were affected by the stored procedure
5. True or false: When you use a
   `com.ibm.pdq.runtime.StoredProcedureResult` object to get the results of
   your stored procedure call, you should get the query results before you call the
   `getOutputParms()` method.
6. A `com.ibm.pdq.runtime.StoredProcedureResult` object's `getOutputParms()` method returns an `Object` array (`Object[]`). Which of the following describes the treatment of IN parameters with regards to this array?

   A. IN parameters are not included in the array. The size of the array is equal to the number of OUT and INOUT parameters.

   B. The items in the array that correspond to IN parameters are populated with `null`.

   C. The items in the array that correspond to IN parameters are populated with the values that were provided as input.

   D. None of the above

7. Imagine that you execute a stored procedure in which the fourth parameter is an INOUT parameter that you represent with an `Integer` object. Provide the code you use to get the updated value of that parameter. Assume that the method that you use returns a `com.ibm.pdq.runtime.StoredProcedureResult` object.

8. How do you free the database resources associated with a `com.ibm.pdq.runtime.StoredProcedureResult` object?

9. In the annotated-method style, if you only want to have `pureQuery` return one query result from your stored procedure call, you can declare the return type of the method to be the format in which you want to retrieve the query result. Which of these types are valid for the return type of such a method?

   A. `java.util.Iterator<EmployeeBean>`

   B. `java.sql.ResultSet`

   C. `java.util.List<DepartmentBean>`

   D. `java.util.Map<String,Object>`

   E. All of the above

10. If none of `pureQuery`'s ways of returning query results meet the needs of your application, what type of object can you use to define how a method returns the query results?

### 12.7 Exercise

Section 12.4.1 described two ways in which we could declare the annotated method that executes the `bonus_increase` stored procedure. In that section, we used the way in which the return type of the annotated method is `com.ibm.pdq.runtime.StoredProcedureResult`. Now, modify the method to use the other way, in which the return type of the method is `java.util.Iterator<EmployeeBean>`. You will need to use a
Getting Started with pureQuery

256

java.util.Map<String, Object> object or pureQuery bean to specify the parameters so that you can retrieve the updated values of the OUT and INOUT parameters.

```java
Iterate<EmployeeBean> callBonusIncrease (Map<String, Object> parameters);
```

Listing 12.11 - The annotated method that would need to be added to hrm.inProgress.pdqMethodStyle.HumanResourcesData

```java
// Screen: "Increase Employee Bonuses"
private void increaseEmployeeBonuses (HumanResourcesData data) {
    // Repeatedly display the "Increase Employee Bonuses" screen until the
    // user selects "0". This means that every time the user returns to
    // screen from another one, this screen will be displayed again.
    boolean userSelectedReturnToPreviousScreen = false;
    while (!userSelectedReturnToPreviousScreen) {
        outputWriter.println();
        outputWriter.println("--------------- Increase Employee Bonuses ---
-----------");
        outputWriter.println();
        outputWriter.println("Increasing bonus sizes for all employees in
the company");

        // Ask the user for the bonus factor.
        Double bonusFactor = null;
        while (null == bonusFactor) {
            String bonusFactorString = getLineFromConsole("You can increase
the size of bonuses by multiplying all the\n" + "bonuses by a number between 1 and 2. Please enter the\n" + "multiplier that you want the application to use: ");
            try {
                bonusFactor = Double.parseDouble(bonusFactorString);
            } catch (NumberFormatException e) {
                outputWriter.println("The value \"" + bonusFactor + \\
" is not a valid number.");
            }
        }
    }
```
// Ask the user for the maximum department sum.
Double bonusMaxSumForDepartment = null;
while (null == bonusMaxSumForDepartment) {
    String bonusMaxSumForDepartmentString = getLineFromConsole
      ("Employee bonuses will be automatically modified, as long as the total of the bonuses in a department does not exceed some limit that you set. Please enter the limit that you want the application to use:");
    try {
        bonusMaxSumForDepartment = Double.parseDouble(bonusMaxSumForDepartmentString);
    } catch (NumberFormatException e) {
        outputWriter.println ("The value \" + bonusMaxFactor + \" is not a valid number.");
    }
}

// Create a map for the parameters.
Map<String, Object> parameters = new HashMap<String, Object>();
parameters.put ("bonusFactor", bonusFactor);
parameters.put ("bonusMaxSumForDepartment", bonusMaxSumForDepartment);

Iterator<EmployeeBean> employees = null;

try {
    // Execute the bonus_increase stored procedure and get the query result, which contains the bonuses for all the employees.
    employees = data.callBonusIncrease (parameters);

    // Get the values of the OUT parameters from bonus_increase.
    String departmentsWithoutNewBonuses = (String) parameters.get ("departmentsWithoutNewBonuses");
    int countDepartmentsViewed = (Integer) parameters.get ("countDepartmentsViewed");
    int countDepartmentsBonusChanged = (Integer) parameters.get ("countDepartmentsBonusChanged");
    String errorMessage = (String) parameters.get ("errorMessage");

    // We will display the total of bonuses in each department. Create a map containing the totals.
    TreeMap<String, Integer> bonusesByDepartment = null;
if (null != employees) {
    bonusesByDepartment = new TreeMap<String, Integer> ();
    while (employees.hasNext ()) {
        // Store the information about this employee.
        EmployeeBean employee = employees.next ();
        String departmentNumber = employee.getDepartmentNumber ();
        String employeeNumber = employee.getEmployeeNumber ();
        double bonus = employee.getBonus ();

        // Add the bonus to the total for its department.
        Integer departmentTotal = bonusesByDepartment.get (departmentNumber);
        if (null == departmentTotal) departmentTotal = 0;
        departmentTotal += bonus;
        bonusesByDepartment.put (departmentNumber, departmentTotal);

        // If this employee is the logged-in employee, update the
        // employee's EmployeeBean instance with the new bonus.
        if (null != employeeNumber && employeeNumber.equals (employeeUsedForLogin.getEmployeeNumber ()))
            employeeUsedForLogin.setBonus (bonus);
    }
}

// Print how many departments were analyzed, and how many were updated.
// Also print the error message, if there was one.
outputWriter.println ();
outputWriter.println ("Bonuses were analyzed for " +
countDepartmentsViewed + " departments.");
outputWriter.println ("Bonuses were updated in " +
countDepartmentsBonusChanged + " departments.");
outputWriter.println ();
if (null != errorMessage && 0 < errorMessage.length ()) {
    outputWriter.println ("The following error occurred:");
    outputWriter.println (" " + errorMessage);
}

// Print the total of bonuses in each department.
if (null != bonusesByDepartment) {
    outputWriter.println ("Employee Bonuses");
    outputWriter.println ("By Department");
outputWriter.println("-----------------------------------------
-----");
outputWriter.println( "Department | Were Bonuses |
Total");
outputWriter.println( "Number | Updated? | |
Bonuses");
outputWriter.println( "-----------|-----------|---------
-----");
for (Entry<String, Integer> department : bonusesByDepartment.entrySet()) {
    String departmentNumber = department.getKey();
    outputWriter.print("      "+departmentNumber+"      |");
    if ((null != departmentsWithoutNewBonuses) &&
    (!departmentsWithoutNewBonuses.contains(departmentNumber)))
        outputWriter.print("      Yes     ");
    else
        outputWriter.print("      No      ");
    outputWriter.println("|  " + Formats.currencyFormat.format(department.getValue()));
}
}
finally {
    // Close employees. This frees the associated database resources.
    if (null != employees) ((ResultIterator<EmployeeBean>) employees).close();
}

// Construct the menu of choices.
StringBuilder choiceBuilder = new StringBuilder();
choiceBuilder.append("What would you like to do?
");
choiceBuilder.append("    0. Return to the previous menu
");
choiceBuilder.append("    1. Increase employee bonuses again
");

// Display the menu, and ask the user to choose one of the options.
int selection = getSelectionFromConsole(choiceBuilder.toString(), 1);

// Go to the previous screen.
userSelectedReturnToPreviousScreen = (0 == selection);
}

Listing 12.12 - The modified version of the increaseEmployeeBonuses method
Chapter 13 - Handlers

In this chapter, you will learn how to use handlers to modify how pureQuery returns query results, and how pureQuery sets the values of SQL parameters.

13.1 Handlers: The big picture

pureQuery provides a wide variety of ways to specify parameters for the SQL statements that you execute, along with many formats for returning query results. However, there might be times when the built-in functionality does not meet the needs of your application. In these cases, you can use handlers to customize various aspects of pureQuery processing by writing classes that implement the handler interfaces. pureQuery provides the following types of handlers:

Customizing input:
- `com.ibm.pdq.runtime.handlers.ParameterHandler`: You can use a ParameterHandler object to determine how the pureQuery sets the values of parameters in prepared SQL statement. The interface defines a method named `handleParameters` that pureQuery calls to set the values of the parameters in the SQL statement.

Customizing output:
- `com.ibm.pdq.runtime.handlers.RowHandler<ROW>`: You can use a RowHandler<ROW> object to determine how pureQuery returns each row of a query result. Use this when you want to use one of pureQuery's built-in objects for returning all the rows, but you want to modify the way pureQuery returns each row. The RowHandler<ROW> interface defines a method named `handle`. pureQuery processes each row of the query result by calling this method. The `handle` method returns an object of type ROW, which pureQuery uses to represent the row in the object that the annotated or inline method returns. If you are using the annotated-method style to execute an SQL INSERT statement, and if you use a single pureQuery bean to provide the values of the SQL parameters, pureQuery can update properties in the bean with values generated by
the database. You can specify a `RowHandler<ROW>` implementation to customize how pureQuery updates the properties in the bean.

- **com.ibm.pdq.runtime.handlers.ResultHandler<RES>:** You can use a `ResultHandler<RES>` to determine how pureQuery returns the entire query result. Like the `RowHandler<ROW>` interface, the `ResultHandler<RES>` interface defines a method named `handle`. pureQuery calls this method to process the entire query result, and the annotated or inline method returns the object of type `RES` that the `handle` method returns.

- **com.ibm.pdg.runtime.handlers.CallHandlerWithParameters<CAL>:** If your SQL is a stored procedure call, you can use a `CallHandlerWithParameters<CAL>` object to determine how pureQuery returns the results of the stored procedure. The `CallHandlerWithParameters<CAL>` method defines a method named `handleCall`. pureQuery calls this method to process the results of the stored procedure call, including query results and OUT and INOUT parameters. The annotated or inline method returns the object of type `CAL` that is returned by this method.
Figure 13.1 provides a high-level view of how you can use handlers to modify the operations of pureQuery.

**Handler that customizes input:**

User input: SQL and Parameters → pureQuery Annotated or Inline method → User-defined ParameterHandler → Prepared SQL statement Used by pureQuery

**Handlers that customize output:**

SQL statement Query result → User-defined ResultHandler<RESULT> → ResultType

SQL statement Query result → User-defined RowHandler<ROW> → pureQuery Processing → List<ROW>

SQL CALL Statement Query results → User-defined CallHandlerWithParameters<CALLRESULT> → CallResultType

Figure 13.1 - An overview of the way pureQuery uses handlers

pureQuery provides three example ResultHandler<RES> implementations. These are:

- `com.ibm.pdq.runtime.data.handlers.IteratorPagingResultHandler<T>`: Returns an Iterator object for one page of the results of an SQL query.
- `com.ibm.pdq.runtime.data.handlers.JSONResultHandler`: Returns the query result in the JSON format.
- `com.ibm.pdq.runtime.data.handlers.XMLResultHandler`: Returns the query result in XML.

**13.2 Specifying handlers in the inline style**

When you want to use handlers in the inline method style, specify the handlers as parameters to the inline methods in the Data interface.

Customizing input:
If you are executing an SQL `SELECT` statement, and you want to customize the way pureQuery sets the parameters to the SQL statement, use a query method that accepts an object of type `ParameterHandler` as a parameter. Note that this method also has a `ResultHandler<RES>` parameter.

If you are executing an SQL stored procedure call, and you want to customize the way pureQuery sets the parameters to the SQL call statement, use the call method that accepts an object of type `ParameterHandler` as a parameter. Note that this method also has a `CallHandlerWithParameters<CAL>` parameter.

Customizing output:

- If you are executing an SQL `SELECT` statement, and you want to customize how each row is returned, use one of the `queryXXX` methods that accept an object of type `RowHandler<ROW>` as a parameter. Use the `queryXXX` method that has the return type that you want.
- If you are executing an SQL `SELECT` statement, and you want to use a type of object to contain all of the rows that is not one of pureQuery’s built in objects, use one of the `query` methods. These methods have a `ResultHandler<RES>` parameter.
- If you are executing an SQL stored procedure call, and you want to use an object other than pureQuery’s `com.ibm.pdq.runtime.StoredProcedureResult` to contain the query results, use one of the `call` methods that accepts an object of type `CallHandlerWithParameters<CAL>` as a parameter.

13.2.1 In practice: Specify a handler using the inline style -- Complete the “Company Employees” screen

The “Company Employees” screen displays all of the employees in the company in pages of 10 employees each. Let’s use the `IteratorPagingResultHandler<T>` to retrieve the employees, one page at a time. Modify the `displayAllEmployees` method in `hrm.inProgress.pdqInlineStyleWithHandlers.HumanResourcesManager`. You will also need to modify the utility method `selectPageOfEmployees`, which is located in the same class.

13.2.1.1 Use pureQuery to create an `Iterator<EmployeeBean>` object that contains the information about the employees

In order to get information about all the employees in the company, we need to execute a query against the database. We will use the query shown in Listing 13.1, which will select all the employees and sort them by their names.
SELECT * FROM EMPLOYEE ORDER BY LASTNAME, FIRSTNME, MIDINIT

Listing 13.1 - An SQL query for selecting all the employees in the company

We need to use one of the `query` methods of the `Data` interface because we want to specify a `ResultHandler<RES>` object. While users are viewing the list of employees, they can request to view additional information about an employee and then return to the list. Therefore, we need to set the cursor holdability to “hold cursors over commit.” As a result, we use a `queryIterator` method that has parameters for cursor attributes.

We need to pass the method an `IteratorPagingResultHandler<T>` object. Observe the constructors available for the `IteratorPagingResultHandler<T>`, as described in the Javadoc for the class. We will use the constructor shown in Listing 13.2. This constructor allows us to specify the page number that we want, the number of rows that we want to go on a page, and the class in which we want each row to be returned.

```
IteratorPagingResultHandler(int pageNumber, int pageSize, Class<T> beanClass)
```

Listing 13.2 - The `IteratorPagingResultHandler<T>` constructor that our application will use

Modify `hrm.inProgress.pdqInlineStyleWithHandlers.HumanResourcesManager` to use the handler to create `Iterator<EmployeeBean>` objects. There are two places in the `displayAllEmployees` that need to get a page of employees: the place that gets the first page of employees, and the place that gets each additional page. Therefore, both of these places call the utility method `selectPageOfEmployees`. Modify the utility method as shown in Listing 13.3 so that it returns a page of employees.

```
private Iterator<EmployeeBean> selectPageOfEmployees (Data data, int pageNumber)
{
    // Use "hold cursors over commit" so that if the user goes to another
    // screen and comes back, we can continue displaying employees.
    return data.query (ResultSet.TYPE_FORWARD_ONLY,
    ResultSet.CONCUR_READ_ONLY,
    ResultSet.HOLD_CURSORS_OVER_COMMIT,
    "SELECT * FROM EMPLOYEE ORDER BY LASTNAME, FIRSTNME, MIDINIT", new
    IteratorPagingResultHandler<EmployeeBean> (pageNumber, 10,
    EmployeeBean.class));
}
```

Listing 13.3 - Code for getting a page of employees from the database

Add code to the finally block in `displayAllEmployees` to close `employees`. This is to make sure that the database resources are freed if an exception occurs before our application retrieves all of the rows from `employees`. Listing 13.4 shows the necessary code.

```
finally {
```

```
```
// Close employees. This frees the associated database resources.
if (null != employees) {((ResultIterator<EmployeeBean>) employees).close();
}

Listing 13.4 - Close employees. This frees the database resources if an exception occurs before the application retrieves all of the rows.

Finally, test the application.

13.3 Specifying handlers in the annotated method style

When you use the annotated-method style, you can specify handlers for customizing the input and output of your annotated methods.

When you specify handlers for annotated methods, make sure that the return type of your method is compatible with your handler. The Java compiler is not able to recognize if the type of a handler is not compatible with the return type of the method. If the type is not compatible, you will not know there is a problem until you generate an implementation class for your interface and observe that it does not compile.

The annotated-method style provides two ways of specifying handlers. You can specify handlers either in the \texttt{com.ibm.pdq.annotation.Handler} annotation, or as parameters to the annotated methods. For an annotated method, you can specify at most one handler that customizes output for each annotated method. You can also specify the \texttt{ParameterHandler}, which customizes input, if you want.

13.3.1 Specifying handlers by using the \texttt{@Handler} annotation

pureQuery provides the annotation \texttt{com.ibm.pdq.annotation.Handler}, which you can add to annotated methods to indicate the handlers that you want them to use. The annotation contains an element for each type of handler. To specify that you want to use a specific handler, set the appropriate annotation element to the class of the handler that you want to use.

Listing 13.5 shows an example of specifying the \texttt{JSONResultHandler} for an annotated method. Note that the return type of the annotated method is \texttt{String} because the \texttt{handle} method of the \texttt{JSONResultHandler} has a return type of \texttt{String}.

\begin{verbatim}
@Handler(resultHandler=JSONResultHandler.class)
@Select(sql = "SELECT * FROM EMPLOYEE ORDER BY LASTNAME, FIRSTNAME, MIDINIT")
String selectAllEmployees ();
\end{verbatim}

Listing 13.5 - An example of using the \texttt{@Handler} annotation to specify a handler

When you specify a handler with the \texttt{@Handler} annotation, the handler must have a public no-argument constructor. pureQuery creates an instance of the handler by using that
constructor, and uses the same instance every time your application calls the annotated methods. pureQuery recommends that you use this approach as much as you can because your application will not have to manage instances of the handler, and only one instance of the handler will be instantiated, no matter how many times your application invokes the annotated method. However, you might have handlers that do not have public no-argument constructors. You also might have handlers that store information about the query result of a query, preventing you from using a single handler for multiple invocations of an annotated method. For these handlers, you will need to specify the handlers as parameters to the annotated methods.

13.3.2 Specifying handlers as method parameters

When you want to specify a handler for an annotated method, if you do not want to (or cannot) specify it by using the @Handler annotation, specify it as a parameter to the annotated method. When you specify handlers as parameters for an annotated method, they must be the last parameters to the method. That is, you must first provide any parameters that provide the information for setting the parameters in the SQL statement, and then you can provide one or two handlers. Listing 13.6 shows an example of specifying the IteratorPagingResultHandler for an annotated method.

```java
@Select(sql = "SELECT * FROM EMPLOYEE WHERE WORKDEPT = ?")
Iterator<EmployeeBean> selectEmployeesInDepartment (String workDept,
    IteratorPagingResultHandler<EmployeeBean> resultHandler);
```

Listing 13.6 - An example of specifying a handler as a parameter to an annotated method

13.3.2.1 In practice: Add an annotated method to HumanResourcesData that executes the SQL and uses a handler for the query result

The “Company Employees” screen displays all of the employees in the company in pages of 10 employees each. Let’s use the IteratorPagingResultHandler<T> to retrieve the employees, one page at a time. Modify the displayAllEmployees method in hrm.inProgress.pdqMethodStyleWithHandlers.HumanResourcesManager, along with the utility method selectPageOfEmployees. You will also need to modify the interface hrm.inProgress.pdqMethodStyleWithHandlers.HumanResourcesData. Refer to Section 8.2.1 for the SQL statement that we need to execute.

13.3.2.1.3 Specify annotated method

Declare an annotated method that executes the SQL statement and uses the IteratorPagingResultHandler<T> to return the results. The IteratorPagingResultHandler<T> does not have a no-argument constructor, so we will specify it as a parameter to the annotated method. The return type of the handle method is ResultIterator<T>. We can use that as the return type of our method, or we can use Iterator<E>, which is a superinterface of ResultIterator<T>. We
choose to use `Iterator<E>`. As in the inline style, we need to set the cursor holdability to "hold cursors over commit." We use the `com.ibm.pdq.annotation.Cursor` annotation to specify the holdability cursor attribute `java.sql.ResultSet.HOLD_CURSORS_OVER_COMMIT`. Add the declaration as shown in Listing 13.7 and build the project so that the implementation class is generated.

```java
// Search the database for the specified page of employees
// and create an Iterator<EmployeeBean> object to retrieve them.
// Use "hold cursors over commit" so that if the user goes to another
// screen and comes back, we can continue displaying employees.
@Cursor(holdability = ResultSet.HOLD_CURSORS_OVER_COMMIT)
@Select(sql = "SELECT * FROM EMPLOYEE ORDER BY LASTNAME, FIRSTNME, MIDINIT")
Iterator<EmployeeBean> selectEmployees (IteratorPagingResultHandler<EmployeeBean> resultHandler);
```

Listing 13.7 - The annotated method that gets a page of employees

13.3.2.1.4 Modify the `selectPageOfEmployees` method in to use the `IteratorPagingResultHandler<T>`

There are two places in the `displayAllEmployees` that need to get a page of employees: the place that gets the first page of employees, and the place that gets each additional page. Therefore, both of these places call the utility method `selectPageOfEmployees`. Modify the utility method as shown in Listing 13.8 so that it calls the new annotated method.

```java
private Iterator<EmployeeBean> selectPageOfEmployees (Data data, int pageNumber)
{
    return data.selectEmployees (new IteratorPagingResultHandler<EmployeeBean> (pageNumber,10,EmployeeBean.class));
}
```

Listing 13.8 - Code for getting a page of employees from the database

13.3.2.1.5 Modify the `displayAllEmployees` method to close `employees`

As in Section 8.2.1.1, add code to the finally block in `displayAllEmployees` to close `employees`. Listing 13.4 shows the necessary code. Finally, test the application.

13.4 Writing handler implementation classes

To create your own handler, write a class that implements the appropriate interface.
13.4.1 Writing a ResultHandler<RES> implementation

The ResultHandler<RES> interface has the method handle, shown in figure Listing 13.9.

```java
RES handle(ResultSet resultSet);
```

Listing 13.9 - The handle method in the ResultHandler<RES> interface

The parameter to the handle method is a java.sql.ResultSet object. This is the query result of the SQL statement. Your implementation of the handle method must use resultSet to create and return the object that you want your annotated or inline method to return.

As an example, imagine that you wanted an annotated or inline method to return a String describing the query result. Spaces should separate the values of the columns, and line breaks should separate the rows. This would probably not be a very useful way to return a query result, but we are using this just as a simple example. Listing 13.10 shows how you could implement such a handler.

```java
import java.sql.ResultSet;
import java.sql.SQLException;

import com.ibm.pdq.runtime.handlers.ResultHandler;

public class SimpleStringResultHandler implements ResultHandler<String> {
    public String handle (ResultSet resultSet) {
        StringBuilder result = new StringBuilder ();
        try {
            int columns = resultSet.getMetaData ().getColumnCount ();
            while (resultSet.next ()) {
                for (int i = 0; i < columns; i++) {
                    result.append (resultSet.getString (i) + " ");
                }
                result.append ("\n");
            }
        } catch (SQLException e) {
            throw new RuntimeException ("An error occurred while processing the query result.", e);
        }
        return result.toString ();
    }
}
```

Listing 13.10 - An example implementation of ResultHandler<RES>
13.4.2 Writing a `RowHandler<ROW>` implementation

The `RowHandler<ROW>` interface has the method `handle`, shown in Listing 13.11

```
ROW handle (ResultSet resultSet, ROW object) throws SQLException;
```

Listing 13.11 - The `handle` method in the `RowHandler<ROW>` interface

One of the parameters to the `handle` method is a `java.sql.ResultSet` object. This is the query result of the SQL statement. When `pureQuery` calls the `handle` method, the cursor of `resultSet` will be pointing to the row that the `ResultHandler<ROW>` must process. Your implementation of the `handle` method must create and return the object that will contain that row. Do not call the `resultSet.next()` method in your implementation of the `handle` method. If the implementation is to be used for SQL `SELECT` statements, the `handle` method must create, populate, and return an instance of type `ROW`.

As an example of a `RowHandler<ROW>` implementation, imagine that you wanted an annotated or inline method to return each row of the query result a `String` in which spaces separate the values of the columns. Listing 13.12 shows how you could implement such a handler.

```
import java.sql.ResultSet;
import java.sql.SQLException;
import com.ibm.pdq.runtime.handlers.RowHandler;

public class SimpleStringRowHandler implements RowHandler<String> {
    public String handle (ResultSet resultSet, String object) throws SQLException {
        StringBuilder result = new StringBuilder ();
        int columns = resultSet.getMetaData ().getColumnCount ();
        for (int i = 0; i < columns; i++)
            result.append (resultSet.getString (i) + " ");
        return result.toString ();
    }
}
```

Listing 13.12 - An example implementation of `RowHandler<ROW>`

13.4.2.1 Using a `RowHandler<ROW>` implementation to set a generated key in a `pureQuery` bean

If your method executes an SQL `INSERT` statement into a table that has generated keys, `pureQuery` can update `pureQuery` beans with the generated values. To use this
functionality, you must specify your parameters using only one pureQuery bean. In that
bean, annotate one or more properties that you want pureQuery to update with the
com.ibm.pdq.annotation.GeneratedKey annotation. If you want to change the way
that pureQuery updates the bean, you can specify an implementation of
com.ibm.pdq.runtime.handlers.RowHandler<ROW>. pureQuery passes the bean
to the handle method as the object parameter. The method must then update object
and return it, rather than creating a new object to return.

Refer to Section 6.6 for an example of using generated keys with an annotated method.
The example defines a table that has a generated key in Listing 11.20. It also defines an
annotated method in Listing 11.22 that executes an INSERT statement on the table. In the
annotated method, pureQuery updates the EmployeeBean parameter with the generated
key. If a RowHandler<ROW> implementation were specified for that method, then
pureQuery would call its handle method to update the bean, rather than updating it
directly. Listing 13.13 shows an example of a RowHandler<ROW> implementation that
could be used to set a generated key in a bean. Listing 13.14 shows how the handler
could be specified for an annotated method. Listing 13.15 shows an example of the output
of the code in Listing 11.23, when the annotated method is defined as shown in Listing

```java
import hrm.completed.EmployeeBean;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.text.NumberFormat;
import com.ibm.pdq.runtime.handlers.RowHandler;

public class SimpleInsertEmployeeRowHandler implements RowHandler<EmployeeBean>
{
    private final NumberFormat empnoFormat;

    public SimpleInsertEmployeeRowHandler ()
    {
        empnoFormat = NumberFormat.getIntegerInstance ();
        empnoFormat.setMinimumIntegerDigits (6);
        empnoFormat.setGroupingUsed (false);
    }

    public EmployeeBean handle (ResultSet resultSet, EmployeeBean employeeBean) throws SQLException
    {
        // Format the new employee number and set it in the bean
        String employeeNumber = empnoFormat.format (resultSet.getInt
```
Getting Started with pureQuery

("EMPNO")
    employeeBean.setEmployeeNumber (employeeNumber);
    return employeeBean;
}
}

Listing 13.13 - A sample RowHandler<ROW> implementation for setting a generated key

@Handler(rowHandler=SimpleInsertEmployeeRowHandler.class)
@Update(sql="INSERT INTO EMPLOYEE_GEN_KEY (FIRSTNME, LASTNAME) VALUES (:firstName, :lastName)")
int insertEmployee(EmployeeBean employee);

Listing 13.14 - Specifying a RowHandler<ROW> implementation for an annotated method that executes an INSERT statement and produces a generated key

Update count: 1
New employee number: 000010

Listing 13.15 - Sample output of the code in Listing 11.23 when executing the annotated method in Listing 13.14

13.4.3 Writing a CallHandlerWithParameters<CAL> implementation

The CallHandlerWithParameters<CAL> interface has the method handleCall, shown in Listing 13.16.

CAL handleCall(CallableStatement cstmt, Object... parameters) throws SQLException;

Listing 13.16 - The handle method in the CallHandlerWithParameters<CAL> interface

The parameters to the handleCall method are the CallableStatement object that pureQuery uses to execute the SQL stored procedure call, and the parameters that were passed to the annotated or inline method (without any handlers that were passed to the method). Your implementation of the handle method must use cstmt to create and return the object that you want your annotated or inline method to return. Do not execute cstmt in the handleCall method (that is, do not call any of the cstmt.executeXXX methods); pureQuery will have already done so when it calls the method.

If any of the parameters to the annotated or inline method are mutable objects that represent OUT or INOUT parameters in the SQL stored procedure call, the handleCall method can update those objects to reflect the new values of the OUT and INOUT parameters. For example, suppose that you have an SQL stored procedure call in which the first parameter is an INOUT parameter that represents the salary of an employee. You pass an EmployeeBean object as the only method parameter, and it provides the initial value of the salary. After the stored procedure executes, the salary might be changed, and the stored procedure would update the value of the parameter. You could put the code in
Listing 13.17 into your `handleCall` method to update the `EmployeeBean` object with the new value.

```java
EmployeeBean bean = (EmployeeBean) parameters[0];
bean.setSalary (cstmt.getDouble("SALARY"));
```

**Listing 13.17 - An example of code in a `handleCall` method that updates a bean**

To give an example of a simple `CallHandlerWithParameters<CAL>` implementation, imagine that you wanted an annotated or inline method to return a `String` describing the query results. Spaces should separate the values of the columns, line breaks should separate the rows, and lines that say "Query Result:" should separate the query results. This would probably not be a very useful way to return the query results, but we are using this just as a simple example. Listing 13.18 shows how you could implement such a handler.

```java
import java.sql.CallableStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import com.ibm.pdq.runtime.handlers.CallHandlerWithParameters;

public class SimpleStringCallHandlerWithParameters implements CallHandlerWithParameters<String> {
    public String handleCall (CallableStatement cstmt, Object... parameters) throws SQLException {
        StringBuilder results = new StringBuilder ();
        ResultSet resultSet;
        while (null != (resultSet = cstmt.getResultSet ())) {
            results.append ("Query Result:
            int columns = resultSet.getMetaData ().getColumnCount ();
            while (resultSet.next ()) {
                for (int i = 0; i < columns; i++) {
                    results.append (resultSet.getString (i) + " ");
                }
                results.append ("\n");
            }
        }
        return results.toString ();
    }
}
```

**Listing 13.18 - An example implementation of `CallHandlerWithParameters<CAL>`**
13.4.4 How to write a ParameterHandler implementation

The ParameterHandler interface has the method handleParameters, shown in Listing 13.19.

```java
void handleParameters(PreparedStatement stmt, Object... parameters) throws SQLException;
```

Listing 13.19 - The handleParameters method of the ParameterHandler interface

The parameters to the handleParameters method are the PreparedStatement object that pureQuery will use to execute the SQL statement, and the parameters to the annotated or inline method. (If handlers passed as parameters to the annotated or inline method, pureQuery will not pass them to handleParameters.) Your implementation of handleParameters must use the method parameters to set the values of the parameters in your SQL statement. Additionally, if the SQL statement is a stored procedure call that has OUT or INOUT parameters, your method must register those parameters.

13.4.5 In practice: Write and specify a ParameterHandler implementation

Let's now write a ParameterHandler implementation and specify it in our application. We will specify it in the method style version of our application that is located in the hrm.inProgress.pdqMethodStyleWithHandlers package. We will use it for the selectEmployee method, which gets detailed information about a single employee for the "Employee Information" screen. Listing 13.20 shows this method.

```java
@Select(sql = "SELECT * FROM EMPLOYEE WHERE EMPNO = ?")
EmployeeBean selectEmployee (String employeeNumber);
```

Listing 13.20 - The method for which we are writing a ParameterHandler implementation

There is no need to write a ParameterHandler implementation for this method, since the default behavior by pureQuery is what we want. However, we are writing an implementation to demonstrate how to use a ParameterHandler implementation.

The name of the ParameterHandler implementation that we are writing SimpleSelectEmployeeParameterHandler, and it is located in the hrm.inProgress package. Of course, it implements the ParameterHandler interface. Locate this class definition in the GettingStartedWithPureQuery project.

Inside the handleParameters method, we have Object... parameters, which is the parameters to the annotated method. The selectEmployee method has only one parameter: a String that contains the employee number. In handleParameters, cast item 0 of parameters as a String and store it in a variable named employeeNumber. Then, use employeeNumber to set the value of the first parameter in the SQL statement, which is a parameter marker for the value of the EMPNO column. Also, add a line that prints a message indicating that our application is using the handler. Listing 13.21 shows the completed SimpleSelectEmployeeParameterHandler.
package hrm.inProgress;

import java.sql.PreparedStatement;
import java.sql.SQLException;

import com.ibm.pdq.runtime.handlers.ParameterHandler;

public class SimpleSelectEmployeeParameterHandler implements ParameterHandler {
    public void handleParameters (PreparedStatement stmt, Object... parameters)
        throws SQLException {
        System.out.println ("Using " +
            SimpleSelectEmployeeParameterHandler.class.getCanonicalName () + 
            "!");
        String employeeNumber = (String) parameters[0];
        stmt.setString (1, employeeNumber);
    }
}

Listing 13.21 - The implementation of SimpleSelectEmployeeParameterHandler

Now, we need to specify SimpleSelectEmployeeParameterHandler for the selectEmployee method in
hrm.inProgress.pdqMethodStyleWithHandlers.HumanResourcesData.
Because the handler has an implicit public no-argument constructor, we can use the
@Handler annotation to specify the handler. Add the annotation to your method, providing
the class of the handler.

@Select(sql = "SELECT * FROM EMPLOYEE WHERE EMPNO = ?")
@Handler(parameterHandler = SimpleSelectEmployeeParameterHandler.class)
EmployeeBean selectEmployee (String employeeNumber);

Listing 13.22 - The selectEmployee method with
SimpleSelectEmployeeParameterHandler

Build the project to regenerate the implementation class and test the application. Look for
the message that SimpleSelectEmployeeParameterHandler prints.

13.5 Summary

pureQuery provides handlers through which you can customize how pureQuery sets the
values of SQL parameters and how it returns query results. Implementations of the
com.ibm.pdq.runtime.handlers.ParameterHandler interface customize how the
values of parameters are set. Implementations of the
com.ibm.pdq.runtime.handlers.ResultHandler<RES> interface customize the
object contains all of the rows. Implementations of the
`com.ibm.pdq.runtime.handlers.RowHandler<ROW>` interface customize the object contains for each row. Implementations of the
`com.ibm.pdq.runtime.handlers.CallHandlerWithParameters<CAL>` interface customize the results of stored procedure calls.

13.6 Review questions

1. If none of the ways in which pureQuery can set the values of SQL parameters meet the needs of your application, what can you do?

2. For an annotated or inline method that executes a stored procedure call, which of the following actions must the `handleParameters` method of a `com.ibm.pdq.runtime.handlers.ParameterHandler` implementation do with the `java.sql.PreparedStatement` object?
   - A. Set the values of IN and INOUT parameters
   - B. Call the `close()` method
   - C. Call one of the `executeXXX()` methods to run the SQL statement
   - D. Register OUT and INOUT parameters

3. If none of the ways in which pureQuery can return the query result of an SQL `SELECT` statement meet the needs of your application, what can you do?

4. What does the type parameter `RES` indicate for the `com.ibm.pdq.runtime.handlers.ResultHandler<RES>` interface?

5. For an implementation of `com.ibm.pdq.runtime.handlers.RowHandler<ROW>`, what should the `handle` method do with the `java.sql.ResultSet` parameter?
   - A. Call the `next()` method
   - B. Use `getXXX` methods to put the contents of the current row in the object that the method returns
   - C. Use the `absolute(int)` method to move to the current row
   - D. Call the `close()` method

6. What is the purpose of the `ROW` object parameter of the `handle` method of the `com.ibm.pdq.runtime.handlers.RowHandler<ROW>` interface?

7. If none of the ways in which pureQuery can return the results of a stored procedure call meet the needs of your application, what can you do?

8. How do you specify handlers in the inline style?
A. In a `getData` method in the
   `com.ibm.pdq.runtime.factory.DataFactory` class

B. With the `setHandler` method on the `com.ibm.pdq.runtime.Data`
   interface

C. As parameters to `queryXXX` and `call` methods in the
   `com.ibm.pdq.runtime.Data` interface

D. None of the above

9. Assume that you have an annotated method that has three method parameters for
   specifying the values of the SQL parameters. If you want to use a handler for this
   method, and the handler has a no-argument constructor, which of the following
   ways can you use to specify the handler?

   A. As the first parameter to the method (making the current method
      parameters the second, third, and forth parameters)

   B. By specifying the class in the in the
      `com.ibm.pdq.annotation.Handler` annotation

   C. As the last parameter to the method

   D. By casting the interface as an instance of `com.ibm.pdq.runtime.Data`
      and calling the `setHandler` method of the `Data` interface

10. Assume that you have an annotated method that has three method parameters for
    specifying the values of the SQL parameters. If you want to use a handler for this
    method, and the handler does not have a no-argument constructor, which of the
    following ways can you use to specify the handler?

    A. As the first parameter to the method (making the current method
       parameters the second, third, and forth parameters)

    B. By specifying the class in the in the
       `com.ibm.pdq.annotation.Handler` annotation

    C. As the last parameter to the method

    D. By casting the interface as an instance of `com.ibm.pdq.runtime.Data`
       and calling the `setHandler` method of the `Data` interface

13.7 Exercise

Write an implementation of
   `com.ibm.pdq.runtime.handlers.CallHandlerWithParameters` and change the
   inline method that executes the stored procedure `bonus_increase` in
   `hrm.inProgress.pdqInlineStyleWithHandlers.HumanResourcesManager` to use
   it. One solution is provided in the `hrm.completed.pdqCompleted` package.
PART V – BEST PRACTICES AND INTEGRATION
Chapter 14 - Best Practices

In the previous chapters of this book, we have introduced pureQuery and described in
detail its major components: API, runtime and tools. In this chapter we will describe best
practices for each one of those components.

14.1 Best Practices: The big picture

pureQuery is a platform designed to deliver increased productivity to developers, and
improved performance, maintainability, monitoring and security for applications. In addition
to the improvements that your application will get by using pureQuery, there are still some
best practices that help you get the most out of pureQuery in certain scenarios. In this
chapter, we will go over some of those best practices and will identify in which scenario we
recommend their usage.

Previously to the writing of this book, the pureQuery team has also published an article on
some best practices, titled "Write high performance Java data access applications, Part 3:
Data Studio pureQuery API best practices". This article was part of a series on developing
high performance Java data access applications and the list of best practices covered
focused on the pureQuery API. In this chapter, we will provide some additional best
practices that focus not only on the API, but also on the tools and runtime components.

14.2 Choose your style: inline vs annotated method style

In Chapters 12 and 13, we described the Inline and Annotated method programming styles
for the pureQuery API. Both styles have their own advantages, so here are some
guidelines on which style to pick when developing your pureQuery application:

Use pureQuery annotated style if you:

- Want to use static SQL at runtime
- Prefer to have your SQL separated from your business logic
- Want a simple data access layer code generated by the Optim Development
  Studio tooling
- Like to use XML files to define your data access layer
Have predefined queries

Use pureQuery inline style if you:

- Like to have your SQL statements inline in your Java code, just like regular JDBC programming
- Have dynamically generated queries

As a rule of thumb, we recommend using the annotated method style. Due to its flexibility and isolation from the business logic, the annotated method style simplifies tasks like refactoring, because code is in a single place, and code reutilization, by sharing your data access interfaces between projects.

Keep in mind that you can also mix both programming styles in your application if your requirements cannot be met by a single style.

### 14.3 Perform Joins with Table collation

In addition to the basic data access generated by pureQuery tooling for a database table, sometimes you need to run a more complex query that joins two tables, and return information from both tables as Java beans.

Table collation in pureQuery lets you reuse the generated beans in these types of queries, returning multiple beans for each result row.

Consider the SQL statement in Listing 14.1, which shows a query that joins data from the tables PROJECT and DEPARTMENT:

```
SELECT PROJECT.PROJNO, DEPARTMENT.DEPTNO
FROM PROJECT, DEPARTMENT
WHERE PROJECT.DEPTNO = DEPARTMENT.DEPTNO
```

**Listing 14.1 - Selecting data from the tables PROJECT and DEPARTMENT**

By using pureQuery’s table collation feature, you can specify that the result of this query should be represented in the business layer using the `Project` and `Department` pureQuery beans, which you can easily generate using tooling.

In order to hold these beans in a result set, you need to define a generic class that can hold the two beans and be used as the pureQuery API method return type:

```java
public class ProjectAndDepartment<P, D> extends LinkedList {}
```

**Listing 14.2 - Generic class to hold a pair of beans**

You can then use this class as the type for the Iterator or List returned by your data access API:
@Select(sql="SELECT PROJECT.PROJNO, DEPARTMENT.DEPTNO " +
              "FROM PROJECT, DEPARTMENT " +
              "WHERE PROJECT.DEPTNO = " +
              "DEPARTMENT.DEPTNO")
List<ProjectAndDepartment<Project, Department>> getProjectsDepartment();

Listing 14.3 – Returning a types list of beans

Using table collation to aggregate results form multiple tables leverages the table-to-bean mapping provided by pureQuery and prevents you from having to design complex beans that would store the result of the join SQL statement.

14.4 Working with Exception categorization

When pureQuery executes JDBC calls, it encapsulates any generated exception into unchecked exceptions, so that you don’t have to enclose all your pureQuery method calls in try/catch blocks. By using this process, pureQuery is also able to provide you with extended metadata about the generated exception when it encapsulates the base JDBC or SQL exception in a pureQuery DataRuntimeException. This metadata provides you with the version of pureQuery you are using, the associated SQL state and error code of the generated exception and also exception categorization information. pureQuery is aggregating errors and their causes into categories that let you better handle and recover from the error situation. Some example categories used by pureQuery are AUTHORIZATION_ERROR, CONSTRAINT_VIOLATION, DUPLICATE_ROW_VIOLATION, SYNTAX_ERROR, amongst others. For a complete list of all possible error types, please refer to the pureQuery documentation.

When developing your application, it is recommended that you use these error types as a strategy for error handling and recovery, as it eliminates the need to handle individual error types for each pureQuery method call.

14.5 Customizing the code generation templates

pureQuery code generation tooling is based on JET (Java Emitter Templates), a generic template engine built into Eclipse that can be used to express code you want to generate.

When you use the tooling to generate pureQuery code, there are templates for all the generated Java class files, as the bean representing the database table, the inline and annotated method style sample classes and even the JUnit test cases.

ODS allows you to override the existing templates, to write customized code that you wish to have added automatically to each one of the generated classes. This code can be as simple as a copyright statement, or more complex like adding some setup code to the generated test cases. See the official pureQuery documentation for more information about creating Eclipse Modeling Framework Technologies (EMFT) Java Emitter Templates (JET) to generate customized code.
This section describes how to use a new JET templates that override the default templates provided by ODS. In this section, we will show you how to override the JET template that ODS uses to generate the Java bean code for a table so that it includes a customized copyright notice at the beginning of each file.

Listing 14.3 - Overriding the GenTableBean.jet template

The code snipped in Listing 14.3 shows, in bold text, the changes made to the default GenTableBean.jet JET template that is used to generate the pureQuery bean for a table.

When generating pureQuery beans using this template, your bean class will automatically contain the copyright notice, as shown in Listing 14.3. Even though this is a very simple scenario, it highlights how easy it is to extend the default code generation with your own snippets.
Creating your own customized JET templates for pureQuery code generation is a best practice that improves your productivity and the maintainability of your code, as mandatory code snippets can be easily maintained through templates instead of adding them to each individual file.

14.6 Recapture the SQL in your application

When using the pureQuery Client Optimizer to convert your application's data access into Static SQL or tune the existing SQL statements, it is important that you are aware of the dynamics of your application. While some applications have a fixed set of executed SQL statements and a static workload, other applications are more dynamic. When working with applications that dynamically generate SQL statement or applications that have varying workloads, it is important that you frequently run your application in capture mode. This will increase your odds of capturing all the SQL executed by the application, and it will allow you to tune the SQL statements based on the workload.

It is also critical that you run your application in capture mode after you make changes to it. When re-executing the capture process after making changes to it, it is important that you
adjust the workload to execute any SQL statements you added during the refactoring of the application.

14.7 Custom handlers and annotated method style

pureQuery annotated method style lets you specify custom handlers using two different methods: by using the `@Handler` annotation; and by passing them as arguments to the annotated method signature.

When using stateless handlers, it is a best practice to use just a single instance of each handler, instead of passing a separate instance to each method call. When you use the `@Handler` annotation to define the custom handler for your method, pureQuery guarantees that only a single instance is created for each specified handler class. Therefore, it is recommended that you use the `@Handler` annotation when possible, or make sure your application reuses the handler between calls. This best practice ensures reduced memory utilization and possible performance improvements as only a single class instance is created and managed.

14.8 Summary

In this chapter, we described best practices that you should use to get the most out of pureQuery when developing new applications or optimizing existing applications. Some of the listed practices provide you with increased productivity, others make your application run faster and others let you create applications that are easier to maintain.

For an additional list of best practices for pureQuery application development, please refer to the IBM developerWorks article “Write high performance Java data access applications. Part 3: Data Studio pureQuery API best practices”. 
Chapter 15 - pureQuery and pureXML

In this chapter we will describe how to integrate data stored using DB2 pureXML into pureQuery applications.

15.1 pureQuery and pureXML: The big picture

Both pureXML and pureQuery are revolutionary technologies in the databases field. pureXML introduced a native way to store, manage and query XML documents into DB2, delivering a top quality hybrid database server.

pureQuery is revolutionizing the data access field, by providing improved performance, optimization of existing applications, monitoring, manageability and advanced security control for Java applications that access data stored in a database.

When it comes to integrate pureXML data into a pureQuery application, there are multiple plug-in points and/or schemes that can be used. In this book we will focus on the following three approaches:

- **Give control to the SQL layer**: with this approach, you use the SQL layer to transform data between XML and relational formats, exposing the relational representation to the application and leveraging pureQuery’s relational mapping support.

- **Give control to the data access API**: with this approach, you can use one of the multiple existing XML-Java mapping frameworks to map XML data into Java objects and vice-versa.

- **Give control to the application layer**: with this approach, you implement your own mapping logic/framework, tightly integrated into the business logic, providing expanded versatility when data is shared by multiple applications.

15.2 Give control to the SQL layer

In this approach, the facilities in the SQL layer are used to map the data from XML into relational format and vice-versa. In the examples we will show, SQL/XML will be used to
create a query that exposes the XML data in relational format and XQuery transform expressions are used to persist updates back into the XML data stored in the database.

Figure 15.1 - Giving control to the SQL layer

Figure 15.1 shows a high level view of this approach. When reading data from the database, an SQL/XML statement containing an XMLTABLE() function is used to expose the XML data in relational format. The relational data rows are then interpreted by the pureQuery API, which uses them to populate the Java beans in the application layer. The SQL/XML statement used to do this mapping can be seen in Listing 16.1:

```java
class Example {
    public static void updateEmployee(Data data, Employee emp){
        String sql = "update employee set doc = xmlquery(" +
            "'copy $new := $DOC " +
            "modify (do replace value of $new/employee/name/first with 
            $firstname," +
            "do replace value of $new/employee/name/last with $lastname," +
            "do replace value of $new/employee/office with $office)" +
            "return $new' " +
            "passing cast(:firstname as varchar(20)) as "firstname", " +
            "cast(:lastname as varchar(20)) as "lastname", " +
            "cast(:office as INTEGER) as "office" )" +
            " where id = :id";
        data.update(sql, emp);
    }
}
```

Listing 15.1 – SQL/XML Mapping

You can use SQL/XML statements in either style of pureQuery programming: inline style or annotated method style. Please refer to Chapters 5 and 6 to learn more about these two different styles of programming with the pureQuery API.

When persisting changes into the database, the pureQuery API fetches the updated values from the Java beans and passes them into the database, where an XQuery Transform
expression will be used to update the correct nodes in the original XML document. *Listing 15.2* shows an example of an XQuery Transform expression that can be used to update the *Employee* information in the database:

```java
public static void updateEmployee(Data data, Employee emp)
{
    String sql = "update employee set doc = xmlquery(" +
        "'copy $new := $DOC " +
        "modify (do replace value of $new/employee/name/first with $firstname,
    "do replace value of $new/employee/name/last with $lastname," +
    "do replace value of $new/employee/office with $office)" +
    "return $new' " +
    "passing cast(:firstname as varchar(20)) as \"firstname\", " +
    "cast(:lastname as varchar(20)) as \"lastname\"," +
    "cast(:office as INTEGER) as \"office\"" +
    ") where id = :id"
    data.update(sql, emp);
}
```

*Listing 15.2 – Using an XQuery Transform expression*

Using this approach allows the application developers to be oblivious of schema evolution in the application data, as the mapping is done in the database, hiding these changes from the application.

### 15.3 Give control to the data access API

In this approach, we will show how XML data can be mapped into Java objects and vice-versa using the data access API, instead of doing this mapping at the database level like in the previous approach.

The pureQuery API provides several plug-in mechanisms, including custom result set and custom data row handlers. These custom handlers provide you with means to override default mapping strategies from relational data into Java beans. The *ResultHandler* lets you control the mapping strategy for the entire result set. This is very useful for aggregation queries and one-to-many relationships, when several consecutive result sets may contain information about the same business object. The *RowHandler* lets you define how individual relational result set rows are mapped into a Java bean. Unlike the *ResultHandler*, the *RowHandler* only has access to the current row in the result set and not the entire result set.

By taking advantage of these plug-in mechanisms, you can use one of the several existing Java-XML mapping frameworks to map from XML documents stored as pureXML data into your business logic Java beans, as shown in Figure 15.2.
Examples of Java-XML mapping frameworks are: Castor, JAXB, JIBX, XMLBeans and the integrated Java-XML mapping support in Java 2 SE v6. In order to avoid extra setup steps, the examples in this chapter use the mapping support integrated into Java2 SE v6, which is built on top of JAXB.

In order to map from XML to Java in the data access API, you need to implement a custom `RowHandler` that uses JAXB as shown in Listing 15.3:

```java
public class JAXBHandler implements RowHandler {

    public Employee handle(ResultSet rs, Object arg1) throws SQLException {
        Employee emp = new Employee();
        JAXBContext jContext;
        try {
            jContext = JAXBContext.newInstance("pdq.purexml.approach2");
            Unmarshaller unmarshaller = jContext.createUnmarshaller();
            emp = (Employee) unmarshaller.unmarshal(rs.getBinaryStream(2));
        } catch (JAXBException e) {
            e.printStackTrace();
        }
        emp.setId(rs.getString(1));
        return emp;
    }
}
```

Listing 15.3 – Mapping from XML to Java using JAXB
pureQuery delegates each row processing to this custom row handler. The handler is responsible for fetching the data from the result set row and populating the Java bean.

By using a custom handler, the data access method call to fetch the data from the database is simple, as shown in Listing 15.4:

```java
public static List<Employee> getEmployees(Data data){
    return data.queryList("select * from employee", new JAXBHandler());
}
```

Listing 15.4 – Simple pureQuery method call to retrieve XML data

Note that the handler is passed as a parameter to the pureQuery API call. This tells pureQuery to use this handler for the data mapping.

By using this middle man approach for the XML<->Java mapping, you will create a solution where XML data flows between the database and the data access API, while relational data flows between the data access API and the business logic. By using a mapping framework to do the XML<->Java transformation, any schema changes will only affect this mapping. Some of the listed frameworks provide tooling to generate the mapping code, making it easy to adopt schema changes.

15.4 Give control to the application layer

The third approach that you can take when handling XML data in your pureQuery application is by giving the application layer all the control over the XML<->Java mapping process. XML flows from the database all the way to the application logic and back into the database. This requires a much simpler data access API that need not be aware of the data structure and changes in schema.

Figure 15.3 - Giving control to the application layer

The mapping at the application layer can be achieved by integrating it into the Java beans themselves. One easy way is to add this functionality to the getter and setter methods called by pureQuery when populating the Java beans with the information from the database.
If the XML data will be stored in the `doc` field in the Java bean, then you should create your own `setDoc()` setter method that extracts the information from the XML document at population time:

```java
public void setDoc(String doc) {
    this.doc = doc;
    populateVariables();
}
```

```java
private void populateVariables()
{
    try {
        DocumentBuilder
        builder = DocumentBuilderFactory.newInstance().newDocumentBuilder();
        Document document = builder.parse(new InputSource(new
        StringReader(this.doc)));
        XPath xpath = XPathFactory.newInstance().newXPath();
        String firstname_expression = "/employee/name/first";
        String lastname_expression = "/employee/name/last";
        String office_expression = "/employee/office";
        String fname = (String)xpath.evaluate(
            firstname_expression, document, XPathConstants.STRING);
        String lname = (String)xpath.evaluate(
            lastname_expression, document, XPathConstants.STRING);
        int office = (new Integer((String)xpath.evaluate(
            office_expression, document,
            XPathConstants.STRING)).intValue();
        setFirstname(fname);
        setLastname(lname);
        setOffice(office);
    } catch (Exception e) {
        //handle exception
    }
}
```

**Listing 15.5 – Allowing the application to control XML to Java mapping**

The method `populateVariables()` will traverse the XML document and extract the required information from it, populating other field in the bean, including `firstname`, `lastname` and `office`.

In a scenario where multiple applications have different usage requirements over the same data set, this approach fits nicely. These applications can share the same data access API, while taking control over extracting the required information from the XML documents fetched from the database.
15.5 Summary

This chapter proposed three approaches for handling pureXML data in pureQuery applications. These approaches highlight the usage of different layers of an application to do the required step of XML<->Java mapping. There are certainly other alternative approaches, and the ones listed can be combined in order to provide customized solutions.

Each one of the approaches has its advantages and disadvantages that you should consider when developing your pureQuery application that uses pureXML data:

- **Giving control to the SQL layer**
  - **Pros**: it takes advantage of pureQuery relational mapping facilities. Reduced I/O, as only required info flows from database to application.
  - **Cons**: requirement changes manifest in updates to both the SQL/XML statement and the bean. SQL/XML statement can become too complex depending on the amount of information fetched from XML data.

- **Giving control to the data access API**
  - **Pros**: Application and SQL layer are oblivious to schema changes. Tooling streamlines adoption of new schema or change in requirements.
  - **Cons**: beans hierarchy can become overly complex when mapping from large XSD schemas. Increased I/O, as the whole document is fetched from database into data access layer.

- **Giving control to the application layer**
  - **Pros**: schema and requirement changes are oblivious to data access API and SQL layer. Flexibility when multiple requirements coexist.
  - **Cons**: mapping complex to implement. Increased I/O, as full document contents are sent from database to application and back.

Chapter 16 - pureQuery and JPA

In this chapter we discuss the integration points between pureQuery and JPA, the Java Persistence API.

16.1 pureQuery and JPA: The big picture

JPA was introduced in the Java language in its release Java EE 5. The creation of this new persistence API aimed at simplifying the development of Java applications that use data persistence.

Due to its late arrival on the persistence framework scene, JPA developers were able to draw upon the best ideas from the existing persistence technologies and today JPA has become almost a de facto standard for data persistence in the enterprise world, with its specification being implemented by multiple providers, both commercial and open-source.

IBM Websphere Application Server V7 delivers an enhanced version of its JPA implementation that supports pureQuery and, therefore, allows the users to leverage static SQL support in DB2 from their Java application. This integrated approach lets you leverage the enhanced security and optimized performance provided by pureQuery in your JPA applications.

In this chapter, we will describe how to take advantage of the performance and security of static SQL, provided by the pureQuery runtime, while leveraging the complete ORM – Object Relational Mapping – capabilities of JPA.

16.2 Static execution of JPA with pureQuery

Websphere Application Server V7 provides you with the option to override JDBC with pureQuery for the application data access.

The flow of a JPA application using pureQuery for its data access is shown in Figure 16.1.
As Figure 16.1 highlights, the process of enabling a JPA application to run on top of pureQuery is comprised of two phases:

- **Generation time**: the utility `wsdb2gen.bat` is used to capture all the SQL statements and generate DB2 packages that contain the statements
- **Execution time**: the JPA calls are redirected to the pureQuery runtime that, in turn, redirects them to the DB2 packages that contain the preprocessed SQL statements.

In the next sections we will describe both phases in detail.

### 16.3 Generating DB2 packages for JPA applications

The utility used to collect the SQL statements from JPA applications and generate DB2 packages is called Static Generator. This utility can be invoked by executing the command `wsdb2gen.bat` that exists in the folder `%WAS_HOME%\bin`, where `%WAS_HOME%` stands for your Websphere Application Server V7 installation folder.

This utility takes as input parameter, amongst other options, the name of a persistence unit. The utility then generates an output file containing SQL statements that represent all the entity operations, including persist, remove, update and find. In addition to these
statements, SQL statements are also generated to reflect the JPA named queries that exist in the application. SQL statements that are dynamically generated by the application cannot be captured by this utility. For more information on how to capture and optimize dynamic queries, refer to Chapter 6 - The Client Optimizer: pureQuery for Existing JDBC Applications.

In this chapter, we will use the sample application provided in the IBM developerWorks article “Integrating JPA and pureQuery: Leveraging DB2 static execution for the Java Persistence API”. In the Resources section of this book you can find a download link for the JPA_EE_SAMPLE.ear application file.

The provided sample application contains a persistence unit, of name JPASample and implements functionality that lets the user create, update and delete customer entities, as well as retrieve information regarding orders.

16.4 Generating SQL statements

Before we describe the parameters of the Static Generator utility, please note that for the utility to function properly, the following items must be part of the DataProvider classpath in your WAS environment:

- pureQuery runtime .jar files
- DB2 driver .jar files
- the entity class .jar file or folder where it is located
- the location of the persistence.xml file

After you have correctly set up the system classpath, the Static Generator utility can be invoked by executing the command wsdb2gen.bat, as shown in Listing 16.1.
**Syntax:** WSDB2GEN -pu <puName> [Flags]

Where

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>puName</td>
<td>Persistence-unit name defined in the persistence.xml</td>
</tr>
<tr>
<td>-collection</td>
<td>Collection id assigned to package names. Default is NULLID.</td>
</tr>
<tr>
<td>-url</td>
<td>The URL is used to validate generated sql. If url option is specified, it overrides the url value in the persistence.xml. If the url is type 4, userid and password are required for connection. If the url is type 2, userid and password are not required.</td>
</tr>
<tr>
<td>-user</td>
<td>The userid to connect to target database. If user option is specified, it overrides the user value in the persistence.xml.</td>
</tr>
<tr>
<td>-pw</td>
<td>The password to connect to target database. If password option is specified, it overrides the value in the persistence.xml.</td>
</tr>
<tr>
<td>-package</td>
<td>A name of 7 or fewer characters for the package name. If specified, all SQL for the application will be generated into this single package. If package option is not specified, then multiple packages will be created; one for each entity class. If the first 7 characters are not unique, the package name will be changed to ensure uniqueness.</td>
</tr>
</tbody>
</table>

**Example Usage:**

```bash
C:\was70\bin>set classpath=C:\db2jcc.jar;C:\pdq.jar;C:\pdqmgmt.jar;%classpath%
C:\was70\bin>set classpath=C:\test1\bin;%classpath%
C:\was70\bin>wsdb2gen -pu JPASample -url jdbc:db2://localhost:50000/demodb -user db2admin -pw passw0rd
```
Listing 16.1 - Static Generator command line execution

The last line in Listing 16.1 shows how to run the Static Generator utility to generate the SQL statements for the JPASample persistence unit:

```plaintext
wsdb2gen -pu JPASample -url jdbc:db2://localhost:50000/demodb -user db2admin -pw passw0rd
```

Upon execution, this command will generate a file consisting of the SQL statements that reflect the data operations implemented by the persistence unit JPASample.

The generated file will have extension .pdqxml and the name will reflect that of the persistence unit. In the example described in this section, the file created has the name JPASample.pdqxml. The .pdqxml file format is known as the pureQuery capture file format, already described in Chapter 6. This file contains SQL statements that will later be converted to DB2 packages.

The Static Generator utility created the bind file in the same location of the persistence.xml file.

16.4.1 Creating and binding the packages

After all the SQL statements have been generated, the next step is to create and bind the DB2 packages that will contain these same SQL statements.

This step can be achieved using two different methods:

- using the command line based tool pureQuery Static Binder
- using the Websphere Application Server V7 administration console

Since the pureQuery Static Binder has already been covered on Chapter 7, we will describe how to generated and bind the packages using the WAS administration console.

Starting with version 7, the WAS admin console provides an option that lets users load and bind a pureQuery bind file. This option is available on the application’s configuration page, as shown in Figure 16.2 and it provides an intuitive wizard that guides the user through the binding process.
Once the packages have been bound to the database, the next step is to make the JPA application run on top of the pureQuery runtime. We will discuss that in the next section.

16.5 Running a JPA application using pureQuery

One of the advantages of running a JPA application on top of pureQuery runtime is the ability to use the enhanced security control of static SQL in DB2. As we explain in Appendix C, static SQL allows specifying individual access privileges at the package level.

The first step to make our JPA application run on top of pureQuery is to set the correct access privileges to the packages generated and bound in the previous section. In order to allow a user to run the packages we created, we need to grant the user execute privilege on the packages. That can be done by specifying the pureQuery StaticBinder option – grant.
If we wanted to give user user1 execute privileges on the generated packages, the pureQuery runtime option to use would be:

```
-grant grantees(user1)
```

In order to simplify our scenario, we will simply grant execute privileges to everyone, so that all users (and the applications they represent) connecting to our database can take advantage of the enhanced performance of static SQL. We do that by granting the privilege to the user `public`, as shown in Figure 16.3.

![Figure 16.3 - Granting execute privileges on the bound packages](image)

Please note that granting execute privileges to the user `public` is generally not a good practice. A tighter and granular security control mechanism is one of the biggest advantages of static SQL execution mode when compared with dynamic SQL execution mode and it should always be taken into consideration when designing and implementing data access applications.
16.5.1 Running the application in static mode

Since we created and bound the packages using the WAS administration console, our application has been configured to use pureQuery instead of JDBC as the data access mechanism.

We can run the application the same way we would do before converting the data access to pureQuery, either from the WAS administration console or by simply typing the application URL in a browser.

The URL for our sample application is `http://localhost:9080/JPA_W_EJB_WEB/`. Enter this address in your browser and the JPA sample application will launch, as seen in Figure 16.4.

![Figure 16.4 - JPA application running in static SQL mode](image)

16.6 Summary

This chapter discussed the integration of JPA and the pureQuery runtime, which allows for JPA applications that run on WAS V7 to take advantage of static SQL execution in DB2.

We described the process of using the Static Generator utility to generate SQL statements in bind files that were later used by the pureQuery Static Binder in order to create and bind the DB2 packages into the database. We also described the process of authorizing users to run the application in static SQL mode.

Using the Static Generator is a quick way to get started with pureQuery execution for JPA applications. If you have an application that generates dynamic SQL statements, then you can leverage the purequery Client Optimizer to capture those statements and improve the performance, security and maintainability of your application.
The approach described in this chapter is complementary to the Client Optimizer approach and can be used simultaneously to maximize the performance, maintainability and security control of your JPA application.
Appendix A  - Solutions to the review questions

Chapter 1

1. The four facets that comprise the pureQuery platform include: 1) APIs 2) runtime 3) application development environment 4) monitoring services.

2. False. pureQuery client monitoring capabilities can help developers find potential hot spots in their SQL so they can address problems in the development cycle or alert their DBA to the problem.

3. False. Although pureQuery is recommended as an alternative to other frameworks for high performance Java applications, it can also be used to optimize applications for which those other frameworks are required. Thus, it can be considered an enhancer rather than a competitor.

4. Although most development benefits are available for Java applications, .NET applications can also take advantage of pureQuery for static SQL execution and the performance and security benefits that come with that.

5. Client optimization (so called because these optimizations are occurring on the database client side rather than using database server optimizations such as access path optimizations).

6. B

7. C.

8. C. (For .Net development against DB2, you can use Visual Studio Add-ins.)

9. B.

10. D. True at the time this was written, but support for SQL Server is planned to be added in the future.

Chapter 2

1. Optim Development Studio can be installed using the following options:
   
   A. Web Install with new Installation Manager

   B. Web Install using existing Installation Manager
Chapter 3

5. When adding pureQuery support to your Java project, either explicitly or implicitly, the workbench will add the pureQuery runtime jars physically to your project depending on your set preferences in the pureQuery Runtime Location group under Windows \Preferences \Data Management \SQL Development \pureQuery.

OR

When adding pureQuery Support the user is asked to select a location from where to get the pureQuery JAR files to add to their project. The user can either choose to import the JARs from the ODS install location or choose a folder path where both, the pdq.jar and the pdqmgmt.jar files are located. For the second option, the JARs will not be imported physically to your project; only the project’s classpath will be modified to include such files.

6. It is often desirable to design applications with an offline model, especially when live remote servers may not always be available or may be slow to use at application design time. This feature can be useful for remote zSeries and iSeries servers with large number of objects.

7. A user might want to override SQL statements in an XML file when you want to use the same interface in another application that runs against a different database that may contain columns with a slightly different name (e.g: \texttt{FIRSTNAME} instead of \texttt{FIRSTNAME}).

8. If an annotated method takes a parameter, the test method contains comments to indicate that you must assign a value to such parameters.

Chapter 4

1. A.

2. A.

3. B.

4. B.
5. D.

6. You can use the filter in the SQL Outline to show only the SQL statements that contain a column with privacy information by selecting to see only SQL containing columns marked as private.

7. In order to view all SQL statements issued by your application to the database and not only the Java expressions, you will have to run your application using the Client Optimizer feature from the pureQuery runtime.

8. You must *Enable SQL capturing and binding for JDBC applications* when either adding pureQuery support to your project or from the project’s properties page. When enabling this feature, a `pdq.properties` file will be added to your project where, through a set of properties, you can enable/disable the capturing of statements being executed by your application.

Chapter 5

1. D.
2. D.
3. B.

Chapter 6

1. Client Optimizer
2. Capture
3. B
4. You must always specify the option `-rootPkgName`. The utility uses the value of this option to determine the base package name of the statement sets in the pureQueryXml file. pureQuery uses the base package name to determine the names of the packages in the database when binding and when executing statically.
5. C
6. C
7. A
8. True
9. A, B and C.

Chapter 7

1. B and D.
2. A and B.
3. The StaticBinder utility binds four packages in the database, one at each of four isolation levels. Each package contains all of the SQL statements in the interface.

4. The names of the packages are the root package name of the interface, generally with 1, 2, 3, or 4 appended to indicate the isolation level. The root package name of an interface is specified with the -rootPkgName option to the Generator utility, and it is stored in the identifier field in the implementation class. If the option -rootPkgName is not specified, the Generator utility chooses a name that is based on the name of the interface.

5. The names of the packages are the base package names of the statement sets in the pureQueryXml file, generally with 1, 2, 3, or 4 appended to indicate the isolation level. The base package name of a statement set is the value in the name attribute of its package element. For pureQueryXml files that are processed with the Configure utility, the root package name is specified to the Configure utility with the option -rootPkgName, and the utility uses the root package name to create the base package names of the statement sets.

6. A.
7. D.
8. Specify the name of the file, followed by a colon, followed by the base package name of the statement set. For example, 
   C:\path\captureFile.pdqxml:MYPKGA
9. .ear, .jar, .war, .zip
10. B.
11. True.

Chapter 8
1. Common problems faced by DBAs in a multi-tiered environment include lack of information about performance statistics in the middle tier, and inability to identify the applications executing problematic SQL.
2. A native pureQuery application is one written in pureQuery annotated method style or using pureQuery development tooling that produces similar source metadata, such as a Data Web Service.
3. True. Extended Insight data integrates seamlessly with database-centric monitor data.
4. Before application metadata can be useful for monitoring and problem determination, it must be registered with OPM.
5. Extended Insight provides new statistics from the middle tier between the application layer and the database layer, including database driver and network statistics.

6. C.

7. D.

8. C.

9. D.

10. A.

Chapter 9

1. The advantages of using a persistence framework are to encapsulate the business logic and relational schema from each other, and to allow the application developer to focus on writing business logic instead of persistence management.

2. Object-relational mappers are the part of a persistence framework that maps between object data and relational data, whenever the backend persistence store is a relational database.

3. False. pureQuery is not a persistence framework, but is a platform which can enhance a variety of existing persistence frameworks.

4. Object-relational mapping are often complicated by database normalization and complex class hierarchies.

5. False. pureQuery supports both simple and complex object-relational mapping schemes.

6. A. JDO has no pureQuery specific integrations.

7. D. XML entity mapping files, ResultHandlers, RowHandlers, and SQL Replacement are all pureQuery features that support advanced object-relational mapping design.

8. C.

9. A.

10. D.

Chapter 10

1. Specify an SQL statement that you want to execute as a parameter to a method in the com.ibm.pdq.runtime.Data interface.

3. Specify the objects that contain the values for the SQL parameters as the value of the varargs parameter `Object... parameters` of the inline method. Use pureQuery's parameter markers in the SQL statement to dictate how pureQuery should set the values of the parameters.

4. B. and D.

5. False.

6. D. All of the above

7. You should probably return the rows in a `java.util.Iterator<E>` object because it does not fully materialize the results initially. Rather, it materializes one row at a time.

8. True.


10. B. Specify the cursor attributes as parameters to the `queryXXX` method.

11. Call the `close()` method in the `Data` interface.

12. Some of the benefits of using the inline style are:

   - It is easy to specify the values of SQL parameters to your SQL statements.
   - It is easy to have pureQuery create and return useful objects from your SQL statements.
   - You can very quickly specify SQL statements in your application.

Chapter 11

1. B

2. Specify an SQL statement that you want to execute in the `@Select`, `@Update`, or `@Call` annotation.

3. Specify the objects that contain the values for the SQL parameters as the parameters of the annotated method. Use pureQuery's parameter markers in the SQL statement to dictate how pureQuery should set the values of the parameters.

4. D

5. B

6. C

7. C
8. Each of the implementation classes implements `com.ibm.pdq.runtime.Data`, in addition to the interface of annotated methods. This is helpful because you can use the methods to manage the `java.sql.Connection` object that the instance contains, and to perform transaction management for the SQL statements that the annotated methods execute.

9. Cast the instance as a `com.ibm.pdq.runtime.Data` object and call the `close()` method in the `Data` interface.

10. Some of the benefits of using the annotated-methods style are:
    A. You can speed-up your application development. All you have to do is declare methods that specify the SQL statement, the parameters, and the return type, and the tools take care of the rest of the work.
    B. You can use annotated-methods style code to execute SQL statements statically.
    C. The interfaces containing annotated methods help to isolate the SQL statements from the application code.

11. The user must override the default JET transformation by creating an EMFT JET project, creating the templates and telling ODS to use your own transformations. The many different JET templates used by the pureQuery tooling are located in the `com.ibm.datatools.javatool.transform.codegen` plugin within ODS. Once you have created your EMFT JET Transformation project and your template to override one of the default templates provided, go to the pureQuery Transforms preference page and specify which transformation the workbench must use.

Chapter 12


2. You use the `com.ibm.pdq.annotation.Call` annotation to have an annotated method execute a stored procedure call.

3. A and C

4. B and C

5. False

6. B

7. Assume that the name of the `StoredProcedureResult` object is `spr`. Use this code: `Integer parm4 = (Integer) spr.getOutputParms()[3]`

8. You can call the `close()` method. Also, if you retrieve all of the query results, pureQuery automatically frees the database resources.
9. E
10. com.ibm.pdq.runtime.handlers.CallHandlerWithParameters

Chapter 13

1. Write an implementation of the interface 
   com.ibm.pdq.runtime.handlers.ParameterHandler, and specify it to the 
   inline or annotated method.
2. A and D.
3. Write an implementation of the interface 
   com.ibm.pdq.runtime.handlers.ResultHandler<RES> or of the interface 
   com.ibm.pdq.runtime.handlers.RowHandler<ROW>. Specify the 
   implementation class to the inline or annotated method.
4. The type parameter RES indicates the return type of the handle method.
5. B.
6. com.ibm.pdq.runtime.handlers.RowHandler<ROW> implementations use 
   the ROW object parameter when updating pureQuery beans with the values of 
   generated keys. pureQuery passes the first parameter of the annotated method as 
   object. The handle method must update object with the new values. object 
   must be a pureQuery bean.
7. Write an implementation of the interface 
   com.ibm.pdq.runtime.handlers.CallHandlerWithParameters<CAL> in 
   which the handleCall method creates an object containing the query results.
8. C.
9. B and C.
10. C.
Appendix B - Understanding the Sample Application

There is a sample application called “Human Resources Manager” that is provided with this book. The Java project named `GettingStartedWithPureQuery` contains five different implementations of the application. It is implemented in standard JDBC. It is also implemented twice in the inline style and twice using annotated method, pureQuery data access objects. The different implementations illustrate some of the features available in pureQuery.

Human Resources Manager uses the `SAMPLE` database that is provided by IBM DB2 for Linux, UNIX, and Windows. This database is included with all versions of DB2 LUW, including DB2 Express-C. Specifically, the application uses the tables `EMPLOYEE` and `DEPARTMENT`, and it uses the SQL stored procedure `bonus_increase`.

This book assumes that you are using the `GettingStartedWithPureQuery` project in IBM Optim Development Studio. For information about downloading and installing Optim Development Studio, refer to Chapter 2.

B.1 A tour of the Human Resources Manager application user interface

Human Resources Manager is a very simple utility that a company could use for managing information about its employees. It is designed to illustrate quickly some of the basic features of pureQuery. Therefore, it is relatively small and simple, and it is not intended to be a production-quality application. The application consists of seven screens. You can navigate between the screens by selecting a menu option from a screen.
B.1.1 The “Login” screen

The initial screen is the “Login” screen. It simply asks for the employee number of the user who is logging in. The employee numbers are those contained in the EMPNO column of the EMPLOYEE table. Figure B.1 shows an example of this screen.

```
============= Human Resources Manager =============
Enter your employee number: 000030
```

Figure B.5 - The "Login" screen

B.1.2 The “Main Menu” screen

The next screen is the “Main Menu” screen, shown in Figure B.2. This screen allows the user to choose any of the other screens.

```
============= Main Menu =============
What would you like to do?
  0. Exit Human Resources Manager
  1. Display information about yourself
  2. Change information about yourself
  3. Display your report chain
  4. Display all employees in the company

Please make a selection. Enter the number for your choice.
```

Figure B.6 - The "Main Menu" screen
B.1.3 The “Employee Information” screen

The “Employee Information” screen displays details about an employee. For a given employee, the screen shows more information to that employee and his or her managers than it does to other employees. Figure B.3 shows the information that Sally Kwan and her managers see about herself; Figure B.4 shows the information Sally Kwan sees about someone else.

The “Employee Information” screen displays details about an employee. For a given employee, the screen shows more information to that employee and his or her managers than it does to other employees. Figure B.3 shows the information that Sally Kwan and her managers see about herself; Figure B.4 shows the information Sally Kwan sees about someone else.

<table>
<thead>
<tr>
<th>Employee information for SALLY A. KWAN:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Last Name: KWAN</td>
</tr>
<tr>
<td>First Name: SALLY</td>
</tr>
<tr>
<td>Middle Initial: A</td>
</tr>
<tr>
<td>Department Number: C01</td>
</tr>
<tr>
<td>Job: MANAGER</td>
</tr>
<tr>
<td>Phone Extension: 4738</td>
</tr>
<tr>
<td>Employee Number: 000030</td>
</tr>
<tr>
<td>Education Level: 20</td>
</tr>
<tr>
<td>Salary: $98,250.00</td>
</tr>
<tr>
<td>Bonus: $800.00</td>
</tr>
<tr>
<td>Commission: $3,060.00</td>
</tr>
<tr>
<td>Date of Birth: May 11, 1971</td>
</tr>
<tr>
<td>Hired Date: Apr 5, 2005</td>
</tr>
<tr>
<td>Manager: CHRISTINE I. HAAS, Department A00, PRES</td>
</tr>
</tbody>
</table>

What would you like to do?
0. Return to the previous menu
1. Display the report chain for this employee
2. Change information for this employee

Please make a selection. Enter the number for your choice.

Figure B.7 - The "Employee Information" screen for Sally Kwan, as seen by Sally and her managers
B.1.4 The "Change Employee Information" screen

The "Change Employee Information" screen allows employees to change information about themselves. It also allows managers to change information about the employees that they manage. Figure B.5 shows the way Sally Kwan sees the screen when she changes information about herself; Figure B.4 shows the way she sees the screen when she changes information for an employee that she manages.
Appendix C – Up and Running with DB2 Express-C 317

------------ Change Employee Information ------------

Changing information for: SALLY A. KWAN

Logged in as: SALLY A. KWAN
Relationship to SALLY A. KWAN: Self

Information about: SALLY A. KWAN
1. Last Name: KWAN
2. First Name: SALLY
3. Middle Initial: A
4. Education Level: 20
5. Date of Birth: May 11, 1971

What would you like to do?
0. Return to the previous menu
1-5. Change the information that is listed at the specified number.

Please make a selection. Enter the number for your choice.

Figure B.9 - The "Change Employee Information" screen that Sally Kwan sees about herself

------------ Change Employee Information ------------

Changing information for: HEATHER A. NICHOLLS

Logged in as: SALLY A. KWAN
Relationship to HEATHER A. NICHOLLS: Manager

Information about: HEATHER A. NICHOLLS
1. Job: ANALYST
2. Salary: $68,420.00
3. Bonus: $600.00
4. Commission: $2,274.00

What would you like to do?
0. Return to the previous menu
1-4. Change the information that is listed at the specified number.

Please make a selection. Enter the number for your choice.

Figure B.10 - The "Change Employee Information" screen that Sally Kwan sees about someone she manages
B.1.5 The “Employee Report Chain” screen

The “Employee Report Chain” screen, shown in Figure B.7, shows the people an employee manages, along with his or her manager.

<table>
<thead>
<tr>
<th>Employee Report Chain</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Employee:</strong></td>
</tr>
<tr>
<td>1. SALLY A. KWAN, Department C01, MANAGER</td>
</tr>
<tr>
<td><strong>Manager:</strong></td>
</tr>
<tr>
<td>2. CHRISTINE I. HAAS, Department A00, PRES</td>
</tr>
<tr>
<td><strong>Managed Employees:</strong></td>
</tr>
<tr>
<td>3. KIM N. NAIZ, Department C01, ANALYST</td>
</tr>
<tr>
<td>4. HEATHER A. NICHOLLS, Department C01, ANALYST</td>
</tr>
<tr>
<td>5. DELORES M. QUINTANA, Department C01, ANALYST</td>
</tr>
<tr>
<td><strong>What would you like to do?</strong></td>
</tr>
<tr>
<td>0. Return to the previous menu</td>
</tr>
<tr>
<td>1. Display information about SALLY A. KWAN</td>
</tr>
<tr>
<td>2. Display information about CHRISTINE I. HAAS</td>
</tr>
<tr>
<td>3-5. Display information about the employee listed at the specified number</td>
</tr>
</tbody>
</table>

Please make a selection. Enter the number for your choice.
B.1.6 The “Company Employees” screen

The “Company Employees” screen shows all of the employees in the company, in pages of 10 employees each. Figure B.8 shows this screen.

<table>
<thead>
<tr>
<th>Company Employees</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page 1</td>
</tr>
<tr>
<td>1. BRUCE ADAMSON, Department D11, DESIGNER</td>
</tr>
<tr>
<td>2. ROY R. ALONZO, Department E21, FIELDREP</td>
</tr>
<tr>
<td>3. DAVID BROWN, Department D11, DESIGNER</td>
</tr>
<tr>
<td>4. JOHN B. GEYER, Department E01, MANAGER</td>
</tr>
<tr>
<td>5. JASON R. GOUNOT, Department E21, FIELDREP</td>
</tr>
<tr>
<td>6. CHRISTINE I. HAAS, Department A00, PRES</td>
</tr>
<tr>
<td>7. DIAN J. HEMINGER, Department A00, SALESREP</td>
</tr>
<tr>
<td>8. EILEEN W. HENDRICKSON, Department E11, MANAGER</td>
</tr>
<tr>
<td>9. JAMES J. JEFFERSON, Department D21, CLERK</td>
</tr>
<tr>
<td>10. REBA K. JOHN, Department D11, DESIGNER</td>
</tr>
</tbody>
</table>

What would you like to do?
- 0. Return to the previous menu
- 1-10. Display information about the employee listed at the specified number
- 11. Display the next page of employees

Please make a selection. Enter the number for your choice.

Figure B.8 - The "Company Employees" screen

B.1.7 The “Increase Employee Bonuses” screen

The “Increase Employee Bonuses” screen is only available to the president of the company, Christine Haas. Figure B.9 shows how she sees an option for it on the main menu. Figure B.10 shows the screen.
Figure B.9 - The "Main Menu" screen, as seen by the company president
Appendix C – Up and Running with DB2 Express-C 321

Figure B.10 - The "Increase Employee Bonuses" screen

B.2 Using the sample application for the “In practice” sections

Several of the chapters of this book contain “In practice” sections that refer to the Human Resources Manager application. In some of the sections, you can use the pureQuery API to implement portions of Human Resources Manager. This enables you to practice using the inline style or by creating pureQuery data access objects. In other sections, you can
use the pureQuery Client Optimizer to optimize a JDBC implementation of Human Resources Manager.

**B.2.1 Packages in the GettingStartedWithPureQuery project**

The `GettingStartedWithPureQuery` project contains the packages `hrm.completed` and `hrm.inProgress`. The `hrm.completed` package contains the completed application. The contents of the `hrm.inProgress` package are almost identical to those of the `hrm.completed` package, but much of the pureQuery code has been removed, and comments that say `TODO: Add pureQuery code here.` have been added. In the “In practice” sections, you can add the missing code. In many places, pureQuery code was removed and replaced with `null`. Therefore, if you try to view a screen that you have not finished developing, a `NullPointerException` will probably occur.

In each of the `hrm.completed` and `hrm.inProgress` packages, multiple packages contain a file named `HumanResourcesManager.java`. In the different packages, Human Resources Manager is implemented differently to highlight different aspects of pureQuery. The sub-packages are as follows:

- `pdqInlineStyle` -- Implemented using the pureQuery inline style
- `pdqInlineStyleWithHandlers` -- Uses the pureQuery inline style, and also handlers
- `pdqMethodStyle` -- Uses the pureQuery annotated method data access objects
- `pdqMethodStyleWithHandlers` -- Uses the pureQuery annotated method data access objects, and also handlers
- `jdbc` -- Uses standard JDBC

Figure B.11 shows the package structure of the `GettingStartedWithPureQuery` project.
Figure B.11 - The package structure of the GettingStartedWithPureQuery project
B.2.2 Files in the GettingStartedWithPureQuery project

As mentioned above, each of the packages in the GettingStartedWithPureQuery project contains a file named HumanResourcesManager.java that implements Human Resources Manager in a particular way. The project contains a few other files that you will need to use. Three of these files are

pdq.DataSourceHumanResourcesManager.properties, Default.genProps, and Default.bindProps.

B.2.2.1 The properties file pdq.DataSourceHumanResourcesManager.properties

You can use properties called pureQuery properties to control the behavior of pureQuery. You can use properties files to specify pureQuery properties. Later chapters in this book describe pureQuery properties and how you can use them. The relevant “In practice” sections instruct you to specify properties in the properties file named pdq.DataSourceHumanResourcesManager.properties. This file is in the src directory, as shown in Figure B.12.

![Properties file](image)

Figure B.112 - The properties file

pdq.DataSourceHumanResourcesManager.properties

B.2.2.2 The options files Default.genProps and Default.bindProps

Some utilities that pureQuery provides allow you to specify options in options files. Later chapters in this book describe options files. The relevant “In Practice” sections instruct you to specify options in two options files named Default.genProps and Default.bindProps. These files are in the pureQueryFolder directory. Also in pureQueryFolder, there is a directory named optionsFilesForReference. This directory includes completed versions of Default.genProps and Default.bindProps that you can refer to if you want help. Figure B.13 shows these files and directories.
B.3 Setting up the GettingStartedWithPureQuery project

B.3.1 Import the GettingStartedWithPureQuery project into your workspace

Import the GettingStartedWithPureQuery project into your workspace.

1. Start by downloading GettingStartedWithPureQuery.zip to your computer.
2. Then, in IBM Optim Development Studio, choose Import from the File menu. (Figure B.14)
3. In the Import dialogue, select Existing Projects into Workspace from the General folder, and click Next. (Figure B.15)
4. Choose Select archive file: and browse to locate GettingStartedWithPureQuery.zip. Click Finish. (Figure B.16)
Figure B.134 - From the File menu, select Import....
Figure B.145 - From the General folder, select Existing Projects into Workspace
B.3.2 Add pureQuery support to the project

Add pureQuery support to the **GettingStartedWithPureQuery** project following the instructions in Chapter 2. When you choose the database connection for the project to use, choose a connection to the **SAMPLE** database. On the Add pureQuery Support pane of the Add pureQuery Support wizard, do not check the box next to **Enable SQL capturing and binding for JDBC applications**. This check box enables Client Optimizer functionality. It will be discussed in Chapter 6.

B.3.3 Modify the **javax.sql.DataSource** objects to use your connection information

Locate the class **hrm.HRMDatasourceFactory** in the **GettingStartedWithPureQuery** project. It is shown in Figure B.17. This class defines two **DataSource** objects -- one that Human Resources Manager uses, and one that the test application uses. Modify these two objects so that they contain the connection information for your **SAMPLE** database.
B.3.4 Run the Human Resources Manager application

At this point, you should be able to run the implementations of the Human Resources Manager application in the hrm.completed package. Browse to some of the HumanResourcesManager.java files in the hrm.completed package and run them as Java applications. When the application asks for an employee number, you can supply the employee number of any employee in the EMPLOYEE table of the SAMPLE database. For example, you supply 000030 to log in as Sally Kwan, and supply 000010 to log in as Christine Haas.
Appendix C  - Up and Running with DB2 Express-C

This appendix is a good foundation for learning about DB2. This appendix is streamlined to help you get up and running with DB2 quickly and easily.

In this appendix you will learn about:

- DB2 packaging
- DB2 installation
- DB2 Tools
- The DB2 environment
- DB2 configuration
- Connecting to a database
- Basic sample programs
- DB2 documentation

Note:
For more information about DB2, refer to the free e-book *Getting Started with DB2 Express-C* that is part of this book series.

C.1 DB2: The big picture

DB2 is a data server that enables you to safely store and retrieve data. DB2 commands, XQuery statements, and SQL statements are used to interact with the DB2 server allowing you to create objects, and manipulate data in a secure environment. Different tools can be used to input these commands and statements as shown in Figure B.1. This figure provides an overview of DB2 and has been extracted from the *Getting Started with DB2 Express-C* e-book.
On the left-hand side of the figure, we provide examples of different commands and statements that users can issue. In the center of the figure, we list some of the tools where you can input these commands and statements, and on the right-hand side of the figure you can see the DB2 environment; where your databases are stored. In subsequent sections, we discuss some of the elements of this figure in more detail.

C.2 DB2 Packaging

DB2 servers, clients and drivers are created using the same core components, and then are packaged in a way that allows users to choose the functions they need for the right price. This section describes the different DB2 editions or product packages available.

C.2.1 DB2 Servers

*Figure B.2* provides an overview of the different DB2 data server editions that are available.
As shown in Figure B.2, all DB2 server editions are built one on top of the other. DB2 Express-C is a free version of DB2, and it is the core component of all DB2 products. When additional functionality is added to DB2 Express-C, it becomes DB2 Express. Additional functionality added to DB2 Express, becomes DB2 Workgroup, and so on. Figure B.2 illustrates why it is so easy to upgrade from DB2 Express-C to any other DB2 server should you need to in the future: *All DB2 servers editions are built based on DB2 Express-C.*

Also applications built for DB2 Express-C are applicable on other DB2 Editions as well. Your applications will function without any modifications required!

### C.2.2 DB2 Clients and Drivers

When you install a DB2 server, a DB2 client component is also installed. If you only need to install a client, you can install either the IBM Data Server Client, or the IBM Data Server Runtime Client. *Figure B.3 illustrates these two clients.*
Figure C.3 - DB2 Clients

From the above figure, you can see the IBM Data Server Runtime client has all the components you need (driver and network support) to connect and work with a DB2 Data Server. The IBM Data Server client has this same support and also includes GUI Tools and libraries for application development.

In addition to these clients, provided are these other clients and drivers:

- **DB2 Runtime Client Merge Modules for Windows**: mainly used to embed a DB2 runtime client as part of a Windows application installation
- **IBM Data Server Driver for JDBC and SQLJ**: allows Java applications to connect to DB2 servers without having to install a client
- **IBM Data Server Driver for ODBC and CLI**: allows ODBC and CLI applications to connect to a DB2 server without having to install a client
- **IBM Data Server Driver Package**: includes a Windows-specific driver with support for .NET environments in addition to ODBC, CLI and open source. This driver was previously known as the IBM Data Server Driver for ODBC, CLI and .NET.

There is no charge to use DB2 clients or drivers.

**C.3 Installing DB2**

In this section we explain how to install DB2 using the DB2 setup wizard.

**C.3.1 Installation on Windows**

DB2 installation on Windows is straight-forward and requires the following basic steps:

1. Ensure you are using a local or domain user that is part of the Administrator group on the server where you are installing DB2.
2. After downloading and unzipping DB2 Express-C for Windows from this link, look for the file `setup.exe`, and double-click on it.
3. Follow the self-explanatory instructions from the wizard. Choosing default values is normally sufficient.
4. The following is performed by default during the installation:
   - DB2 is installed in `C:\Program Files\IBM\SQLLIB`
   - The DB2ADMNS and DB2USERS Windows operating system groups are created.
   - The instance **DB2** is created under `C:\Program Files\IBM\SQLLIB\DB2`
   - The DB2 Administration Server (DAS) is created
- Installation logs are stored in:
  My Documents\DB2LOG\db2.log
  My Documents\DB2LOG\db2wi.log
- Several Windows services are created.

C.3.2 Installation on Linux

DB2 installation on Linux is straightforward and requires the following basic steps:

1. Log on as the Root user to install DB2.
2. After downloading DB2 Express-C for Linux from this link, look for the file db2setup, and execute it: ./db2setup
3. Follow the self-explanatory instructions from the wizard. Choosing default values is normally sufficient.
4. The following is performed by default during installation:
   - DB2 is installed in /opt/ibm/db2/V9.7
   - Three user IDs are created. The default values are listed below:
     db2inst1 (instance owner)
     db2fenc1 (Fenced user for fenced routines)
     dasusr1 (DAS user)
   - Three user groups are created corresponding to the above user IDs:
     db2iadm1
     db2fadm1
     dasadm1
   - Instance db2inst1 is created
   - The DAS dasusr1 is created
   - Installation logs are stored in:
     /tmp/db2setup.his
     /tmp/db2setup.log
     /tmp/db2setup.err

C.4 DB2 Tools

There are several tools that are included with a DB2 data server such as the DB2 Control Center, the DB2 Command Editor, and so on. Starting with DB2 version 9.7 however; most of these tools are deprecated (that is, they are still supported but no longer enhanced) in favor of a new tool called IBM Optim Development Studio. Optim Development Studio is provided as a separate package not included with DB2.
C.4.1 IBM Optim Development Studio

Optim Development Studio is the primary tool to use for database administration, and database development with DB2. Optim Development Studio is free. It can run on Linux and Windows, and is part of the IBM Integrated Data Management portfolio of products. The development of Optim Development Studio follows a schedule which is not tied to the releases of DB2; however the products do coordinate their releases as often as possible. Figure B.4 shows the Optim Development Studio interface.

Figure C.4 - IBM Optim Development Studio

If you are familiar with Eclipse, you will note that Optim Development Studio is based on Eclipse. Typically, with Optim Development Studio, you will work within the Data perspective window (highlighted in the figure at the top right corner). You can also switch to the Java perspective, if you are developing a Java program. There are two views highlighted in the figure:

- Data Project Explorer (top left)
- Data Source Explorer (bottom left)

The Data Project Explorer view is used by database developers to work with SQL scripts, XQuery, stored procedures, UDFs, and Data Web services.
The Data Source Explorer view is used by database administrators to manage DB2 instances and databases. Using this view you can perform most of the functionality previously available in the Control Center.

In the figure, the view with title PROCEDURE1 is an editor for the procedure being highlighted in the Data Project Explorer. Depending on the task you are executing, editors or other windows will appear, allowing you to either code or perform more configurations.

With IBM Optim Development Studio, you can also work with other data servers, such as Informix. Companies that work with several data servers and have a small DBA or database developer team now have the convenience of working and managing all of their data servers from one tool.

**Note:**
For more information about Optim Development Studio, refer to the *Getting Started with Data Studio* free e-book which is part of this book series.

### C.4.2 Control Center

Prior to DB2 9.7, the primary DB2 tool for database administration was the Control Center, as illustrated in *Figure B.5*. This tool is now deprecated, but still included with DB2 servers.
To start the Control Center on Windows use `Start -> Programs -> IBM DB2 -> DB2COPY1 (Default) -> General Administration Tools -> Control Center` or alternatively, type the command `db2cc` from a Windows Command Prompt or Linux shell.

The Control Center is a centralized administration tool that allows you to:

- View your systems, instances, databases and database objects;
- Create, modify and manage databases and database objects;
- Launch other DB2 graphical tools

The pane on the left-hand side provides a visual hierarchy of the database objects on your system(s), providing a folder for Tables, Views, etc. When you double-click a folder (for example, the Tables folder, as shown in Figure C.5), the pane on the top right will list all of the related objects, in this case, all the tables associated with the `SAMPLE` database. If you select a given table in the top right pane, the bottom right pane provides more specific information about that table.

Right-clicking on the different folders or objects in the Object tree will bring up menus applicable to the given folder or object. For example, right-clicking on an instance and choosing `Configure parameters` would allow you to view and update the parameters at the
instance level. Similarly, if you right-click on a database and choose Configure parameters, you would be able to view and update parameters at the database level.

C.4.3 Command Line Tools
There are three types of Command Line tools:

- DB2 Command Window (only on Windows)
- DB2 Command Line Processor (DB2 CLP)
- DB2 Command Editor (GUI-based, and deprecated)

These tools are explained in more detail in the next sections.

C.4.3.1 DB2 Command Window
The DB2 Command Window is only available on Windows operating systems; it is often confused with Windows Command Prompt. Though they look the same, the DB2 Command Window, however, initializes the environment for you to work with DB2. To start this tool, use Start -> Programs -> IBM DB2 -> DB2COPY1 (Default) -> Command Line Tools -> Command Window or alternatively, type the command db2cmd from a Windows Command Prompt to launch it on another window. Figure C.6 shows the DB2 Command Window.

![DB2 Command Window](image)

**Figure C.6 - The DB2 Command Window**

You can easily identify you are working in the DB2 Command Window by looking at the window title which always includes the words DB2 CLP as highlighted in the figure. From
the DB2 Command Window, all commands must be prefixed with `db2`. For example, in the above figure, two statements are issued:

```
db2 connect to sample
db2 select * from staff
```

For Linux, the equivalent of the DB2 Command Window is simply the Linux shell (or terminal) where the DB2 environment has been set up by executing the `db2profile` file. This file is created by default and added to the `.login` file for the DB2 instance owner. By default the DB2 instance owner is `db2inst1`.

### C.4.3.2 DB2 Command Line Processor

The DB2 Command Line Processor (CLP) is the same as the DB2 Command Window, with one exception that the prompt is `db2=>` rather than an operating system prompt. To start the DB2 Command Line Processor on Windows, use `Start -> Programs -> IBM DB2 -> DB2COPY1 (Default) -> Command Line Tools -> Command Line Processor` or alternatively from a DB2 Command Window or Linux shell type `db2` and press `Enter`. The prompt will change to `db2` as shown in Figure C.7.

![Figure C.7 - The DB2 Command Line Processor (CLP)](image-url)
Note that Figure C.7 also illustrates that when working in the CLP, you do not need to prefix commands with DB2. To exit from the CLP, type `quit`.

### C.4.3.3 DB2 Command Editor

The DB2 Command Editor is the GUI version of the DB2 Command Window or DB2 Command Line Processor as shown in Figure C.8. This tool is deprecated for DB2 version 9.7.

![Figure C.8 - The DB2 Command Editor](image)

### C.5 The DB2 environment

*Figure C.9* provides a quick overview of the DB2 environment.
Figure C.9 - The DB2 Environment

The figure illustrates a server where DB2 Express-C has been installed. The smaller boxes in light green (Environment Variables, Database Manager Configuration File, Database Configuration File, DB2 Profile Registry) are the different areas where a DB2 server can be configured, and they will be explained in more detail in the next section. The larger dark green box represents an instance which in this example has the name *myinst*.

An **instance** is an environment where database objects can be created. On the same server, you can create several instances, each of which is treated independently. For example, you can use an instance for development, another one for test, and another one for production. *Table C.1* shows some useful commands you can use at the instance level. Note that the commands shown in this section can also be performed from DB2 GUI Tools.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>db2start</td>
<td>Starts the current instance</td>
</tr>
<tr>
<td>db2stop</td>
<td>Stops the current instance</td>
</tr>
<tr>
<td>db2icrt &lt;instance_name&gt;</td>
<td>Creates a new instance</td>
</tr>
<tr>
<td>db2idrop &lt;instance_name&gt;</td>
<td>Drops an instance</td>
</tr>
<tr>
<td>db2list</td>
<td>Lists the instances you have on your system</td>
</tr>
<tr>
<td>db2 get instance</td>
<td>Lists the current active instance</td>
</tr>
</tbody>
</table>
Table C.1 - Useful instance-level DB2 commands

Within an instance you can create many databases. A database is a collection of objects such as tables, views, indexes, and so on. For example, in Figure C.9, the database MYDB1 has been created within instance myinst. Table C.2 shows some commands you can use at the database level.

<table>
<thead>
<tr>
<th>Command/SQL statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>create database &lt;database_name&gt;</td>
<td>Creates a new database</td>
</tr>
<tr>
<td>drop database &lt;database_name&gt;</td>
<td>Drops a database</td>
</tr>
<tr>
<td>connect to &lt;database_name&gt;</td>
<td>Connects to a database</td>
</tr>
<tr>
<td>create table/create view/create index</td>
<td>SQL statements to create table, views, and indexes respectively</td>
</tr>
</tbody>
</table>

Table C.2 - Commands and SQL Statements at the database level

C.6 DB2 configuration

DB2 parameters can be configured using the Configuration Advisor GUI tool. The Configuration Advisor can be accessed through the Control Center by right clicking on a database and choosing Configuration Advisor. Based on your answers to some questions about your system resources and workload, the configuration advisor will provide a list of DB2 parameters that would operate optimally using the suggested values. If you would like more detail about DB2 configuration, keep reading. Otherwise, use the Configuration Advisor and you are ready to work with DB2!

A DB2 server can be configured at four different levels as shown earlier in Figure C.9:

- **Environment variables** are variables set at the operating system level. The main environment variable to be concerned about is DB2INSTANCE. This variable indicates the current instance you are working on, and for which your DB2 commands will apply.

- **Database Manager Configuration File (dbm cfg)** includes parameters that affect the instance and all the databases it contains. Table C.3 shows some useful commands to manage the dbm cfg.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
get dbm cfg retrieves information about the dbm cfg

update dbm cfg using <parameter_name> <value> updates the value of a dbm cfg parameter

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get dbm cfg for &lt;database_name&gt;</td>
<td>Retrieves information about the dbm cfg for a given database</td>
</tr>
<tr>
<td>update dbm cfg for &lt;database_name&gt; using &lt;parameter_name&gt; &lt;value&gt;</td>
<td>Updates the value of a dbm cfg parameter</td>
</tr>
</tbody>
</table>

Table C.3 - Commands to manipulate the dbm cfg

- Database Configuration File (db.cfg) includes parameters that affect the particular database in question. Table C.4 shows some useful commands to manage the db.cfg.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get db cfg for &lt;database_name&gt;</td>
<td>Retrieves information about the db cfg for a given database</td>
</tr>
<tr>
<td>update db cfg for &lt;database_name&gt; using &lt;parameter_name&gt; &lt;value&gt;</td>
<td>Updates the value of a db cfg parameter</td>
</tr>
</tbody>
</table>

Table C.4 - Commands to manipulate the db cfg

- DB2 Profile Registry variables includes parameters that may be platform specific and can be set globally (affecting all instances), or at the instance level (affecting one particular instance). Table C.5 shows some useful commands to manipulate the DB2 profile registry.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>db2set -all</td>
<td>Lists all the DB2 profile registry variables that are set</td>
</tr>
<tr>
<td>db2set &lt;parameter&gt;=&lt;value&gt;</td>
<td>Sets a given parameter with a value</td>
</tr>
</tbody>
</table>

Table C.5 - Commands to manipulate the DB2 profile registry

C.7 Connecting to a database

If your database is local, that is, it resides on the same system where you are performing your database operation; the connection setup is performed automatically when the database is created. You can simply issue a connect to database_name statement to connect to the database.

If your database is remote, the simplest method to set up database connectivity is by using the Configuration Assistant GUI tool following these steps:
5. Start the Configuration Assistant from the system where you want to connect to the database. To start this tool, use the command \texttt{db2ca} from a Windows command prompt or Linux shell. Figure C.6 shows the Configuration Assistant.

![Configuration Assistant](Image)

**Figure C.6 - The DB2 Configuration Assistant**

6. From the Configuration Assistant, click on the Selected -> Add database using Wizard menu.

7. From the Select how you want to set up a connection window, you can use Search the network if your network is small without many hubs. If you know the name of the server where DB2 resides, choose Known systems and drill down all the way to the database you want to connect. Proceed with the wizard using default values. If you do not know the name of your system, choose Other systems (Search the network). Note that this may take a long time if your network is large.

8. If Search the network does not work, go back to the Select how you want to set up a connection window, and choose Manually configure a connection to a database. Choose TCP/IP and click next. Input the hostname or IP address where your DB2 server resides. Input either the service name or the port number.

9. Continue with the wizard prompts and leave the default values.

10. After you finish your set up, a window will pop up asking you if you want to test your connection. You can also test the connection after the setup is finished by right-clicking on the database, and choosing Test Connection.
C.8 Basic sample programs

Depending on the programming language used, different syntax is required to connect to a DB2 database and perform operations. Below are links to basic sample programs which connect to a database, and retrieve one record. We suggest you first download all the sample programs in this section:

- CLI program
- ODBC program
- C program with embedded SQL
- JDBC program using Type 2 Universal (JCC) driver
- JDBC program using Type 4 Universal (JCC) driver
- Visual Basic and C++ ADO program - Using the IBM OLE DB provider for DB2 (IBMDADB2)
- Visual Basic and C++ ADO program - Using the Microsoft OLE DB Provider for ODBC (MSDASQL)
- Visual Basic and C# ADO.Net using the IBM DB2 .NET Data Provider
- Visual Basic and C# ADO.Net using the Microsoft OLE DB .NET Data Provider
- Visual Basic and C# ADO.Net using the Microsoft ODBC .NET Data Provider

C.9 DB2 documentation

The DB2 Information Center provides the most up-to-date online DB2 documentation. The DB2 Information Center is a web application. You can access the DB2 Information Center online, or you can download and install the DB2 Information Center to your local computer. Links to the online DB2 Information Center as well as downloadable versions are available using this link.
Appendix D – Static SQL in DB2

D.1 What is static SQL?

Static SQL means different things to different database vendors. This appendix describes static SQL and its benefits for DB2.

Whenever DB2 executes an SQL statement, the system creates an access plan describing the steps it will take, and in what order, to perform the execution. For example, the system might create a plan by analyzing table statistics to decide what indexes to use and which columns to access first to perform a join.

When SQL is executed dynamically, the access plan is selected on the fly, at runtime, as part of execution processing. As a result, dynamic execution of an SQL statement is usually delayed by the creation of its access plan. Even though mechanisms such as dynamic statement caching exist to improve this sequence of steps, in dynamic and improvised workloads, generating access plans can be a key negative factor in the overall system performance.

Static SQL is an alternative to this dynamic execution model. When SQL is executed statically, it uses a precompiled version of the SQL statement, which means that the access path was generated before the statement was executed (see the section on “Precompile and Bind” for more information). As a result, when the SQL execution request is sent to the database, the system already knows the access plan it will use to evaluate the statement.

*Figure D.5* compares the runtime processing required for dynamic vs. static SQL execution modes.
D.2 Prerequisites for Static SQL

Since static SQL must be defined ahead of execution time, everything must be known about the static statement when the application is defined (precompiled and bound). The only unknowns allowed for a static statement are the values of the host variables (parameters) within the SQL string. These can be supplied at runtime. However, table names, column names, and the rest of the SQL statement structure must be fully predefined.

D.3 Benefits of static SQL

The primary benefits of static SQL are that the access plan can be predetermined, and a static SQL security model can be used. In addition, the usage of pre-defined access plans will deliver more consistent performance compared to a dynamic execution model.

D.3.1 Predetermined access plan for more consistent performance

As mentioned earlier, a predetermined access plan means that the database engine does not need to calculate an access plan at runtime. This cuts a piece of required processing during statement execution, and reduces the total execution time. Prior to execution, you can develop, test, optimize, and compare access paths. The best plan can then be selected and set for execution. A variety of automated tools are used to analyze the performance of SQL statements and evaluate access plans. The plans can then be modified using programmatic or human input.
Once the optimal plan is identified it can be locked in for the static SQL statement. This means that for static SQL, performance can be maximized and then locked-in via a fixed access plan. For certain applications it is essential to understand and lock in performance behavior before a production system is rolled out.

Dynamic SQL can approach the benefits of a predetermined access plan by using a dynamic statement cache. With a dynamic statement cache, the access plan is calculated the first time a dynamic statement is executed. The plan for the dynamic SQL string is cached by the database and reused the next time the same statement is executed. For certain applications and environments that have high dynamic statement cache hit ratios, it’s possible to approach the database performance of static SQL, by limiting the number of times access plans are calculated at runtime. However, these dynamic applications must be coded to produce maximum cache hits, and the dynamic cache environment must be well tuned, in order to get a similar advantage using a dynamic cache to the one obtained with static SQL. Static SQL is the only way to completely lock in the access plan ahead of execution time for every statement.

D.3.2 Package level security

In addition to a locked-in access plan, static SQL provides a different SQL security model using package level privileges. Because static SQL statements are known ahead of time, users can be given authorization only for a specific set of SQL statements. These statements are combined into a unit called a package. Static SQL statements can be logically grouped into packages according to the needs of the security model. Users are only granted permission to execute packages containing SQL that matches their access level. With package level privileges, users cannot execute any SQL statements unless the statements are contained in the packages the users have been granted the privilege to execute.

This provides an important alternative to table-based privileges, which is the common security model for dynamic SQL. With dynamic SQL, privileges are granted for specific table operations, such as read or update on a table. The risk with granting table privileges is they do not limit users to executing specific SQL statements. Users could potentially execute any query or any update SQL statement on a table, once they have been granted read or update privileges on the table. Even though a user’s application may be hard-coded with a specific SQL string, a malicious user could potentially use table privileges to execute dynamic SQL that is not contained within their application. Dynamic SQL injection is one form of malicious behavior that takes advantage of this fact. Static SQL, with a previously designed fix set of packages, and package level privileges, prevents this unwanted behavior. Only the package owner, not each user, needs to be granted table level privileges for static SQL. The package owner is usually the administrator who binds the packages. Therefore good static package design can eliminate the risk of a user maliciously executing unwanted SQL.
D.4 The mechanics of static SQL with DB2

When an application executes static SQL statements in DB2, it needs an alternate way of
telling the server which statement it wants to execute. Since static SQL is already defined
at the server, the application does not need to send an SQL string to the server at runtime,
as it does for dynamic SQL. It only needs to identify to the server which pre-defined SQL
statement to execute. DB2 uses sections and packages as identifiers to organize and
manage the static SQL statements that have been defined. (higher level organizing units
such as collections, or for DB2 for z/OS plans or DBRMs are out of the scope of this
appendix.)

A package is the basic organizing unit for SQL statements at the server. A package is a
named and uniquely identifiable grouping of related SQL statements.

A section is the building block of a package. Each section is associated with one SQL
statement. A package usually has one or more sections identified by an integer number.
DB2 uses sections to execute both dynamic and static SQL.

Packages and sections are internally used by DB2 for execution of SQL statements in both
dynamic and static mode. While in static mode these organizing units are explicitly created
by the user at precompile and bind time, in dynamic mode they are used implicitly by DB2
when it binds the dynamic SQL statements. When connecting to DB2 from a Java
application, DB2 uses the implicit JDBC packages and sections to store the access plans
for the dynamically executed SQL statements. Because execution is dynamic, the SQL
string is provided by the application at runtime. The statement and its dynamically
determined access plan are depicted outside the package because the package does not
contain the definition of the statement or know the access plan prior to execution time, as
illustrated in Figure D.6.
On the other hand, static sections are pre-associated with an SQL string. A static section will only be used to execute the single SQL string defined in the section itself, and the section will use the access plan that was pre-determined when the package was defined (precompiled and bound) in the database server. This reduces the number of runtime steps required to execute the statement. In Figure D.7, three static packages are depicted in the DBMS. Each package contains sections with predetermined SQL strings and access plans, managed by the database catalog and package directory. The application does not need to send the SQL string at runtime or perform the steps required to ensure an access plan is created at runtime (prepare). The application simply identifies which predefined statement (identified by its package and section number) to execute. The figure shows a statement in the Payroll package being executed. DB2 will check the package authorizations, retrieve the SQL string and plan and execute the statement immediately.
Figure D.19 - Static SQL statements and access plans are known prior to execution time and stored by DB2. This results in fewer execution steps at runtime.

D.4.1 Precompiling and binding

Precompiling and **binding** are the terms commonly used in DB2 to describe the steps required to create an application that will execute static SQL.

Precompiling refers to writing the application source code and the SQL strings the application will execute, and preparing the application code for execution. It includes all the steps required by the programming language, framework, or platform to produce and compile the client code. But from the database perspective, the application is not fully "compiled" until the static SQL has been given to the database server and the database server has determined the access plan to execute it.

Binding refers to the process by which the static SQL strings are provided to a database server, the database server prepares its access plan, then stores the information it needs to execute the static SQL at runtime. Once binding is complete and execute privileges have been granted on the resulting package, the application can be run against the database package executing static SQL.

D.4.2 Precompiling and binding static SQL with pureQuery

Since pureQuery supports several different ways to create applications which execute statically, there is more than one way to "precompile" a pureQuery program:

- Write a pureQuery annotated method style Java data access interface, define the SQL strings it will use, and generate the java method implementations for each statement using Optim Development Studio (See Chapter 11).
Appendix D – Static SQL in DB2

- Perform SQL capture on a pre-existing JDBC or CLI application while it executes dynamic SQL, and then configure the capture file (See Chapter 6).
- Use Websphere JPA support, Hibernate, or another persistence framework integration with pureQuery to create a static application within the given persistence framework (See Chapter 15, Chapter 16).

The pureQuery support for each of the above also includes a pureQuery StaticBinder utility. The StaticBinder utility is launched differently depending upon the development environment used. The pureQuery StaticBinder supports specification of a set of SQL to bind (for example a pureQuery annotated method style data access interface, or a pureQuery capture file containing bindable SQL), database server connection properties that indicate where to bind the SQL, and package options and bind properties supported by the database server. For more information on the pureQuery StaticBinder tool, refer to Chapter 7 - The StaticBinder Utility.

D.5 Summary

Static SQL provides an important alternative to standard dynamic execution by offering a fixed access plan and package level security, to maximize SQL optimization and support fine-tuning of the security model. Static SQL can improve manageability of, and control over, any data access application whose SQL strings are known at pre-compile time.
Appendix E - Options Files

This appendix discusses options files. You can use options files to specify the options that you want the Generator utility, the Configure utility, and the StaticBinder utility to use.

E.1 Options files: The big picture

pureQuery provides multiple utilities. All of these utilities allow you to specify options on the command line. For the Generator utility, the Configure utility, and the StaticBinder utility, you can also specify the pureQueryXml files or interfaces that you want the utility to process, along with the options that you want the utility to use, in a type of file that is known as an options file.

pureQuery allows two types of options files: genProps, which is used for the Configure utility (described in Section 9.6) and the Generator utility (described in Section 6.3), and bindProps, which is used for the StaticBinder utility (described in Chapter 10). The contents of the two types of options files are mostly the same, but they have a few differences. The convention is to use .genProps as the file extension for genProps files, and .bindProps as the file extension for bindProps files.

Options files make it easy to run a utility one time, while specifying many interfaces or pureQueryXml files, each processed using distinct options. Additionally, if you want IBM Optim Development Studio to run the utility for you, the only way you can specify options is by using options files.

The IBM Optim pureQuery Runtime documentation for the utilities provides details about using options files from the command line. The IBM Optim Development Studio documentation provides details about using genProps and bindProps files in the tool.

E.2 Content of options files

An options file is a text file that lists interfaces and pureQueryXml files to be processed, and the options to apply to each interface or file. Each interface and pureQueryXml file must be listed on a separate line in the options file, followed by an equal sign (=), and then any options that pureQuery should use when it processes that item. The extensions of pureQueryXml files must be either .pdqxml or .xml.
All of the options for an entry must be on a single line; that is, a line break signals the beginning of the next entry. For each line, all of the characters to the left of the equal sign are considered part of the interface or pureQueryXml file name, so do not include any quotation marks or other characters that you do not want to be part of the name. The options and values to the right of the equal sign are space-separated, just as they are on the command line. Therefore, to the right of the equal sign, if the value of an option contains a space, you must surround the value in double quotes.

In genProps files, you can only provide one entry for each interface or pureQueryXml file. You can provide multiple entries for the interfaces and pureQueryXml files that are in a bindProps file, however. It does not make sense to allow duplicate entries in genProps files, since the Generator utility and the Configure utility would repeatedly create or modify the same file. On the other hand, it makes sense for the StaticBinder utility to process a single interface or pureQueryXml file more than once; for example, you might want to bind the SQL in one interface or file in multiple database servers.

Listing E.18.2 shows how you could specify two interfaces, two pureQueryXml files, and one statement set of a third pureQueryXml file in an options file. You can specify specific statement sets only for the StaticBinder utility, so the last line of the listing is only valid in a bindProps file.

myPackage1.MyInterface1 = <options for myPackage1.MyInterface1>
myPackage1.MyInterface2 = <options for myPackage1.MyInterface2>
C:\path1\captureFile1.pdqxml = <options for captureFile1.pdqxml>
C:\path2\captureFile2.pdqxml = <options for captureFile2.pdqxml>
C:\path3\captureFile3.pdqxml:MYPKGA = <options for the statement set MYPKGA>

Listing E.18.2 - Example of listing interfaces and pureQueryXml files in an options file

Additionally, in each options file, you can provide default options on a line that starts with defaultOptions. pureQuery uses each option that you specify on the defaultOptions line for all the items, except for those items where the option is specified with a different value. You can only specify one line that starts with defaultOptions. Listing E.18.3 shows how you could specify these options.

defaultOptions = <options to use as defaults>

Listing E.18.3 - Example of listing defaultOptions in an options file

In addition to lines of options, you can have comments in an options file. Use the # character to designate lines as comments, as in Listing E.18.4:

# This line is a comment

Listing E.18.4 - Example of a comment in an options file

If you run a utility from the command line and specify an options file, you can specify some of the options on the command line as well. When you specify an option both on the
command line and in the options file, pureQuery will use the value that is on the command line.

E.3 Specifying option files in IBM Optim Development Studio

pureQuery-enabled projects in IBM Optim Development Studio contain a folder named `pureQueryFolder`. The folder contains two options files: one named `Default.genProps`, and one named `Default.bindProps`. Notice these files in Figure E.18.8.

![Figure E.18.20 - The options files Default.genProps and Default.bindProps in a pureQuery-enabled project](image)

By default, IBM Optim Development Studio runs the Generator utility to generate implementation classes when you build your application. It also runs the Configure utility automatically. By default, you must use menu options to instruct it when to run the StaticBinder utility, although you can set options to have it run that utility automatically as well. When IBM Optim Development Studio runs the Generator and Configure utilities, `Default.genProps` specifies the options to use. When it runs the StaticBinder utility, `Default.bindProps` specifies the options. If the interface or pureQueryXml file is not in the corresponding options file, the utility just uses any options that are on the `defaultOptions` line.

In bindProps files, you can have more than one entry for each interface or pureQueryXml file. When you bind an interface, a pureQueryXml file, or a single statement set from a pureQueryXml file, the utility binds using every entry in the options file that refers to the item that you selected. If you specify a pureQueryXml file, the utility also binds entries that represent statement sets in the pureQueryXml file. This allows different statement sets within a single pureQueryXml file to be bound using different options.

The StaticBinder utility performs binds for entries in the order of the entries in the file. When you specify a pureQueryXml file that is not in the bindProps file at all, or for which only certain statement sets are specified in the file, the utility first binds the file using the default options, and then it binds every entry in the options file that contains a statement set from the file. Performing a bind in IBM Optim Development Studio is equivalent to running the StaticBinder utility on the command-line while specifying an options file and interfaces or pureQueryXml files.
Through menu options and content assist, IBM Optim Development Studio can help you to create options files. It can help you to add interfaces and pureQueryXml files to your options files, and it can help you add options. IBM recommends that you use Development Studio assistance to add pureQueryXml files to your options files, rather than adding them manually. In order for the utilities to determine which options to use, the paths of pureQueryXml files in the options files must exactly match the paths that IBM Optim Development Studio uses when it runs the utilities. Using assistance will ensure the paths match.

E.4 Specifying option files on the command line

E.4.1 Specifying an options file on the command line, without interfaces or pureQueryXml files

For the Generator, Configure, and StaticBinder utilities, you can use the option –optionsFile to specify an options file that contains the interfaces and pureQueryXml files that you want the utility to process, along with the options that it should use for each of them. pureQuery processes each entry in the options file, in the order in which they are listed. Listing E.18.5 shows an example of specifying an options file.

-optionsFile "C:\path\Default.bindProps"

Listing E.18.5 - Example of specifying an options file that lists the interfaces and pureQueryXml files to bind

If you run a utility from the command line and specify an options file, you can specify some of the options on the command line as well. When you specify an option both on the command line and in the options file, pureQuery will use the value that is on the command line.

E.4.2 Specifying an options file on the command line, in addition to interfaces or pureQueryXml files

For the StaticBinder utility, you can specify both an options file and interfaces or pureQueryXml files on the command line. If you do this, pureQuery binds the interfaces and pureQueryXml files that you specify on the command line, using the options that you specify on the command line, combined with the options in the bindProps file. For pureQueryXml files, the path of the file that you specify on the command line must match exactly the path provided in the options file. Listing E.18.6 shows an example of specifying an options file and a pureQueryXml file.

-optionsFile "C:\path\Default.bindProps" -pureQueryXml "C:\path\captureFile.pdqxml"

Listing E.18.6 - Example of specifying an options file and a pureQueryXml file on the command line
In bindProps files, you can have more than one entry for each interface or pureQueryXml file. If a given interface or pureQueryXml file is listed in your options file, the StaticBinder utility binds the same packages in the database for the interface or file when you specify only the options file on the command line as when you additionally specify the interface or pureQueryXml file on the command line. The difference is that in the first case, the utility also binds the other entries that are listed in the options file. When you specify an interface, a pureQueryXml file, or a single statement set from a pureQueryXml file on the command line, the utility binds using every entry in the options file that refers to the item that you selected. If you specify a pureQueryXml file, the utility also binds entries that represent statement sets that are in the pureQueryXml file. This allows different statement sets within a single pureQueryXml file to be bound using different options.

The StaticBinder utility performs binds for the entries in the order of the entries in the file. When you specify a pureQueryXml file that is not in the bindProps file at all, or for which only certain statement sets are specified in the file, the utility first binds the file using the command line and default options, and then it binds every entry in the options file that contains a statement set from the file. Refer to the IBM Optim pureQuery Runtime documentation for more details about specifying options files and interfaces or pureQueryXml files on the command line.
References

1. Published performance numbers comparing static to dynamic SQL execution on DB2 with pureQuery:

2. Static SQL:

3. DB2 security model:
Resources

Web sites

1. pureQuery platform website:
   http://www-01.ibm.com/software/data/optim/purequery-platform/
   Use this website for information about the pureQuery platform.

2. pureQuery Infocenter:
   http://publib.boulder.ibm.com/infocenter/idmhelp/dev-v2r2/index.jsp
   Use this website as your official reference for pureQuery 2.2 features.

3. Optim Development Studio and pureQuery Runtime Forum:
   Use this forum to find answers and communicate with the pureQuery community.

4. Optim family web site:
   Use this website for information about pureQuery and the entire Optim family of products.

Books

1. Free eBook: Getting Started with IBM Data Studio for DB2
   Debra Eaton, Vitor Rodrigues, Manoj K. Sardana, Michael Schenker, Kathryn Zeidenstein, Raul F. Chong. September 2010
   https://www.ibm.com/developerworks/wikis/display/db2oncampus/FREE+ebook+-+Getting+started+with+IBM+Data+Studio+for+DB2
Getting started with pureQuery couldn't be easier.

Read this book to:

- Find out what IBM® pureQuery is
- Learn how to use pureQuery tooling in Optim™ Development Studio
- Write efficient Java™ database applications using pureQuery APIs
- Learn how to optimize your existing Java database applications using pureQuery
- Understand how pureQuery can optimize Java Persistence API (JPA) and Hibernate applications
- Practice while developing a sample Java application

IBM pureQuery is a high-speed database access platform that consists of APIs, a runtime, tooling and monitoring services, all of which are designed to help you develop and tune new or existing Java database access application more easily, while optimizing performance, stability, and security. pureQuery includes features of interest to both developers and DBAs. Java developers will like the easy-to-use APIs and the Eclipse-based tooling for productivity, while DBAs will like the more efficient and secure database access made possible by pureQuery as well as the ability to better manage and control Java database applications. And pureQuery is flexible enough to fit into whatever Java framework you are currently using and can even help optimize existing Java applications, all without changing application code.

This book provides a comprehensive look at pureQuery and includes hands-on exercises using Java and DB2® Express-C, the free version of DB2.

IBM pureQuery is a key capability of the Integrated Data Management solutions from IBM. Get started with IBM pureQuery, and make your database access more efficient!

To learn more or download a trial of IBM Optim Development Studio and pureQuery Runtime, visit ibm.com/developerworks/downloads/im/datastudiodev/

To learn more or download DB2 Express-C, visit ibm.com/db2/express

To socialize and watch related videos, visit channelDB2.com

This book is part of the DB2 on Campus book series, free eBooks for the community. Learn more at db2university.com

Price: 24.99USD