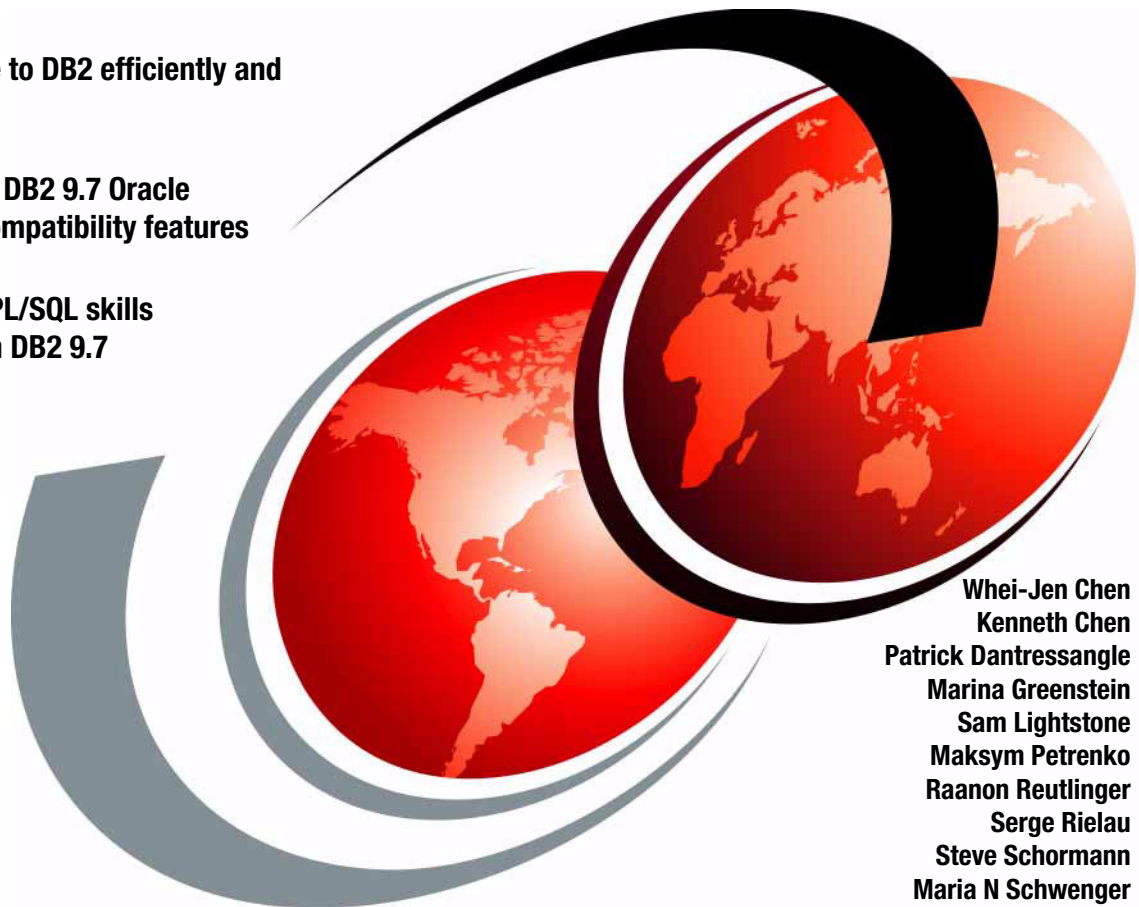


Oracle to DB2 Conversion Guide: Compatibility Made Easy

Move Oracle to DB2 efficiently and effectively

Learn about DB2 9.7 Oracle Database compatibility features

Use Oracle PL/SQL skills directly with DB2 9.7



Whei-Jen Chen
Kenneth Chen
Patrick Dantressangle
Marina Greenstein
Sam Lightstone
Maksym Petrenko
Raanon Reutlinger
Serge Rielau
Steve Schormann
Maria N Schwenger



International Technical Support Organization

**Oracle to DB2 Conversion Guide: Compatibility
Made Easy**

November 2009

Note: Before using this information and the product it supports, read the information in “Notices” on page vii.

First Edition (November 2009)

This edition applies to DB2 for Linux, UNIX, and Windows Version 9.7 Fix Pack 1, Oracle 10g, and Oracle 11g.

© Copyright International Business Machines Corporation 2009. All rights reserved.

Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Notices	vii
Trademarks	viii
 Preface	 ix
The team who wrote this book	x
Acknowledgement	xii
Become a published author	xiii
Comments welcome	xiv
 Chapter 1. Introduction	 1
1.1 DB2 family of products	2
1.1.1 DB2 editions	2
1.1.2 DB2 value added features	5
1.1.3 DB2 9 autonomic computing features	8
1.2 DB2 Oracle Database compatibility features overview	9
1.2.1 Concurrency control	10
1.2.2 Data types	11
1.2.3 Implicit casting	11
1.2.4 SQL Standard	12
1.2.5 PL/SQL	12
1.2.6 Built-in packages	15
1.2.7 Oracle specific JDBC extensions	15
1.2.8 SQL*Plus scripts	15
1.2.9 Compatibility	16
1.3 DB2 education resource	16
1.3.1 DB2 9.7 videos and articles	18
 Chapter 2. Language compatibility features	 21
2.1 DB2 compatibility features references	23
2.1.1 SQL compatibility setup	23
2.1.2 New PL/SQL and SQL types	27
2.1.3 Basic procedural statements	38
2.1.4 Control of flow statements	44
2.1.5 Condition (exceptions) handling	50
2.1.6 Cursors data type	55
2.1.7 Static and dynamic SQL support	60
2.1.8 Support for built-in scalar functions	64
2.1.9 Routines, procedures, and functions compatibility	74
2.1.10 Moving PL/SQL packages	86

2.1.11	Moving triggers	90
2.1.12	Moving SQL statements	93
2.2	Schema compatibility features	101
2.2.1	Extended data type support	102
2.2.2	Flexible schema changes in DB2	102
2.2.3	Sequences	104
2.2.4	Index enablement	105
2.2.5	Constraints enablement	107
2.2.6	Created temporary tables	108
2.2.7	Synonyms	110
2.2.8	Views and Materialized Views	111
2.2.9	Object types	112
2.2.10	Partitioning and MDC	114
2.2.11	Oracle database links and DB2 federation	124
2.2.12	Oracle Data Dictionary compatible views	128
2.3	DB2 command-line utilities	133
2.3.1	Command Line Processor Plus	133
2.3.2	Command Line Processor	139
Chapter 3.	Enablement tools	143
3.1	IBM Data Movement Tool	144
3.1.1	Moving DDLs and data with the IBM Data Movement Tool	146
3.1.2	Deploying PL/SQL objects with the IBM Data Movement Tool	148
3.1.3	Maintaining the current version of the tool	150
3.2	IBM Optim Development Studio	151
3.2.1	Moving DDLs and data with Optim Development Studio	151
3.2.2	Deploying PL/SQL objects with Optim Development Studio	153
3.3	MEET DB2 evaluation utility	155
3.4	Conclusion	156
Chapter 4.	Enablement scenario	157
4.1	Installing DB2 and creating an instance	158
4.1.1	Install DB2 using db2setup	158
4.2	Enabling SQL compatibility	160
4.3	Creating and configuring the target DB2 database	160
4.4	Defining an additional database user with the Database Administrator authority	163
4.5	Enabling using the IBM Data Movement Tool	163
4.5.1	Getting started	163
4.5.2	Extracting DDL, table data, and PL/SQL objects	164
4.5.3	Deploying DDL and table data into a DB2 target database	166
4.5.4	Deploying PL/SQL objects into a DB2 target database	166
4.5.5	Resolving incompatibilities with Interactive Deploy	168

4.6 Verification of enablement	171
4.7 Summary	172
Chapter 5. Application conversion	173
5.1 DB2 application development introduction	174
5.1.1 Driver support	174
5.1.2 Embedded SQL	177
5.2 Application enablement planning	180
5.3 Converting XML queries	182
5.3.1 SQL/XML	183
5.3.2 XQuery	186
5.3.3 Updates and deletes	189
5.4 Converting Oracle Pro*C applications to DB2	191
5.4.1 Connecting to the database	191
5.4.2 Host variable declaration	192
5.4.3 Exception handling	195
5.4.4 Error messages and warnings	196
5.4.5 Passing data to a stored procedure from a C program	197
5.4.6 Building a C/C++ DB2 application	199
5.5 Converting Oracle Java applications to DB2	200
5.5.1 Java access methods to DB2	201
5.5.2 JDBC driver for DB2	201
5.5.3 JDBC driver declaration	202
5.5.4 Stored procedure calls	204
5.6 Converting Oracle Call Interface applications	209
5.6.1 DB2 OCI functions	209
5.7 Converting ODBC applications	210
5.7.1 Introduction to DB2 CLI	211
5.7.2 Setting up the DB2 CLI environment	211
5.8 Converting Perl applications	212
5.9 Converting PHP applications	217
5.10 Converting .NET applications	225
Appendix A. Terminology mapping	233
Appendix B. Data types	237
B.1 Supported SQL data types in C/C++	238
B.2 Supported SQL data types in Java	242
B.3 Data types available in PL/SQL	244
B.4 Mapping Oracle data types to DB2 data types	246
Appendix C. Built-in packages	249
C.1 DBMS_OUTPUT	250
C.2 DBMS_ALERT	251

C.3 DBMS_PIPE	252
C.4 DBMS_JOB	255
C.5 DBMS_LOB	256
C.6 DBMS_SQL	257
C.7 DBMS_UTILITY	261
C.8 UTL_FILE	264
C.9 UTL_DIR	266
C.10 UTL_MAIL	267
C.11 UTL_SMTP	268
Appendix D. DB2 OCI sample program	273
Appendix E. Test cases	279
E.1 Oracle DDL	280
E.1.1 Tables and views	280
E.1.2 Packages, procedures, and functions	284
E.1.3 Triggers and anonymous blocks	297
E.2 DB2 DDL	299
E.2.1 Table and view	299
E.2.2 Package, procedure, and function	302
E.2.3 Trigger	314
Appendix F. Additional material	317
Locating the Web material	317
Using the Web material	318
System requirements for downloading the Web material	318
How to use the Web material	318
Related publications	319
IBM Redbooks	319
Other publications	319
Online resources	321
How to get Redbooks	322
Help from IBM	322
Index	323

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 give 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.

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.

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.

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.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at <http://www.ibm.com/legal/copytrade.shtml>

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

AIX 5L™	IBM®	pureXML®
AIX®	Informix®	Redbooks®
AS/400®	InfoSphere™	Redbooks (logo)  ®
DB2 Connect™	iSeries®	RS/6000®
DB2®	Optim™	System z®
developerWorks®	OS/390®	Tivoli®
DRDA®	Passport Advantage®	WebSphere®
Express™	POWER®	z/OS®

The following terms are trademarks of other companies:

Snapshot, and the NetApp logo are trademarks or registered trademarks of NetApp, Inc. in the U.S. and other countries.

SUSE, the Novell logo, and the N logo are registered trademarks of Novell, Inc. in the United States and other countries.

Oracle, JD Edwards, PeopleSoft, Siebel, and TopLink are registered trademarks of Oracle Corporation and/or its affiliates.

Red Hat, and the Shadowman logo are trademarks or registered trademarks of Red Hat, Inc. in the U.S. and other countries.

SAP, and SAP logos are trademarks or registered trademarks of SAP AG in Germany and in several other countries.

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

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

Intel, Intel logo, Intel Inside logo, and Intel Centrino logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

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

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

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

Preface

In this IBM® Redbooks® publication, we describe the new DB2 Oracle Database compatibility features. The latest version of DB2® includes extensive native support for the PL/SQL procedural language, new data types, scalar functions, improved concurrency, built-in packages, OCI, SQL*Plus, and more. These features dramatically improve the ease of developing applications that run on both DB2 and Oracle, and simplify the process of moving from Oracle to DB2.

In addition, IBM now provides rich tooling to simplify the enablement process, such as the highly scalable IBM Data Movement Tool for moving schema and data into DB2, and an Editor and Profiler for PL/SQL provided by way of the Optim™ Data Studio tool suite.

This Oracle to DB2 migration guide describes new technology, best practices for moving to DB2, and how to handle some common scenarios. It is now dramatically easier than ever to move to DB2 and experience the benefits of the IBM industry leading database, which offers a lower cost of ownership, ease of use, storage savings, industry leading performance, and superior customer support.

Within six weeks of DB2 9.7 becoming generally available, scores of companies had already reported very high compatibility between the code they developed for Oracle and what they were experiencing with DB2 9.7. These companies reported out-of-the-box compatibility rates between 90%-100% (measured by the percentage of statements that required modification), with typical compatibility in the high 90s. These statistics come from dozens of early adopters who shared their code and their experiences with IBM.

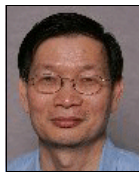
DB2's compatibility with Oracle has been provided through native support. This means the new capabilities in DB2 that provide compatibility have been implemented at the lowest and most intimate levels of the database kernel, as though they were originally engineered for DB2. Native support means that DB2's implementation is done without the aid of an emulation layer. This intimacy leads to the scalable implementation that DB2 offers, providing virtually identical performance between DB2's compatibility features and DB2's other language elements. For example, DB2 executes SQL PL at virtually the same performance as PL/SQL implementations of the same function.

This book describes the compatibility features as well as how these features and a handful of easy to use tools can be used to migrate from Oracle to DB2 with dramatically less effort than ever before. Chapter 1, “Introduction” on page 1 provides an overview of the features. Chapter 2, “Language compatibility features” on page 21 reviews the language compatibility elements. Chapter 3, “Enablement tools” on page 143 summarizes the major tools that help move schema and data, and provide debugging and editing capabilities around PL/SQL. Chapter 4, “Enablement scenario” on page 157 walks you through a migration scenario showing how all the components and tools come together. Chapter 5, “Application conversion” on page 173 concludes the body of the book with a review of application side compatibility topics, including JDBC and OCI.

The team who wrote this book

This book was produced by a team of specialists from around the world working at the International Technical Support Organization, San Jose Center.

Whei-Jen Chen is a Project Leader at the International Technical Support Organization, San Jose Center. She has extensive experience in application development, database design and modeling, and DB2 system administration. Whei-Jen is an IBM Certified Solutions Expert in Database Administration and Application Development, as well as an IBM Certified IT Specialist.



Kenneth Chen is an Executive IT Specialist with Information Management Partner technologies based at the IBM DB2 Toronto Lab. He is recognized for his technical expertise and experiences across the Information Management portfolio products. His leading role in application integration and enablement from competitive databases with IBM Business Partners and clients worldwide has contributed to many success stories in the field. In addition, he collaborates closely with DB2 development on external requirements to continue responding to the database application development community.

Ken’s current focus is to help drive DB2 successful adoption in the Asia Pacific region.



Patrick Dantressangle is a Senior Software Engineer for DB2, based at the IBM Hursley Lab in UK. His work focuses on all aspects of migrations from other RDBMSs to DB2 in EMEA. Patrick works closely with customers and Business Partners, assisting them in the design, implementation, and optimization of RDBMS solutions. He is also involved in emerging technologies as part of the local UK team, investigating new ways of “doing things” with RDBMSs.



Marina Greenstein is an Executive IT Specialist with the IBM Data Servers and Application Development Team. She is an IBM Certified Solutions Expert and has experience in database application architecture and development. She joined IBM in 1995, and in the past 14 years Marina has assisted customers with their migrations from Microsoft® SQL Server, Sybase, and Oracle databases to DB2.

She has presented migration methodology at numerous DB2 technical conferences and at SHARE. She is also the author of multiple articles and tutorials about DB2 application development.



Sam Lightstone is a Program Director and Senior Technical Staff Member with the IBM DB2 for Linux®, UNIX®, and Windows® development team, where he works on product strategy, architecture, and design. His recent work includes numerous topics in database language interfaces, data warehousing, autonomic computing, and relational database management systems. Sam is

an IBM Master Inventor with over 30 patents and patents pending; he has published widely on self-managing database systems and is co-author of five books. Sam has been with IBM since 1991. He has a Master in Mathematics in Computer Science from the University of Waterloo.



Maksym Petrenko is a DB2 Enablement Specialist based at IBM Toronto Lab. Currently, he assists companies who have quickly moved their applications to latest DB2 version. Maksym joined IBM Toronto Lab in 2001 and has worked as a developer, technical support analyst, and lab services consultant. His experience includes supporting clients with installation, configuration,

application development, and performance issues related to DB2 databases on Linux, UNIX, and Windows platforms. Maksym is a certified DB2 Advanced Database Administrator and DB2 Application Developer.



Raanon Reutlinger joined IBM in 1995 after nearly 10 years in the application development field, working with a variety of databases. As an IBM Technical Sales Specialist and an expert in DB2 for Linux, UNIX, and Windows, he has worked with a wide range of customers implementing a variety of solutions, including DB2 on SAP, data warehousing and replication of over 1000 servers, as well as

Federation Server. Raanon delivers DB2 courses and has experience with benchmarks and performance tuning, and was one of the authors of *DB2 UDB V7.1 Performance Tuning Guide*, SG24-6012. In January 2009, Raanon joined the IBM Toronto Lab, working out of Israel, as a DB2 Enablement Specialist, to work with customers on the early adoption of DB2 technologies.



Serge Rielau has worked with the DB2 for Linux, UNIX, and Windows SQL compiler for 12 years. He has a Master degree in Computer Science from the University of Kaiserslautern, Germany and participated in a year long internship at the IBM Almaden Research Lab. Presently, he is the technical architect responsible for SQL compatibility on DB2 9.7. As an expert in the SQL language, Serge is an active participant in the comp.databases.ibm-db2 newsgroup.



Steve Schormann has been with IBM in various positions for over 29 years. For the past 17 years, Steve has been working in the DB2 development group in Toronto, with much of that time spent on performance engineering working with the performance group in IBM DB2 to achieve some world record performance benchmarks. For the past 1 1/2 years, Steve has been involved in database compatibility efforts within DB2, developing tools and assisting ISVs and customers in their migrations to DB2.



Maria N Schwenger joined IBM in 2005 as a member of the Entity Analytic Solutions team, bringing in more than 10 years experiences in performance engineering, database architecture, administration, and development on Oracle and Microsoft SQL server, as well as extensive experiences in migration from older systems to relational databases. Currently, Maria is a part of the DB2 Open Database Technology team in IBM Canada, where she works with the “high touch” model with early release participants to promote DB2 technology early adoption, gathers feedback and references on pre-release products, and leads early enablement of applications written for competitive databases to DB2 9.7.

Acknowledgement

Thanks to the following people for their contributions to this project:

Berni Schiefer
Gustavo Arocena
Kelvin Ho
Bob Sawyer
Bill McLaren
Kathy McKnight
Richard Swagerman
IBM Toronto Laboratory, Canada

Shawn D. Moe
Igor Siljanovski
Diem H. Mai
Tuong C. Truong
IBM Silicon Valley Laboratory, USA

Vikram S Khatri
IBM Sales and Distribution, USA

Faiz Husain
EnterpriseDB Corporation

An Na Choi
Marina Greenstein
Scott J Martin
Fraser McArthur
Carlos Eduardo Abramo Pinto
Arthur V Sammartino
Nora Sokolof
Authors of *Oracle to DB2 Conversion Guide for Linux, UNIX, and Windows*,
SG24-7048

Emma Jacob
International Technical Support Organization, San Jose Center

Become a published author

Join us for a two- to six-week residency program! Help write a book dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You will have the opportunity to team with IBM technical professionals, Business Partners, and Clients.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you will develop a network of contacts in IBM development labs, and increase your productivity and marketability.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our books to be as helpful as possible. Send us your comments about this book or other IBM Redbooks publications in one of the following ways:

- ▶ Use the online **Contact us** review Redbooks form found at:

ibm.com/redbooks

- ▶ Send your comments in an e-mail to:

redbooks@us.ibm.com

- ▶ Mail your comments to:

IBM Corporation, International Technical Support Organization
Dept. HYTD Mail Station P099
2455 South Road
Poughkeepsie, NY 12601-5400



Introduction

IBM DB2 9.7 for Linux, UNIX, and Windows has out-of-the-box support for Oracle's SQL and PL/SQL dialects. This allows many applications written against Oracle to execute against DB2 with minimal or no changes.

In this book, references to “DB2 UDB” or “DB2” should be interpreted as “DB2 Universal Database Version 9.7 for Linux, and UNIX and Windows.”

This chapter includes the following topics:

- ▶ DB2 family of products
- ▶ DB2 Oracle Database compatibility features overview
 - Concurrency control
 - Data types
 - Implicit casting
 - SQL Dialect
 - PL/SQL
 - Built-in packages
 - Oracle specific JDBC extensions
 - SQL*Plus scripts

- ▶ DB2 features

DB2 added features, autonomic features, compression, and scan sharing

1.1 DB2 family of products

In the era of *Information On Demand*, IBM Information Management software offers a wide range of DB2 database products to accommodate different business needs and technical requirements in order to provide customers with a robust and scalable enterprise wide solution. Here we introduce the DB2 servers editions for Linux, UNIX, and Windows and some of the DB2 features.

1.1.1 DB2 editions

DB2 offers database solutions that run on all platforms, including Microsoft Windows, AIX®, Solaris, HP-UX, Linux, AS/400®, OS/390®, and z/OS®. Furthermore, DB2 technologies support both 32-bit and 64-bit environments, providing support for 32-bit operating systems on Linux on x86 and Windows, and 64-bit operating systems on Linux, UNIX, and Windows. The DB2 product family comprises a variety of packages that provide customers with choices based on business need. Here we discuss the edition of DB2 9.7 for Linux, UNIX, and Windows.

Enterprise Server Edition

DB2 Enterprise is designed to meet the data server needs of mid-size to large-size businesses. It can be deployed on Linux, UNIX, or Windows servers of any size, from one processor to hundreds of processors, and from physical to virtual servers. DB2 Enterprise is an ideal foundation for building on demand enterprise-wide solutions, such as high-performing 24 x 7 available high-volume transaction processing business solutions or Web-based solutions. It is the data server back end of choice for industry-leading ISVs building enterprise solutions, such as business intelligence, content management, e-commerce, Enterprise Resource Planning, Customer Relationship Management, or Supply Chain Management. Additionally, DB2 Enterprise offers connectivity, compatibility, and integration with other enterprise DB2 and IDS data sources.

Program charges

DB2 Enterprise includes Table Partitioning, High-Availability Disaster Recovery (HADR), Online Reorganization, Materialized Query Tables (MQTs), Multidimensional Clustering (MDC), Query Parallelism, Connection Concentrator, the Governor, pureXML®, backup compression, and Tivoli® System Automation for Multiplatforms (SA MP). The product also includes DB2 Homogeneous Federation and Homogeneous SQL Replication allowing federated data access and replication between DB2 servers as well as Web services federation. DB2 Enterprise is available on either a Processor Value Unit or per Authorized User pricing model. You must acquire a separate user license

for each Authorized User of this product with a minimum purchase of 25 users per 100 Processor Value Units.

Workgroup Server Edition

DB2 Workgroup is the data server of choice for deployment in a departmental, workgroup, or medium-sized business environment. It is offered in per Authorized User, Value Unit, or limited use socket pricing models to provide an attractive price point for medium-size installations while providing a full-function data server.

Program charges

DB2 Workgroup includes High-Availability Disaster Recovery (HADR), Online Reorganization, pureXML, Web services federation support, backup compression, and Tivoli System Automation for Multiplatforms (SA MP). DB2 Workgroup can be deployed in Linux, UNIX, and Windows server environments and will use up to 16 cores and 16 GB of memory. DB2 Workgroup is restricted to a stand-alone physical server with a specified maximum number of Processor Value Units based on the total number and type of processor cores, as determined in accordance with the *IBM Express Middleware Licensing Guide*, available at:

<ftp://ftp.software.ibm.com/software/smb/pdfs/LicensingGuide.pdf>

If licensed using per Limited Use Socket licensing, you can deploy on servers with a maximum of four sockets. You must acquire a separate user license for each authorized user of this product, with a minimum purchase of five users per server.

Express Edition

DB2 Express™ is a full-function DB2 data server, which provides very attractive entry-level pricing for the small and medium business (SMB) market. It is offered in per Authorized User, Value Unit, or Limited Use Virtual Server based pricing models to provide choices to match SMB customer needs. It comes with simplified packaging and is easy to transparently install within an application. DB2 Express can also be licensed on a yearly fixed term Limited Use Virtual Server license. While it is easy to upgrade to the other editions of DB2 9.7, DB2 Express includes the same autonomic manageability features of the more scalable editions. You never have to change your application code to upgrade; simply install the license certificate to upgrade.

Program charges

DB2 Express can be deployed on pervasive SMB operating systems, such as Linux, Windows, or Solaris, and includes pureXML, Web services federation, and backup compression. If licensed as a yearly subscription (DB2 Express FTL), it also includes a High Availability feature as long as both primary and secondary servers in the high availability cluster are licensed. If licensed using processor value units (PVUs), the limit is 200 PVUs. If licensed under the Limited Use Virtual Server metric, DB2 Express will use up to four cores on the server. The DB2 data server cannot use more than 4 GB of memory per server. You must acquire a separate user license for each authorized user of this product with a minimum purchase of five users per server.

Express-C

DB2 Express-C is a free, entry-level edition of the DB2 data server for the developer and IBM Business Partner community. It is designed to be up and running in minutes, is easy-to-use and embed, includes self-management features, and embodies all of the core capabilities of DB2 for Linux, UNIX, and Windows, such as pureXML. Solutions developed using DB2 Express-C can be seamlessly deployed using more scalable DB2 editions without modifications to the application code.

Program charges

DB2 Express-C can be used for development and deployment at no charge, and can also be distributed with third-party solutions without any royalties to IBM. It can be installed on physical or virtual systems with any amount of CPU and RAM, and is optimized to utilize up to a maximum of two processor cores and 2 GB of memory. DB2 Express-C is refreshed at major release milestones and comes with online community-based assistance. Users requiring more formal support, access to Fix Packs, or additional capabilities, such as high availability clustering and replication features, can purchase an optional yearly subscription for DB2 Express (FTL) or upgrade to other DB2 editions.

Personal Edition

DB2 Personal Edition is a single-user, full-function relational database, with built-in replication, ideal for desktop or mobile computer-based deployments. DB2 Personal can be remotely managed, making it the perfect choice for deployment in occasionally connected or remote office implementations that do not require multiuser capability.

Program charges

DB2 Personal Edition can be deployed on Linux or Windows. You must acquire a separate user license for each Authorized User of this product.

Developer Edition

This edition offers a package for a single application developer to design, build, and prototype applications for deployment on any of the IBM Information Management client or server platforms. This comprehensive developer offering includes DB2 Workgroup, DB2 Enterprise, IDS Enterprise Edition, DB2 Connect™ Unlimited Edition for System z®, and all the DB2 9.7 features, allowing you to build solutions that utilize the latest data server technologies.

Program charges

The software in this package cannot be used for production systems. You must acquire a separate user license for each Authorized User of this product.

1.1.2 DB2 value added features

While DB2 9 includes capabilities that serve the needs of most deployments, additional capabilities are required for certain application types, workloads, or environments that are not required by every deployment. Rather than build a one-size-fits-all offering, IBM makes these capabilities available as optional features to give you the flexibility to purchase only what you need. Here we introduce the available value-added features.

IBM DB2 9.7 Enterprise Edition priced features

The following features are available for DB2 9.7 Enterprise Edition.

IBM DB2 Storage Optimization Feature

This feature includes data row compression and other compression types to help maximize the use of existing storage:

- ▶ *Data row compression* uses a static dictionary-based compression algorithm to compress data by row. Compressing data at the row level is advantageous because it allows repeating patterns that span multiple column values within a row to be replaced with shorter symbol strings, allowing for improved performance. In I/O-bound systems, it will not only reduce storage requirements but may improve overall performance.
- ▶ *Index compression* utilizes multiple algorithms to reduce the storage requirements for indexes and reduces the total storage footprint of your DB2 database.
- ▶ *Temp compression* automatically compresses the amount of space used for temp tables. This automated compression will help to both improve performance of business intelligence workloads using large temp tables as well as reducing the total storage needed by DB2.

Program charges

This feature is available as an option for DB2 Enterprise only and can only be acquired if the underlying DB2 data server is licensed with the Processor Value Unit charge metric.

IBM DB2 Advanced Access Control Feature

This feature increases the control you have over who can access your data. This feature delivers Label Based Access Control (LBAC) and lets you decide exactly who has write access and who has read access to individual rows and individual columns.

Program charges

This feature is available as an option for DB2 Enterprise only. You must acquire this feature under the same charge metric as the underlying DB2 data server.

IBM DB2 Performance Optimization Feature

DB2 Performance Optimization leverages DB2 Workload Management, DB2 Performance Expert, and DB2 Query Patroller components to bring you the best in performance tuning and monitoring. The Performance Optimization Feature contains the following components:

- ▶ *DB2 Workload Management (WLM)* lets you divide your work into classes and tailor your data server to support a variety of users and applications on the same system by prioritizing those classes.
- ▶ *DB2 Performance Expert (PE)* is a performance analysis and tuning tool for managing a heterogeneous mix of DB2 systems with a single user interface. DB2 PE simplifies DB2 performance management by providing you with the capability to monitor applications, system statistics, and system parameters using a single tool.

IBM DB2 Geodetic Data Management Feature

This feature provides the ability to store, access, manage, or analyze location-based round earth information for weather, defense, intelligence, or natural resource applications for commercial or government use. It provides the ability to manage and analyze spatial information with accuracies in distance and area by treating the earth as a continuous spherical coordinate system.

Program charges

This feature is available as an option for DB2 Enterprise only. You must acquire this feature under the same charge metric as the underlying DB2 data server.

IBM Homogeneous Replication Feature

This feature is available as an option for DB2 Enterprise edition. The homogeneous replication feature allows you to use Q replication to copy data at high speed and low latency to multiple DB2 or IDS remote databases. The homogeneous replication feature is part of a larger IBM solution for replicating data in small and large enterprises.

Program charges

This feature is available as an option for DB2 Enterprise Server Edition. It can only be acquired if the underlying DB2 data server is licensed with the Processor Value Unit charge metric.

IBM DB2 9.7 Express Edition priced features

The following features apply to DB2 9.7 Express Edition.

IBM DB2 High Availability Feature

The DB2 High Availability Feature provides 24 x 7 continuous availability for your DB2 data server. The High Availability Feature contains the following components:

- ▶ *Highly Available Disaster Recovery (HADR)* allows failover to a standby system in the event of a software or hardware failure on the primary system. HADR is included in the DB2 Enterprise and DB2 Workgroup.
- ▶ *Online reorganization* reconstructs the rows in a table to eliminate fragmented data and compact information for better performance while permitting uninterrupted access to the table data. Online reorganization is included in DB2 Enterprise and DB2 Workgroup.
- ▶ *IBM Tivoli System Automation for Multiplatforms (SA MP)* provides high availability by automating the control of IT resources, such as processes, file systems, IP addresses, and other resources. It can coordinate the automatic failover to a standby DB2 data server using HADR. SA MP is included in DB2 Enterprise and DB2 Workgroup.

Program charges

This feature is available as an option for DB2 Express. You must acquire this feature under the same charge metric as the underlying DB2 data server. DB2 Express FTL includes this feature at no charge provided both servers in the high availability cluster are licensed under the DB2 Express Fixed Term License.

1.1.3 DB2 9 autonomic computing features

The DB2 autonomic computing environment is self-configuring, self-healing, self-optimizing, and self-protecting. By sensing and responding to situations that

occur, autonomic computing shifts the burden of managing a computing environment from database administrators to technology.

The features are discussed in the following list:

- ▶ Automatic features

Automatic features assist you in managing your database system. They allow your system to perform self-diagnosis and to anticipate problems before they happen by analyzing real-time data against historical problem data. You can configure some of the automatic tools to make changes to your system without intervention to avoid service disruptions.

- ▶ Self tuning memory

The DB2 memory-tuning feature simplifies the task of memory configuration by automatically setting values for several memory configuration parameters. When enabled, the memory tuner dynamically distributes available memory resources among the following memory consumers: buffer pools, locking memory, package cache, and sort memory.

- ▶ Configuring memory and memory heaps

With the simplified memory configuration feature, you can configure memory and memory heaps required by the DB2 data server by using the default AUTOMATIC setting for most memory-related configuration parameters, thus requiring much less tuning.

- ▶ Automatic storage

Automatic storage simplifies storage management for table spaces. When you create an automatic storage database, you specify the storage paths where the database manager will place your data. Then, the database manager will manage the container and space allocation for the table spaces as you create and populate them.

- ▶ Automatic compression dictionary creation

A compression dictionary is used to compress data moved into a table to free up space so that more data can be added in the table. A compression dictionary is automatically created and inserted or appended to a table during a data population operation, such as a load or an insert.

- ▶ Automatic maintenance

The database manager provides automatic maintenance capabilities for performing database backups, keeping statistics current, and reorganizing tables and indexes as necessary. Performing maintenance activities on your databases is essential in ensuring that they are optimized for performance and recoverability.

- ▶ Configuration advisor

You can use the Configuration Advisor to obtain recommendations for the initial values of the buffer pool size, database configuration parameters, and database manager configuration parameters.

- ▶ Design advisor

The DB2 Design Advisor is a tool that can help you significantly improve your workload performance. The DB2 Design Advisor provides recommendations about selecting indexes and other physical database structures, such as materialized query tables (MQTs), multidimensional clustering tables (MDC), and database partitioning features (used with DPF). The Design Advisor identifies all of the objects that are needed to improve the performance of your workload.

- ▶ Utility throttling

Utility throttling regulates the performance impact of maintenance utilities so that they can run concurrently during production periods. Although the impact policy, a setting that allows utilities to run in throttled mode, is defined by default, you must set the impact priority, a setting that each cleaner has indicating its throttling priority when you run a utility (if you want to throttle it).

1.2 DB2 Oracle Database compatibility features overview

To allow an application written for one RDBMS to run on another RDBMS virtually unchanged, many pieces have to fall into place. Different locking mechanisms, data types, SQL, procedural language residing on the server, and even the client interfaces used by the application itself need to be aligned not only in syntax, but also in semantics.

All these steps have been taken in DB2, so changes to an application are an exception and not the rule.

Note: PL/SQL, the built-in package library, and CLPPlus are presently not available for DB2 Express-C and DB2 Personal Edition.

1.2.1 Concurrency control

Traditionally cursor stability (CS) has been implemented so that writers block readers and, in some cases, readers can block writers. The reason for this is that, traditionally, a transaction under CS isolation will “wait for the outcome” of a pending concurrent transaction's changes.

There is no semantic reason why a transaction running under CS isolation should wait for an outcome when it encounters a changed row. An equally satisfactory behavior is to read the currently committed version of the changed row.

This behavior has been implemented in DB2 9.7 so that DB2 simply retrieves the currently committed version of a locked row from the log. In most common cases, the row is still in the log buffer because the change has not been committed yet. If the row has been written out and has also been overwritten in the log buffer, DB2 knows exactly where to find it, so that a single I/O will retrieve the desired version.

Figure 1-1 shows an application updating a name in an employee table. Before that application has committed the change, another application scans that table. Traditionally, the second user would have had to wait for the first application to commit or rollback. Due to read currently committed data, the scan for the second application will simply retrieve the version of the row from the log buffer that does not contain the first user's changes.

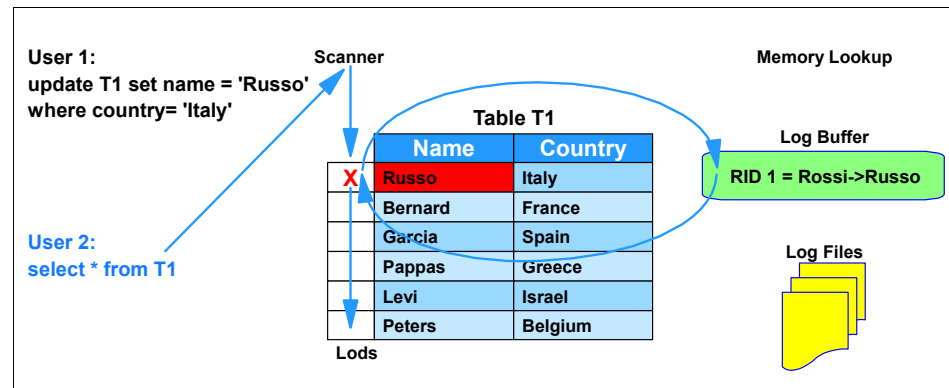


Figure 1-1 Concurrency control

This new behavior and its implementation introduces no new objects, such as a rollback segment, and has no performance impact on the writer, because the log needs to be written regardless.

This new behavior cannot cause a situation similar to “Snapshot™ too old” because in the extremely unlikely event that the log file needed has been archived, DB2 will simply fall back and wait for the lock to go away. It would be a very rare occurrence for this situation to occur, as it would require archival of a log file while a transaction was still uncommitted.

In addition to these changes, additional lock avoidance techniques have been introduced into DB2 to eliminate a reader holding a lock under CS isolation.

1.2.2 Data types

The heart of every database is its data. Mismatched types or mismatched semantics of these types can seriously impact the ability to enable an application to another RDBMS. In order to allow Oracle applications to run on DB2, it is crucial to support its nonstandard basic types, such as strings, dates, and numerics. Beyond aligning these basic types, there are other, more complex types that are commonly used in Oracle's PL/SQL that have been added in DB2 9.7. (Refer to B.3, “Data types available in PL/SQL” on page 244.)

1.2.3 Implicit casting

Implicit casting is the automatic conversion of data of one data type to another data type based on an implied set of conversion rules. If two objects have mismatched types, implicit casting is used to perform comparisons or assignments if a reasonable interpretation of the data types can be made.

In adherence with the SQL Standard and following a philosophy that a type mismatch is likely an indication of a coding mistake, DB2 has traditionally followed strong typing rules, where strings and numerics cannot be compared unless one is explicitly cast to the other.

Very often, Oracle applications use weak typing in their SQL. These applications have previously failed to compile against DB2. In DB2 9.7, implicit casting (or weak typing) has been added, that is, strings and numbers can be compared, assigned, and operated on in a very flexible fashion.

In addition, untyped NULLs can be used in many more places, while untyped parameter markers can be used nearly anywhere, thanks to deferred prepare. That is, DB2 will not resolve the type of a parameter marker until it has seen the first actual value.

DB2 also supports defaulting procedure parameters as well as the association of arguments to parameters by name.

Implicit casting is also supported during function resolution. When the data types of the arguments of a function being invoked cannot be promoted to the data types of the parameters of the selected function, the data types of the arguments are implicitly cast to the data types of the parameters.

1.2.4 SQL Standard

DB2 has a tradition of supporting the SQL Standard; in contrast, Oracle has implemented many nonstandard keywords and semantics. DB2 9.7 now supports many of these keywords and semantics, for example:

- ▶ CONNECT BY recursion
- ▶ (+) join symbol
- ▶ DUAL table
- ▶ ROWNUM pseudo column
- ▶ ROWID pseudo column
- ▶ MINUS SQL operator
- ▶ SELECT INTO FOR UPDATE
- ▶ PUBLIC SYNONYM
- ▶ CREATE TEMPORARY TABLE
- ▶ TRUNCATE TABLE

1.2.5 PL/SQL

DB2 9.7 introduces native PL/SQL support. Figure 1-2 shows that the DB2 engine now includes a PL/SQL compiler along side the SQL PL compiler. Both compilers produce virtual machine code for DB2's SQL Unified Runtime Engine. It is important to note that monitoring and development tools such as Optim Development Studio are hooked into DB2 at the runtime engine level. DBAs and application programmers develop and debug their PL/SQL source.

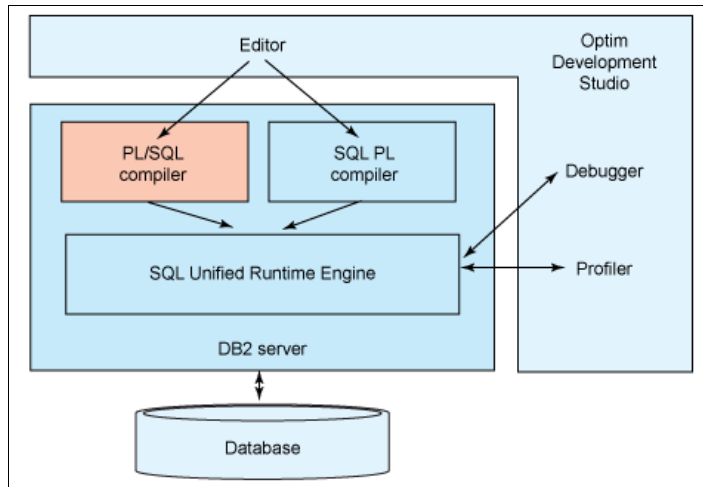


Figure 1-2 SQL compiler

The integration of PL/SQL into DB2 as a first class procedural language has several implications:

- ▶ There is no translation. The source code remains as it is in the schema catalog.
- ▶ Developers can continue working in the language with which they are familiar. There is no need to translate logic to DB2's dialect even if new logic is written in SQL PL. Routines using different dialects can call each other.
- ▶ Packaged application vendors can use one source code against both Oracle and DB2.
- ▶ Both PL/SQL and SQL PL produce the same virtual machine code for DB2's SQL Unified Runtime Engine. Therefore, by design, both PL/SQL and SQL PL perform at the same speed.
- ▶ Because the debugger infrastructure hooks directly into the SQL Unified Runtime Engine, PL/SQL is naturally supported by Optim Development Studio.

PL/SQL syntax details

DB2 supports all the common constructs of PL/SQL, such as:

- ▶ If Then Else.
- ▶ While loops.
- ▶ := assignments.
- ▶ Local variables and constants.

- ▶ #PRAGMA EXCEPTION and exception handling.
- ▶ Various forms of for loops (range, cursor, and query).
- ▶ %TYPE and %ROWTYPE anchoring of variables and parameters to other objects.
- ▶ #PRAGMA AUTONOMOUS transactions, which allow procedures to execute in a private transaction.

PL/SQL object support

PL/SQL can be used in various different objects that allow procedural logic:

- ▶ Scalar functions
- ▶ Before each row triggers
- ▶ After each row triggers
- ▶ Procedures
- ▶ Anonymous blocks
- ▶ PL/SQL packages

PL/SQL package support

Most PL/SQL in Oracle applications is contained within *packages*. A PL/SQL package (not to be confused with a DB2 package) is a collection of individual objects with the ability to differentiate between externally accessible objects and those that are mere helpers for use within the package. The ANSI SQL equivalent of a package is a MODULE. DB2 now provides support for ANSI SQL modules as well as PL/SQL packages. In particular, the following capabilities are provided:

- ▶ CREATE [OR REPLACE] PACKAGE, which defines prototypes for externally visible routines. It also defines all externally visible, non-procedural objects, such as variables and types.
- ▶ CREATE [OR REPLACE] PACKAGE BODY, which implements all private and public routines as well as all other private objects.
- ▶ Within a package or package body, the following objects can be defined:
 - Variables and constants
 - Data types
 - Exceptions
 - Scalar functions
 - Procedures
 - Cursors
- ▶ Package initialization.
- ▶ Public synonyms on packages.

1.2.6 Built-in packages

Some Oracle applications utilize packages that are provided by the RDBMS. In particular, libraries that provide reporting, e-mail, or cross-connection communication can be popular. The following packages, available in DB2, facilitate enablement of these applications for DB2:

- ▶ DBMS_OUTPUT
- ▶ DBMS_SQL
- ▶ DBMS_ALERT
- ▶ DBMS_PIPE
- ▶ DBMS_JOB
- ▶ DBMS_LOB
- ▶ DBMS_UTILITY
- ▶ UTL_FILE
- ▶ UTL_MAIL
- ▶ UTL_SMTP

1.2.7 Oracle specific JDBC extensions

JDBC is a standard Java™ client interface. There are, however, extensions that have been added to Oracle's JDBC driver in order to support specific nonstandard data types.

To maximize the level of compatibility for Java technology-based applications, the DB2 9.7 JDBC driver provides, among other things, support for calling procedures with reference cursor and VARRAY parameters.

1.2.8 SQL*Plus scripts

Often times, DDL scripts and even reports are written using the SQL*Plus command-line processor. To make it easier to transfer these scripts as well as the skills of developers writing them, DB2 provides an SQL*Plus-compatible command-line processor, called *CLPPlus*. The tool provides the following functionality:

- ▶ SQL*Plus-compatible command options
- ▶ Variable substitution
- ▶ Column formatting
- ▶ Reporting functions
- ▶ Control variables

1.2.9 Compatibility

During the one year beta phase of DB2 9.7, many applications totaling over 750,000 lines of PL/SQL code were analyzed in detail with an average out-of-the-box transfer rates in the range of 90%-99%, as shown in Figure 1-3.

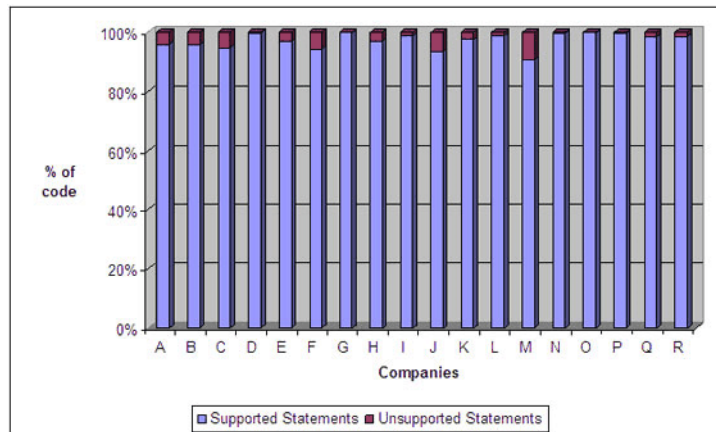


Figure 1-3 9. 98% average rate of supported statements

1.3 DB2 education resource

IBM has new offerings to support your training needs, enhance your skills, and boost your success with IBM software.

IBM offers a range of training options from traditional classroom to Instructor-Led Online (ILO) to meet your demanding schedule. ILO is an innovative learning format where students get the benefit of being in a classroom with the convenience and cost savings of online training.

Go Green with IBM on site training for groups as small as three or as large as fourteen. Choose from the same quality training delivered in classrooms, or customize a course or a selection of courses to best suit your business needs.

Enjoy further savings when you purchase training at a discount with an IBM Education Pack online account, which is a flexible and convenient way to pay, track, and manage your education expenses online. Check your local Information Management Training and Education Web site or with your training representative for the most recent training schedule

The following educational offerings are available:

Course title	Classroom Code	ILO Code
-----	-----	-----
DB2 Family Fundamentals	CE03	3E03
SQL Workshop	CE12	3E12
SQL Workshop for Experienced Users	CE13	3E13
Fast Path to DB2 9 for Experienced Relational DBAs	CL28	3L28
DB2 9 for LUW Quickstart for Experienced Relational DBAs	CL48	3L48
DB2 9 Database Administration Workshop for LUW	CL2X	3L2X
Query XML Data with DB2 9	CL12	3L12
Manage XML Data with DB2 9	CL14	3L14
Query and Manage XML Data with DB2 9	CL13	3L13
DB2 Performance Tuning and Monitoring	CL41	3L41
DB2 Advanced Database Administration for Experts	CL46	3L46
DB2 Advanced Database Recovery	CF49	3L49

Descriptions of courses for IT professionals and managers are available at:

http://www.ibm.com/services/learning/ites.wss/tp/en?pageType=tp_search

Visit <http://www.ibm.com/training> or call IBM training at 800-IBM-TEACH (426-8322) for scheduling and enrollment.

IBM professional certification

Information Management Professional Certification is a business solution for skilled IT professionals to demonstrate their expertise to the world. Certification validates skills and demonstrates proficiency with the most recent IBM technology and solutions. The following is a few IBM professional certification offerings:

- ▶ Exam 730, DB2 9 Family Fundamentals: IBM Certified Database Associate - DB2 9 Fundamentals
- ▶ Exam 541, DB2 9.7 DBA for Linux UNIX and Windows: IBM Certified Database Administrator - DB2 9.7 DBA for Linux UNIX and Windows
- ▶ Exam 543, DB2 9.7 Application Developer IBM Certified Application Developer - DB2 9.7
- ▶ Exam 735, DB2 SQL Procedure Developer

For additional information, go to:

<http://www.ibm.com/software/data/education/certification.html>

Other resources

Here are some additional DB2 educational resources:

- ▶ DB2 manuals can be found at:
<http://www.ibm.com/software/data/db2/udb/support/manualsv9.html>
- ▶ IBM Redbooks can be found at:
<http://ibm.com/redbooks>
- ▶ IBM Press books on DB2 and other products can be found at:
<http://www.redbooks.ibm.com/ibmpress/>
- ▶ The DB2 Express-C Edition is a free version of DB2 and can be downloaded from the following address:
http://www.ibm.com/developerworks/downloads/im/udbexp/?S_TACT=105AGX01&S_CMP=HP
- ▶ The IBM developerWorks® Web site offers white papers and other technical information about DB2 development:
<http://www.ibm.com/developerworks>

1.3.1 DB2 9.7 videos and articles

The following videos and articles provide introductions to some of the features available in DB2 9.7

Videos

The following videos are available from YouTube.

- ▶ Native PL/SQL support, found at:
<http://www.youtube.com/watch?v=EnpDMvobUmE>
- ▶ Moving to DB2 is easy, found at;
<http://www.youtube.com/watch?v=HJx3KZ5byN0>
- ▶ CLPPlus, found at;
<http://www.youtube.com/watch?v=3PndCKWlpJk>
- ▶ Online Schema change, found at;
<http://www.youtube.com/watch?v=vvM0x190XyE>

Articles

“Run Oracle applications on DB2 9.7 for Linux, UNIX, and Windows” is available from IBM developerWorks at:

<http://www.ibm.com/developerworks/data/library/techarticle/dm-0907oracleappsondb2/index.html>



Language compatibility features

DB2 for Linux, UNIX, and Windows Version 9.7 (DB2) introduces the new SQL and PL/SQL capabilities that facilitate database enablement from Oracle. DB2's approach to compatibility is to provide native support for the data types, scalar functions, packages and language elements, built-in packages, and the PL/SQL procedural language. Native support means that these interfaces are supported in the engine of the DB2 database server at the same level of integrity and efficiency as any other DB2 native language element. As a result, when you use these features, they perform with the natural speed and efficiency that the DB2 product offers.

In this chapter, we introduce the language features supported by DB2 along with examples of their usage. This chapter describes DB2's compatibility with Oracle through:

- ▶ SQL language extensions
- ▶ Routines, procedures, and functions
- ▶ Built-in packages
- ▶ Improved concurrency
- ▶ Schema compatibility
- ▶ SQL*Plus compatibility

We summarize the major enhancements that provide this significant compatibility, which makes it easier than ever to move to DB2. DB2's support for Oracle language dialects is excellent. There are rare cases where elements are not available in DB2. We show some examples of how these cases can be resolved with minimal or modest effort.

A comprehensive description of DB2's SQL and PL/SQL support is provided in the DB2 documentation, and is well documented in the literature available in stores and on the Web.

2.1 DB2 compatibility features references

DB2 Oracle Database compatibility features eliminates the need to convert most Oracle database objects and Oracle SQL to DB2. In this section, we show some examples of typical Oracle syntax constructs that are now also supported on DB2. We also discuss how to perform manual conversions for Oracle objects and features that are not natively supported on DB2. By way of examples, we illustrate techniques for handling exceptions in the database enablement process that utilize the power of both the PL/SQL and SQL PL languages. We describe these examples with the assumption that the reader already has some prior application development experience in either the Oracle PL/SQL or DB2 SQL PL languages. If no DDL is specified, the examples in this book are based on the tables and other database objects located in Appendix E, “Test cases” on page 279 or from the DB2 Sample database provided at DB2 installation time.

2.1.1 SQL compatibility setup

DB2 Oracle Database compatibility features eases the task of moving applications written for Oracle to DB2. The DB2_COMPATIBILITY_VECTOR registry variable is used to enable one or more DB2 compatibility features. This DB2 registry variable is represented as a hexadecimal value where each bit in the variable corresponds to one of the DB2 compatibility features. To take full advantage of the DB2 compatibility features, use the **db2set** command to set the value to ORA (the recommended setting). You can also selectively enable specific compatibility futures by setting specific bits of the DB2_COMPATIBILITY_VECTOR variable. Once established, we recommend keeping the selected compatibility level for the life of the database.

Table 2-1 presents the possible variable settings for the DB2_COMPATIBILITY_VECTOR registry variable.

Table 2-1 DB2_COMPATIBILITY_VECTOR values

Bit position	Compatibility feature	Description
1 (0x01)	ROWNUM	Enables the use of ROWNUM as a synonym for ROW_NUMBER() OVER(), and permits ROWNUM to appear in the WHERE clause of SQL statements.
2 (0x02)	DUAL	Resolves unqualified table references to “DUAL” as SYSIBM.DUAL.

Bit position	Compatibility feature	Description
3 (0x04)	Outer join operator	Enables support for the outer join operator (+).
4 (0x08)	Hierarchical queries	Enables support for hierarchical queries using the CONNECT BY clause.
5 (0x10)	NUMBER data type ^a	Enables the NUMBER data type and associated numeric processing.
6 (0x20)	VARCHAR2 data type ^a	Enables the VARCHAR2 data type and associated character string processing.
7 (0x40)	DATE data type ^a	Enables use of the DATE data type as TIMESTAMP(0), a combined date and time value.
8 (0x80)	TRUNCATE TABLE	Enables alternate semantics for the TRUNCATE statement, under which IMMEDIATE is an optional keyword that is assumed to be the default if not specified. An implicit commit operation is performed before the TRUNCATE statement executes if the TRUNCATE statement is not the first statement in the logical unit of work.
9 (0x100)	Character literals	Enables the assignment of the CHAR or GRAPHIC data type (instead of the VARCHAR or VARGRAPHIC data type) to character and graphic string constants whose byte length is less than or equal to 254.
10 (0x200)	Collection methods	Enables the use of methods to perform operations on arrays, such as first, last, next, and previous. Also enables the use of parentheses in place of square brackets in references to specific elements in an array; for example, array1(<i>i</i>) refers to element <i>i</i> of array1.
11 (0x400)	Data dictionary-compatible views ^a	Enables the creation of data dictionary-compatible views.
12 (0x800)	PL/SQL compilation ^b	Enables the compilation and execution of PL/SQL statements and language elements.

- a. Applicable only during database creation. Enabling or disabling this feature only affects subsequently created databases.
- b. See Restrictions on PL/SQL support at <http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/index.jsp?topic=/com.ibm.db2.luw.apdv.plsql.doc/doc/c0053608.html>.

You must set the DB2_COMPATIBILITY_VECTOR registry variable to the desired level before creating a new database. It is also essential to restart the DB2 instance after the value has been changed in order for it to take effect. Example 2-1 demonstrates the commands and the proper sequence of these steps.

Example 2-1 Setting DB2_COMPATIBILITY_VECTOR

```
db2inst1> db2set DB2_COMPATIBILITY_VECTOR=ORA
db2inst1> db2set -all
[i] DB2_COMPATIBILITY_VECTOR=ORA
[i] DB2COMM=tcPIP
[g] DB2INSTDEF=db2inst1
db2inst1> db2stop
SQL1064N  DB2STOP processing was successful.
db2inst1> db2start
SQL1064N  DB2STOP processing was successful
db2inst1> db2 "create database testdb PAGESIZE 32 K"
```

DB2_DEFERRED_PREPARE_SEMANTICS is another DB2 registry variable that enhances compatibility between Oracle and DB2 user applications, such as those written in Java. By setting this to YES, dynamic SQL statements will not be evaluated at the PREPARE step, but rather on OPEN or EXECUTE calls. This setting is important in order to take advantage of DB2's new implicit data type casting feature and avoids errors that might otherwise occur during the PREPARE step when untyped parameter markers are present. Example 2-2 demonstrates how to set this variable.

Example 2-2 Setting DB2_DEFERRED_PREPARE_SEMANTICS

```
db2set DB2_DEFERRED_PREPARE_SEMANTICS=YES
```

If you plan on using DBMS_JOB module/package, you might also want to activate the Administrative Task Scheduler (ATS). This facility is turned off by default, although you are still able to define and modify jobs (tasks). To enable the ATS, set the variable as shown below:

```
db2set DB2_ATS_ENABLE=YES
```

For more details, refer to the “Setting up the administrative task scheduler” topic in the Information Center, found at:

<http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/topic/com.ibm.db2.luw.admin.gui.doc/doc/t0054396.html>

Several database configuration parameters are also intended to simplify the enablement process. Among them are AUTO_REVAL and DECFLT_ROUNDING.

The AUTO_REVAL automatic revalidation parameter determines the behavior when invalid objects are encountered. The default value is DEFERRED, which means that if an object, such as a view or function, is invalidated for any reason (for example, dropping of underlined table), an attempt to revalidate it will automatically be attempted the next time it is referenced. Changing the AUTO_REVAL parameter value to DEFERRED_FORCE also allows new objects to be successfully created even though they may depend on invalid objects. The DEFERRED_FORCE value will also enable the CREATE value with an error feature. For example, a procedure will be successfully created even though it relies on a table that does not exist yet. The procedure will be marked as invalid until the table is created and the procedure is automatically revalidated on first use. This is especially convenient when creating large number of objects where it is difficult to execute the scripts in proper dependency order.

The DECFLT_ROUNDING database configuration parameter allows you to specify the rounding mode for a decimal floating point (DECFLOAT). This parameter defaults to round-half-even, but can be set to round-half-up to more closely match the Oracle rounding mode. Note that in order for the change to take effect, you must deactivate the database after updating of the database configuration, as shown in Example 2-3.

Example 2-3 Automatic revalidation and rounding mode setup

```
connect to testdb;  
db2 update db cfg using auto_reval DEFERRED_FORCE;  
db2 update db cfg using decflt_rounding ROUND_HALF_UP;  
db2 connect reset;  
db2 deactivate db testdb;  
db2 connect to testdb;
```

2.1.2 New PL/SQL and SQL types

DB2 provides extended support for SQL data types, such as NUMBER, VARCHAR2, and DATE, as well as some PL/SQL scalar types, including BOOLEAN, BINARY_INTEGER, PLS_INTEGER, RAW, and so on. This extended support ensures a simplified enablement process and running PL/SQL code in DB2 without code changes. A summary of the supported PL/SQL and SQL data types is available in Appendix B, “Data types” on page 237.

Record and collection types in PL/SQL

DB2 provides support for the most commonly used record types and most commonly used collection types.

Record types

Record types are supported in PL/SQL contexts when declared as part of PL/SQL packages (header or body). Examples of type implementation in PL/SQL are available in “Declaring a record type” on page 35. Alternatively, you can create same types constructs outside of a PL/SQL package and still reference them inside of routines and packages. See Example 2-75 on page 98 for an example.

Collection types

The use of collection types, such as VARRAY and associative arrays in PL/SQL, is supported by the DB2 data server. A PL/SQL collection is a set of ordered data elements with the same data type. Individual data items in the set can be referenced by using subscript notation within parentheses. When the database is in Oracle compatible mode, collection methods can be used to obtain information about collections or to modify them.

VARRAYS

DB2 supports VARRAY collection, a type of collection in which each element of a scalar data type is referenced by a positive integer called the array index. The maximum cardinality of the VARRAY (the maximum value of the array index) is defined in the type definition of the VARRAY, but cannot exceed 2147483647. In DB2, VARRAYs are one dimensional only. You cannot create multi-dimensional arrays.

The PL/SQL VARRAY syntax is as follow:

```
TYPE <varraytype> IS VARRAY( <max_index_value> ) OF <datatype>;
```

Table 2-2 summarizes the VARRAY collection methods that are supported by the DB2 data server in a PL/SQL context.

Table 2-2 VARRAY collection methods supported in PL/SQL

Collection method	Description
COUNT	Returns the number of elements in a collection.
DELETE	Removes all elements from a collection. You cannot delete individual elements from a VARRAY collection type.
EXISTS (n)	Returns TRUE if the specified element exists.
EXTEND	Appends a single NULL element to a collection (NO-OP).
EXTEND (n)	Appends n NULL elements to a collection (NO-OP).
EXTEND (n1, n2)	Appends n1 copies of the n2th element to a collection (NO-OP).
FIRST	Returns the smallest index number in a collection.
LAST	Returns the largest index number in a collection.
LIMIT	Returns the maximum number of elements for a VARRAY.
NEXT (n)	Returns the index number of the element immediately following the specified element.
PRIOR (n)	Returns the index number of the element immediately prior to the specified element.
TRIM	Removes a single element from the end of a collection.
TRIM (n)	Removes n elements from the end of a collection.

In DB2, VARRAYs should be defined either inside the PL/SQL packages or in stand-alone CREATE TYPE statements. Example 2-4 shows how to declare a VARRAY as part of a PL/SQL package and how to invoke this VARRAY and its collection methods in an anonymous block. As you see, this is the same familiar syntax used in Oracle. The example also demonstrates an easy way DB2 offers to do insert of multiple rows in a table.

Example 2-4 VARRAY usage in DB2

```
-- Example setup:
CREATE TABLE emp(ENAME VARCHAR2(10))/
INSERT INTO emp(ENAME) VALUES
('Mike' ), ('Peter'), ('Larry'), ('Joe'), ('Curly')/

CREATE PACKAGE Types_package
AS
```

```

        TYPE emp_arr_typ IS VARRAY(5) OF VARCHAR2(10);
END;
/

SET SERVEROUTPUT ON
/

DECLARE
    emp_arr          Types_package.emp_arr_typ;
    CURSOR emp_cur IS SELECT ename FROM emp
        WHERE ROWNUM <= 5;
    i                INTEGER := 0;
    k                INTEGER := 0;
    l                INTEGER := 0;
BEGIN

    FOR r_emp IN emp_cur LOOP
        i := i + 1;
        emp_arr(i) := r_emp.ename;
    END LOOP;

    -- Use FIRST/LAST to specify the lower/upper bounds of a loop range:
    FOR j IN emp_arr.FIRST..emp_arr.LAST LOOP
        DBMS_OUTPUT.PUT_LINE(emp_arr(j));
    END LOOP;

    -- Use NEXT(n) to obtain the subscript of the next element:
    k := emp_arr.FIRST;
    WHILE k IS NOT NULL LOOP
        DBMS_OUTPUT.PUT_LINE(emp_arr(k));
        k := emp_arr.NEXT(k);
    END LOOP;

    -- Use PRIOR(n) to obtain the subscript of the previous element:
    l := emp_arr.LAST;
    WHILE l IS NOT NULL LOOP
        DBMS_OUTPUT.PUT_LINE(emp_arr(l));
        l := emp_arr.PRIOR(l);
    END LOOP;

    DBMS_OUTPUT.PUT_LINE('COUNT: ' || emp_arr.COUNT);

    emp_arr.TRIM;
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || emp_arr.COUNT);

    emp_arr.TRIM(2);
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || emp_arr.COUNT);

    DBMS_OUTPUT.PUT_LINE('Max. No. elements = ' || emp_arr.LIMIT);

END;
/

```

It is also possible to use DB2 SQL PL syntax to declare an array and use it in a PL/SQL procedure in the same fashion as we would do with the VARRAYs. This is demonstrated in Example 2-5, where myVarrayType is defined as a stand-alone array type with DB2 syntax and used later in the TEST_ARRAY, a PL/SQL procedure.

Example 2-5 PL/SQL procedure with array type

```
CREATE TYPE myVarrayType as VARCHAR2(20) ARRAY[20];

CREATE OR REPLACE PROCEDURE test_array
IS
    my_arr myVarrayType;
BEGIN
    my_arr := myVarrayType('value1','value2','value3','value4');
    FOR j IN 1..5 LOOP
        DBMS_OUTPUT.PUT_LINE(my_arr(j));
    END LOOP;
    my_arr.trim(1);
END;
```

Associative arrays (index by tables)

DB2 provides support for associative arrays and the collection methods associated with them.

An associative array data type is a data type used to represent a generalized array with no predefined cardinality. Associative arrays contain an ordered set of zero or more elements of the same data type, where elements are ordered by and can be referenced by an index value. The index values of associative arrays are unique, they are of the same data type, and do not have to be contiguous.

The associative array data type supports the following associative array properties:

- ▶ No predefined cardinality is specified for associative arrays. This enables you to continue adding elements to the array without concern for a maximum size, which is useful if you do not know in advance how many elements will constitute a set.
- ▶ The array index value can be a non-integer data type. VARCHAR and INTEGER are supported index values for the associative array index.
- ▶ Index values do not have to be contiguous. In contrast to a conventional array, which is indexed by position, an associative array is an array that is indexed by values of another data type, and there are not necessarily index elements for all possible index values between the lowest and highest. This is useful if, for example, you want to create a set that stores names and phone numbers.

Pairs of data can be added to the set in any order and sorted using the data item in the pair defined as the index.

- ▶ The elements in an associative array are sorted in ascending order of index values. The insertion order of elements does not matter.
- ▶ Associative array data can be accessed and set using direct references or by using a set of available scalar functions.
- ▶ Associative arrays are supported in SQL PL contexts.
- ▶ Associative arrays can be used to manage and pass sets of values of the same kind in the form of a collection instead of having to:
 - Reduce the data to scalar values and use one-element-at-a-time processing, which can cause network traffic problems.
 - Use cursors passed as parameters.
 - Reduce the data to scalar values and reconstitute them as a set using a VALUES clause.

The PL/SQL associative array syntax is as follows:

```
TYPE <myArray> IS TABLE OF <myElementType> INDEX BY  
INTEGER|BINARY_INTEGER|PLS_INTEGER|VARCHAR2(size);
```

Table 2-3 summarizes the associative array collection methods supported by the DB2 data server in a PL/SQL context. Note that, in general, DB2 ARRAYs are not bounded and can increase dynamically, so EXTEND() is similar to a NO-OP.

Table 2-3 Associative arrays collection methods supported in PL/SQL

Collection method	Description
COUNT	Returns the number of elements in a collection.
DELETE	Removes all elements from a collection.
DELETE (n)	Removes element n from an associative array. You cannot trim elements from an associative array collection type.
DELETE (n1, n2)	Removes all elements from n1 to n2 from an associative array. You cannot trim elements from an associative array collection type.
EXISTS (n)	Returns TRUE if the specified element exists.
EXTEND	Appends a single NULL element to a collection (NO-OP).
EXTEND (n)	Appends n NULL elements to a collection (NO-OP).
EXTEND (n1, n2)	Appends n1 copies of the n2th element to a collection (NO-OP).
FIRST	Returns the smallest index number in a collection.
LAST	Returns the largest index number in a collection.
LIMIT	Returns the maximum number of elements for a VARRAY, or NULL for nested tables.
NEXT (n)	Returns the index number of the element immediately following the specified element.
PRIOR (n)	Returns the index number of the element immediately prior to the specified element.

The associative arrays in DB2 can be defined inside the PL/SQL packages or using SQL PL syntax as stand-alone types.

Example 2-6 on page 33 shows how to create, initialize, and display the values of an associative array. Note that the emp%ROWTYPE attribute is used to define emp_arr_typ.

Example 2-6 Associative array

```
CREATE OR REPLACE PACKAGE pkg_test_type
IS
    TYPE emp_arr_typ IS TABLE OF emp%ROWTYPE INDEX BY BINARY_INTEGER;
END pkg_test_type;
/

DECLARE
    emp_arr          pkg_test_type.emp_arr_typ;
    CURSOR emp_cur IS
        SELECT empno, ename
        FROM emp WHERE ROWNUM <= 10;
    i                INTEGER := 0;
BEGIN
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----');
    FOR r_emp IN emp_cur LOOP
        i := i + 1;
        emp_arr(i).empno := r_emp.empno;
        emp_arr(i).ename := r_emp.ename;
    END LOOP;
    FOR j IN 1..10 LOOP
        DBMS_OUTPUT.PUT_LINE(emp_arr(j).empno || '    ' ||
                               emp_arr(j).ename);
    END LOOP;
END;
/
```

%ROWTYPE attribute

DB2 also provides support for the %ROWTYPE attribute. In PL/SQL, the %ROWTYPE attribute is used to declare PL/SQL variables of type record with fields that correspond to the columns of a table or view. Each field in a PL/SQL record assumes the data type of the corresponding column in the table. The fields inside the records type are referred to by using dot notation, with the record name as a qualifier.

Example 2-7 demonstrates a declaration of a variable associated with the table T1 row type declared with the %ROWTYPE attribute. Note the calls to the record fields with a dotted notation of variable.table_field.

Example 2-7 Using %ROWTYPE attribute

```
CREATE TABLE t1 ( key INTEGER NOT NULL, col1 VARCHAR2(20))

insert into t1 values(1, 'A')
/

DECLARE
    myvar1 T1%ROWTYPE; -- myvar is a row of type similar to table t1
begin
    select * into myvar1 from t1 where key = 1;
    DBMS_OUTPUT.put_line(myvar1.col1 || ':' || myvar1.key); -- print the row
end;
/
```

Example 2-8 shows another example of the %ROWTYPE attribute.

Example 2-8 Using %ROWTYPE attribute

```
CREATE OR REPLACE PROCEDURE delete_employee(
    p_empno          IN employee.empno%TYPE
)
IS
    r_emp            employee%ROWTYPE;
BEGIN
    DELETE FROM employee WHERE empno = p_empno
    RETURNING
        *
    INTO
        r_emp;

    IF SQL%FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Deleted Employee # : ' || r_emp.empno);
        DBMS_OUTPUT.PUT_LINE('Name : ' || r_emp.LASTNAME);
        DBMS_OUTPUT.PUT_LINE('Job : ' || r_emp.job);
        DBMS_OUTPUT.PUT_LINE('Hire Date : ' || r_emp.hiredate);
        DBMS_OUTPUT.PUT_LINE('Salary : ' || r_emp.salary);
        DBMS_OUTPUT.PUT_LINE('Commission : ' || r_emp.comm);
        DBMS_OUTPUT.PUT_LINE('Department : ' || r_emp.WORKDEPT);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee # ' || p_empno || ' not found');
    END IF;
END;
/
```

Declaring a record type

PL/SQL record type declarations are supported by the DB2 data server in PL/SQL contexts. Globally, these types can be defined using SQL PL syntax.

A record type is a user definition of a record (row) that consists of one or more identifiers (fields), each with a corresponding data type. The following is the syntax of a user-defined record types:

```
TYPE <type_name> IS RECORD (field1 datatype, field2 datatype, ...);
```

This statement is supported in a PL/SQL context, only as a part of a package specification or package body. For a data type of the fields declared, you can use any valid SQL data type or %TYPE attribute.

A record variable (or record) is an instance of a record type. The properties of the record, such as its field names and types, are inherited from the record type. Dot notation is used to reference fields in a record, for example, record.field.

Example 2-9 shows a package specification that creates a user-defined record type. This type is immediately used as an IN OUT parameter in the procedure test_record_sp. One of the fields in the records is defined with the %TYPE attribute.

Example 2-9 Declaration and usage of a user-defined record type

```
CREATE OR REPLACE PACKAGE type_pkg
IS
TYPE t1_type IS RECORD (
    c1 T1.C1%TYPE,
    c2 VARCHAR(10)
);
PROCEDURE test_record_sp (
    p_testing_rec IN OUT t1_type,
    p_status      OUT    VARCHAR2
);
END;
/
```

%TYPE attribute

DB2 supports the %TYPE attribute, commonly used in PL/SQL for declaration of variables and parameters. Using this attribute ensures that the compatibility between table columns and PL/SQL variables are automatically maintained for us. If the data type of the column or the variable changes, there is no need to modify the declaration code.

The %TYPE attribute requires the column name to be prefixed by a qualifying table name in a dot notation. We could also assign %TYPE attribute with a name of a previously declared variable. The data type of this column or variable will be assigned to the variable being declared with the %TYPE attribute.

For demonstration purposes, we create a table called Test_table and a simple procedure named test_attribute1:

```
CREATE TABLE Test_table ( ID INTEGER NOT NULL, col1 VARCHAR2(20));
```

Example 2-10 shows that the %TYPE attribute can also be used with formal parameter declarations. Here v_variable1 is declared as a type of col1 of the Test_table utilizing the %TYPE attribute. The IN parameter p_id1 is also defined with %TYPE attribute and will have the same data type as ID column of the Test_table.

Example 2-10 Using %TYPE in parameter declaration

```
CREATE OR REPLACE PROCEDURE test_attribute1 (  
    p_id1          IN Test_table.id%TYPE  
)  
IS  
    v_variable1    Test_table.col1%TYPE := 'my_list';  
BEGIN  
    DBMS_OUTPUT.PUT_LINE('My ID is: ' || p_id1);  
    DBMS_OUTPUT.PUT_LINE('Col1 one is now: ' || v_variable1);  
END;  
/
```

This example is equivalent to Example 2-11, where simple data type declarations are used.

Example 2-11 Simple data type declaration

```
CREATE OR REPLACE PROCEDURE test_attribute2 (  
    p_id2          IN INTEGER  
)  
IS  
    v_variable2    VARCHAR2(20) := 'my_list';  
BEGIN  
    DBMS_OUTPUT.PUT_LINE('My ID is: ' || p_id2);  
    DBMS_OUTPUT.PUT_LINE('Col1 one is now: ' || v_variable2);  
END;  
/
```

Example 2-12 on page 37 shows another example of the %TYPE attribute.

Example 2-12 Using %TYPE attribute

```
CREATE OR REPLACE PROCEDURE emp_comp_update (  
    p_empno      IN employee.EMPNO%TYPE,  
    p_sal        IN employee.salary%TYPE,  
    p_comm       IN employee.comm%TYPE  
)  
IS  
    v_empno      employee.EMPNO%TYPE;  
    v_ename      employee.LASTNAME%TYPE;  
    v_job        employee.job%TYPE;  
    v_sal        employee.SALARY%TYPE;  
    v_comm       employee.comm%TYPE;  
  
BEGIN  
    UPDATE employee SET salary = p_sal, comm = p_comm WHERE empno = p_empno  
    RETURNING  
        EMPNO,  
        LASTNAME,  
        job,  
        SALARY,  
        comm  
    INTO  
        v_empno,  
        v_ename,  
        v_job,  
        v_sal,  
        v_comm;  
  
    IF SQL%FOUND THEN  
        DBMS_OUTPUT.PUT_LINE('Updated Employee # : ' || v_empno);  
        DBMS_OUTPUT.PUT_LINE('Name                : ' || v_ename);  
        DBMS_OUTPUT.PUT_LINE('Job                  : ' || v_job);  
        DBMS_OUTPUT.PUT_LINE('New Salary           : ' || v_sal);  
        DBMS_OUTPUT.PUT_LINE('New Commission       : ' || v_comm);  
    ELSE  
        DBMS_OUTPUT.PUT_LINE('Employee # ' || p_empno || ' not found');  
    END IF;  
END;  
/  
  
-- Calling procedure emp_comp_update:  
call emp_comp_update(000010, 6540, 1200)  
Return Status = 0  
Updated Employee # : 000010  
Name                : HAAS  
Job                  : PRES  
New Salary           : 6540  
New Commission       : 1200
```

2.1.3 Basic procedural statements

The programming statements that can be used in a PL/SQL application include the assignment statement and SQL statements, such as INSERT, UPDATE, DELETE, MERGE, SELECT INTO, NULL, and EXECUTE IMMEDIATE.

Assignment statement

The assignment statement sets a previously declared variable or formal parameter (OUT or IN OUT) to the value of an expression.

Example 2-13 shows the assignment syntax in several combinations, such as variables, parameters, constants, and so on.

Example 2-13 Assignment statement

```
CREATE OR REPLACE PROCEDURE dept_salary_rpt (  
    p_deptno      IN    NUMBER,  
    p_comm_rate   IN    NUMBER,  
    p_base_annual OUT    NUMBER  
)  
IS  
    todays_date    DATE;  
  
    -- CONSTANT assignment only in DECLARE section -----  
  
    rpt_title_base CONSTANT VARCHAR2(60) := 'Report For Department # '  
    rpt_title      VARCHAR2(60);  
    base_sal       INTEGER;  
    base_comm_rate NUMBER;  
BEGIN  
    todays_date := SYSDATE;  
  
    -- Expression assignment -----  
  
    rpt_title := rpt_title_base || ' ' || p_deptno || ' on '  
                || todays_date;  
    base_sal := 35525;  
    base_comm_rate := p_comm_rate;  
  
    -- Functional assignment -----  
  
    p_base_annual := ROUND(base_sal * base_comm_rate, 2);  
  
    DBMS_OUTPUT.PUT_LINE(rpt_title);  
    DBMS_OUTPUT.PUT_LINE('Base Annual Salary: ' || p_base_annual);  
END;  
/
```

Variable assignment

The variable assignment is the most common form of assignment statement in PL/SQL. You can assign a variable, a function returned value, a BOOLEAN value, an array element, an expression, a SQL result, and so on. For the CONSTANT variable, you must initialize or assign it in the DECLARE section. Expanding on Example 2-13 on page 38, we demonstrate the variable assignment in Example 2-14, where the cursor cur0 is assigned the current row.

Example 2-14 Variable assignment statement

```
DECLARE
    found BOOLEAN;
    var1 NAMEARRAY;
    cur0 SYS_REFCURSOR;
    n number :=0;
BEGIN
    FOR cur0 IN (SELECT name FROM sysibm.systables ORDER BY 1) LOOP
        n := n + 1;
        var1(n) := cur0.name;
        IF ( var1(n) = 'TEST' ) THEN
            found := TRUE;
            dbms_output.put_line('TEST table found at index entry = ' || N);
            EXIT;
        END IF;
    END LOOP;
END;
/
```

PL/SQL Blocks

PL/SQL block structures can be included within PL/SQL procedure, function, or trigger definitions or executed independently as an anonymous block. PL/SQL block structures and the anonymous block statement contain one or more of the following sections: an optional declaration section, a mandatory executable section, and an optional exception section. Each of these sections can include data type and variable declarations, SQL or PL/SQL statements, or other PL/SQL language elements.

Example 2-15 demonstrates the use of a PL/SQL block.

Example 2-15 PL/SQL Block

```
DECLARE
    var1 VARCHAR2(200);
BEGIN
    var1:= 'declare currenttime timestamp;
           BEGIN
               currenttime := sysdate;
               dbms_output.put_line(''The time now is ''||currenttime);
           END';
    EXECUTE IMMEDIATE var1;
END;
/
```

For more details about anonymous blocks, refer to “Anonymous blocks” on page 75.

SQL statements

SQL statements that are supported within PL/SQL contexts can be used to modify data or to specify the manner in which statements are to be executed. These SQL statements have the same usage and meaning in both PL/SQL and SQL PL.

Table 2-4 lists the SQL statements that can be executed by the DB2 server within PL/SQL contexts.

Table 2-4 SQL statements

Command	Description
DELETE	Deletes rows from a table.
INSERT	Inserts rows into a table.
MERGE	Inserts rows into a table.
SELECT INTO	Retrieves a single row from a table.
UPDATE	Updates rows in a table.

NULL statement

The NULL statement can act as a placeholder where an executable statement is required, but an SQL operation is not wanted. Example 2-16 on page 41 demonstrates a NULL statement used as a minimum executable statement that must exist between the BEGIN-END block; otherwise, there will be a syntax error.


```
BEGIN
    NULL;
END;
/
```

Sometimes, a NULL statement is used in the EXCEPTION section to indicate that the raised exception condition should be ignored, and processing should continue to next sequential block or statement.

EXECUTE IMMEDIATE statement

The EXECUTE IMMEDIATE statement prepares an executable form of an SQL statement from an SQL character string and then executes the SQL statement. EXECUTE IMMEDIATE combines the basic functions of the PREPARE and EXECUTE statements. In addition, EXECUTE IMMEDIATE can also execute an anonymous PL/SQL block. Since EXECUTE IMMEDIATE primarily is used for dynamic SQL execution, we cover this topic in 2.1.7, “Static and dynamic SQL support” on page 60.

RETURNING INTO clause

One very useful feature of SQL execution is the RETURNING INTO clause that can be optionally appended to the INSERT/DELETE/UPDATE statement that typically does row by row processing. This feature allows you to access the changed rows in an atomic manner, without making an additional selection, which may have locking implications, because the selection happens later, thereby making the SQL code more efficient. If there is not a changed row, then the RETURNING INTO values are undefined.

Example 2-17 shows how the RETURNING INTO clause is used with the INSERT, DELETE, and UPDATE statements that return one value. In this example, we also demonstrate an arithmetic operation with implicit casting in the VALUES clause of the SELECT statement.

Assume that you have a denormalized table EMP with three columns (empid, emplastname, and empfirstname). The example hypothetically cleans up (maintains) data to change the empid, lastname, and firstname of the EMP table.

Example 2-17 DELELET/INSERT/UPDATE with RETURNING INTO

```
CREATE OR REPLACE PROCEDURE update_emp(v_empid VARCHAR2,
                                         d_lastname OUT VARCHAR2, d_firstname OUT VARCHAR2)
IS
    d_empid VARCHAR2(6);
BEGIN
    -- Return names information about deleted rows

    DELETE FROM emp WHERE empid=v_empid
        RETURNING emplastname, empfirstname INTO d_lastname, d_firstname;

    -- Return empid after modification, as all the employee IDs will have the
    prefix '99'

    INSERT INTO emp(empid, emplastname, empfirstname)
        VALUES(990000+v_empid, d_lastname, d_firstname)
        RETURNING empid INTO d_empid;

    -- Return properly formatted names with INITCAP as part of data cleaning
    UPDATE emp SET emplastname = INITCAP(emplastname),
        empfirstname = INITCAP(empfirstname)
        WHERE empid=d_empid
        RETURNING emplastname, empfirstname INTO d_lastname, d_firstname;
END;
/
```

BULK COLLECT and FORALL

DB2 supports BULK COLLECT and FORALL syntax, including INDICES OF and VALUES OF clauses in FORALL.

BULK COLLECT and FORALL are PL/SQL statements that optimize processing and fetching of collection information from associative arrays and varrays.

Statement attributes

SQL%FOUND, SQL%NOTFOUND, and SQL%ROWCOUNT are PL/SQL attributes that can be used to determine the effect of an SQL statement:

► SQL%FOUND

This attribute has a Boolean value that returns TRUE if at least one row was affected by an INSERT, UPDATE, or DELETE statement, or if a SELECT INTO statement retrieved one row. Example 2-18 shows an anonymous block in which a row is inserted and a status message is displayed.

Example 2-18 Using SQL%FOUND

```
BEGIN
    INSERT INTO employee (empno, lastname, job, salary)
        VALUES (9001, 'JONES', 'CLERK', 850.00);
    IF SQL%FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Row has been inserted');
    END IF;
END;
/
```

► SQL%NOTFOUND

This attribute has a Boolean value that returns TRUE if no rows were affected by an INSERT, UPDATE, or DELETE statement. The value is also TRUE if no row is retrieved by a SELECT INTO statement, although it is more typical to check for When NO_DATA_FOUND exception rather than SQL%NOTFOUND in this case.

Example 2-19 shows how to use SQL%NOTFOUND.

Example 2-19 Using SQL%NOTFOUND

```
BEGIN
    UPDATE employee SET hiredate = '03-JUN-07' WHERE empno = 90000;
    IF SQL%NOTFOUND THEN
        DBMS_OUTPUT.PUT_LINE('No rows were updated');
    END IF;
END;
/
```

► **SQL%ROWCOUNT**

This attribute has an integer value that represents the number of rows that were affected by an INSERT, UPDATE, or DELETE statement. Example 2-20 illustrates how to use SQL%ROWCOUNT.

Example 2-20 Using SQL%ROWCOUNT

```
BEGIN
  UPDATE employee SET hiredate = '03-JUN-07' WHERE empno = 000010;
  DBMS_OUTPUT.PUT_LINE('# rows updated: ' || SQL%ROWCOUNT);
END;
/
```

2.1.4 Control of flow statements

The control of flow statements are programming constructions that allow you to group, relate, and organize different SQL or PL/SQL statements in internal procedural logic rather than relying on their sequential execution. In other words, these statements control the execution logic (“flow”) of the basic procedural statements, statement blocks, routines, and others. As such, they are a very important part of the programming concepts in PL/SQL and are supported by both Oracle and DB2.

Decision making statement: IF and CASE

You can use the IF statement within PL/SQL contexts to execute PL/SQL statements on the basis of certain criteria.

Example 2-21 shows a IF THEN ELSIF END IF routing statement.

Example 2-21 Routing statement

```
DECLARE
  a INTEGER := 1;
  b VARCHAR2(30) := 'default string';
BEGIN
  IF (a = 1) THEN
    B:='this is string';
  ELSIF (a = 2) THEN
    B:='this is another string';
  ELSE
    NULL;
  END IF;
END;
/
```

Similar to an IF statement, the CASE statements and expressions execute one or a set of statements when a specified search condition is true. CASE is a stand-alone statement that is distinct from the CASE expression, which must appear as part of an expression.

There are two forms of the CASE statements and expressions:

► Simple CASE

The simple CASE statement and expression attempt to match an expression (known as the selector) to another expression that is specified in one or more WHEN clauses. A match results in the execution of one or more corresponding statements.

Example 2-22 to Example 2-24 on page 46 present examples of using a simple CASE statement and expression in SQL and PL/SQL, where the assignments are based on matched values for a department code in the employee table.

Example 2-22 Simple CASE expression in SQL

```
SELECT LASTNAME, empno,  
       (CASE workdept  
         WHEN 'A00' THEN 'SPIFFY COMPUTER SERVICE DIV.'  
         WHEN 'B01' THEN 'PLANNING'  
         WHEN 'C01' THEN 'INFORMATION CENTER'  
         WHEN 'D01' THEN 'DEVELOPMENT CENTER'  
         WHEN 'E01' THEN 'SUPPORT SERVICES'  
         WHEN 'E11' THEN 'OPERATIONS'  
         ELSE 'Unknown'  
       END) current_department  
FROM employee  
ORDER BY empno;
```

Example 2-23 shows the CASE statement from Example 2-22 on page 45 in PL/SQL.

Example 2-23 CASE statement in PL/SQL

```
BEGIN
  FOR my_cursor IN (SELECT lastname, empno, workdept FROM employee ORDER BY
empno) LOOP
    DBMS_OUTPUT.PUT(my_cursor.lastname || ', ' || my_cursor.empno || '
belongs to: ');
    CASE my_cursor.workdept
      WHEN 'A00' THEN
        DBMS_OUTPUT.PUT_LINE('SPIFFY COMPUTER SERVICE DIV. ');
      WHEN 'B01' THEN
        DBMS_OUTPUT.PUT_LINE('PLANNING ');
      WHEN 'C01' THEN
        DBMS_OUTPUT.PUT_LINE('INFORMATION CENTER ');
      WHEN 'D01' THEN
        DBMS_OUTPUT.PUT_LINE('DEVELOPMENT CENTER ');
      WHEN 'E01' THEN
        DBMS_OUTPUT.PUT_LINE('SUPPORT SERVICES ');
      WHEN 'E11' THEN
        DBMS_OUTPUT.PUT_LINE('OPERATIONS ');
      ELSE
        DBMS_OUTPUT.PUT_LINE('Not IT related ');
    END CASE;
  END LOOP;
END;
/
```

Example 2-24 shows a simple CASE expression in PL/SQL.

Example 2-24 Simple CASE expression in PL/SQL

```
DECLARE
  workdept      VARCHAR2(3);
  current_dept  VARCHAR2(40);
BEGIN
  current_dept := CASE workdept
    WHEN 'A00' THEN 'SPIFFY COMPUTER SERVICE DIV.'
    WHEN 'B01' THEN 'PLANNING'
    WHEN 'C01' THEN 'INFORMATION CENTER'
    WHEN 'D01' THEN 'DEVELOPMENT CENTER'
    WHEN 'E01' THEN 'SUPPORT SERVICES'
    WHEN 'E11' THEN 'OPERATIONS'
    ELSE 'Not IT related'
  END;
  DBMS_OUTPUT.PUT_LINE(current_dept);
END;
```

/

► Searched CASE

A searched CASE statement uses one or more Boolean expressions to determine which statements to execute.

Example 2-25 presents an example of a searched CASE statement. The usage of the searched CASE expression in PL/SQL is similar to the simple CASE shown in Example 2-22 on page 45; only a comparison expression is used to find a match.

Example 2-25 Searched CASE

```
SELECT lastname, empno,  
       (CASE  
         WHEN salary < 30000 THEN 'Below Average'  
         WHEN salary BETWEEN 30000 AND 55000 THEN 'Average'  
         WHEN salary > 55000 THEN 'Above Average'  
         ELSE 'Need evaluation'  
       END) salary_review  
FROM employee  
ORDER BY empno;
```

Loops and iterative statements

To repeat a series of commands in PL/SQL code, DB2 supports loops and iterative statements, such as EXIT, FOR, LOOP, and WHILE statements.

The LOOP statement executes a sequence of one or more PL/SQL or SQL statements within a PL/SQL code block multiple times, and it could be part of a PL/SQL procedure, function, or anonymous block. These statements are executed during each iteration of the loop. The following lines show the syntax of a simple LOOP statement with the optional EXIT WHEN condition:

```
LOOP  
    <statements>  
    [< EXIT WHEN condition;>]  
END LOOP;
```

The EXIT statement terminates execution of a loop within a PL/SQL code block. It can be embedded within a FOR, LOOP, WHILE statement, or within a PL/SQL procedure, function, or anonymous block statement.

Example 2-26 shows basic LOOP and EXIT statements within an anonymous block.

Example 2-26 Classic LOOP statement

```
DECLARE
  a INTEGER := 0;
BEGIN
  a:= 1;
  LOOP
    DBMS_OUTPUT.PUT_LINE('this is string #' || a);
    EXIT WHEN a <= 10;
    a:=a+1;
  END LOOP;
END;
/
```

The WHILE statement repeats a set of SQL statements as long as a specified expression is true and it can be embedded within a PL/SQL procedure, function, or anonymous block statement. The condition is evaluated immediately before each entry into the loop body. The following lines are the syntax of the WHILE LOOP statement:

```
WHILE <condition> LOOP
  <statements>
  [< EXIT WHEN condition>]
END LOOP;
```

Example 2-27 shows a WHILE LOOP (statement).

Example 2-27 WHILE loop

```
DECLARE
  a INTEGER := 0;
BEGIN
  a:= 1;
  WHILE (a <= 10) LOOP
    DBMS_OUTPUT.PUT_LINE('this is string #' || a);
    a:=a+1;
  END LOOP;
END;
/
```

A FOR LOOP over a predetermined number of values (FOR with integer variant) can iterate over a range of values to execute a set of SQL statements more than once. With the REVERSE keyword, it decrements the variable over the range of value. The syntax is as follows:

```
FOR <loop_variable> IN [REVERSE] <range_of_values> LOOP
    <statements>
    [<EXIT WHEN condition>]
END LOOP;
```

Example 2-28 shows a simple FOR (integer variant) statement (or LOOP) that loops 10 times.

Example 2-28 FOR LOOP over predetermined number of values

```
DECLARE
    a INTEGER := 0;
BEGIN
    a:= 1;
    FOR a in 1..10 LOOP
        DBMS_OUTPUT.PUT_LINE('this is string #' || a);
    END LOOP;
END;
/
```

Another variation of the FOR LOOP in PL/SQL is the FOR (cursor variant) statement. The cursor FOR loop statement opens a previously declared cursor, fetches all rows in the cursor result set, and then closes the cursor. The columns of the SELECT statement are directly accessible inside the body of the FOR LOOP with the <for_loop_variable.column_name> syntax.

Example 2-29 shows a FOR LOOP over a cursor statement that uses the cursor values mapped from the cursor column for display.

Example 2-29 FOR LOOP over a cursor

```
DECLARE
    CURSOR ACCOUNT_cur IS
        SELECT ACCOUNT_id, NUM_PROJECTS
        FROM ACCOUNTS WHERE CREATE_DATE < SYSDATE + 30;
BEGIN
    FOR ACCOUNT_rec IN ACCOUNT_cur
    LOOP
        update_ACCOUNT (ACCOUNT_rec.ACCOUNT_id, ACCOUNT_rec.NUM_PROJECTS);
    END LOOP;
END;
```

GOTO and LABEL statements

Even though we do not recommend using “GOTO label” syntax, this programming construction could be used in case of errors to unconditionally redirect the programming logic to a statement or block label.

DB2 supports the *GOTO label_name* where the label must be defined with *<<label_name>>* syntax and be contained in the same PL/SQL block, either at the same level or higher up in the hierarchy.

Example 2-30 shows how to use a GOTO statement to exit the loop prematurely to the label *exit_perm*.

Example 2-30 GOTO statement

```
BEGIN
  FOR i IN (SELECT   table_name FROM user_tables )
    LOOP
      DBMS_OUTPUT.PUT_LINE(i.table_name);
      IF (i.table_name = 'DEPARTMENT') THEN GOTO exit_perm; END IF;
      DBMS_OUTPUT.PUT_LINE(i.table_name);
    END LOOP;
    <<exit_perm>>
    DBMS_OUTPUT.PUT_LINE('done');
END;
/
```

2.1.5 Condition (exceptions) handling

An exception is a separately encapsulated section of the SQL procedural code that captures and conditionally processes runtime errors. DB2 supports most of the Oracle PL/SQL syntax for exception handling, such as defining exception blocks, declaring custom exceptions, and raising custom defined errors (RAISE and RAISE_APPLICATION_ERROR).

The following is the general syntax for exception handling in a BEGIN block:

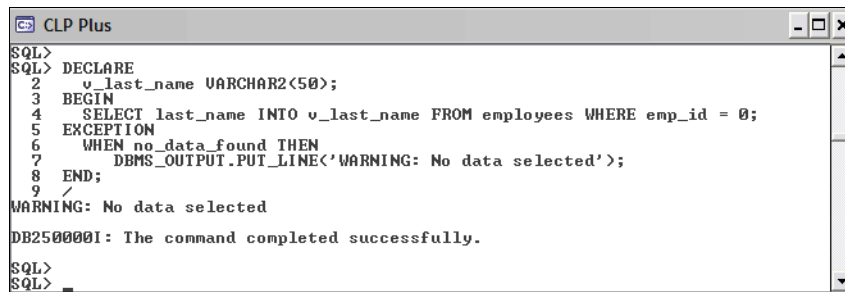
```
BEGIN
< executable statements>
EXCEPTION
  WHEN condition1 [ OR condition2 ]... THEN
    <exception handler logic>
  [ WHEN condition3 [ OR condition4 ]... THEN
    <exception handler logic>      ]...
END;
```

Predefined exceptions

DB2 provides support for most of the Oracle predefined exceptions, including CASE_NOT_FOUND, CURSOR_ALREADY_OPEN, DUP_VAL_ON_INDEX, INVALID_CURSOR, INVALID_NUMBER, NO_DATA_FOUND, OTHERS, NOT_LOGGED_ON, SUBSCRIPT_BEYOND_COUNT, LOGIN_DENIED, SUBSCRIPT_OUTSIDE_LIMIT, TOO_MANY_ROWS, VALUE_ERROR, and ZERO_DIVIDE. This is their general syntax:

```
WHEN exception_name THEN <executable statements>
```

Figure 2-1 demonstrates an anonymous block returning a warning as a result of catching a NO_DATA_FOUND exception in CLPPlus.



```
CLP Plus
SQL>
SQL> DECLARE
2   v_last_name VARCHAR2(50);
3 BEGIN
4   SELECT last_name INTO v_last_name FROM employees WHERE emp_id = 0;
5 EXCEPTION
6   WHEN no_data_found THEN
7     DBMS_OUTPUT.PUT_LINE('WARNING: No data selected');
8 END;
9 /
WARNING: No data selected
DB250000I: The command completed successfully.
SQL>
SQL>
```

Figure 2-1 Receiving NO_DATA_FOUND exception in an anonymous block

Custom exceptions

DB2 supports defining and calling custom defined exceptions. In Example 2-31, custom exceptions are defined with an exception declaration and the PRAGMA EXCEPTION_INIT syntax.

Example 2-31 Custom exception

```
CREATE OR REPLACE PROCEDURE Remove_Account
(p_AccountId      IN accounts.acct_id%TYPE,
 p_DeptCode       IN accounts.dept_code%TYPE
) IS

    -- exception declaration
    e_AccountNotRegistered EXCEPTION;
    PRAGMA EXCEPTION_INIT (e_AccountNotRegistered, -20050);

BEGIN
    DELETE FROM accounts
    WHERE acct_id = p_AccountId
      AND dept_code = p_DeptCode;

    IF SQL%NOTFOUND THEN
        RAISE e_AccountNotRegistered;
    END IF;

EXCEPTION
    WHEN e_AccountNotRegistered THEN
        DBMS_OUTPUT.PUT_LINE('Account ' || p_AccountId || ' does not exist.');
```

```
    WHEN others THEN
        RAISE
END Remove_Account;
/
```

Custom exceptions can be declared in a separate package to make them “global” and reusable throughout the application. In Example 2-32, we declare a user defined exception named AccountNotRegistered in the package header. Later on, a separate procedure named Remove_Account reuses this global exception by calling it with <package_name>.<exception_name>, as shown in Example 2-33 on page 53.

Example 2-32 Custom exception in a separate package

```
CREATE OR REPLACE PACKAGE pkgs_employees_activity AS
-- exception declaration
    e_AccountNotRegistered EXCEPTION;
    PRAGMA EXCEPTION_INIT (e_AccountNotRegistered, -20050);

    < procedures and functions declaration >

END pkgs_employees_activity;
```

/

Example 2-33 Exception call in a procedure

```
CREATE OR REPLACE PROCEDURE Remove_Account
(p_AccountId      IN accounts.acct_id%TYPE,
 p_DeptCode IN accounts.dept_code%TYPE
) IS

BEGIN
    < procedure specific code >

EXCEPTION
    WHEN pkgs_employees_activity.e_AccountNotRegistered THEN

        < application specific handling >

        RAISE;
END;
/
```

RAISE statement

The RAISE statement allows you to raise a previously-defined condition. Refer to Example 2-32 on page 52 and Example 2-33 for details.

RAISE_APPLICATION_ERROR

The procedure RAISE_APPLICATION_ERROR provides the capability to intentionally abort a process within an SQL procedural code from which it is called to cause an exception. The exception is handled in the same manner as described above. In addition, the RAISE_APPLICATION_ERROR procedure makes a user-defined code and error message available to the program, which can then be used to identify the exception. This is the general syntax:

```
RAISE_APPLICATION_ERROR(error_number, message);
```

Example 2-34 shows a procedure based on the EMPLOYEES table with multiple declarations of RAISE_APPLICATION_ERROR and its output when executed for an employee who has no corresponding manager ID in the table.

Example 2-34 RAISE_APPLICATION_ERROR

```
CREATE OR REPLACE PROCEDURE verify_employee
( p_empno          NUMBER
) IS
    v_ename          employees.last_name%TYPE;
    v_dept_code      employees.dept_code%TYPE;
    v_mgr            employees.emp_mgr_id%TYPE;
    v_hiredate        employees.create_date%TYPE;

BEGIN

    SELECT last_name, dept_code, emp_mgr_id, create_date
       INTO v_ename, v_dept_code, v_mgr, v_hiredate
       FROM employees
       WHERE emp_id = p_empno;

    IF v_ename IS NULL THEN
        RAISE_APPLICATION_ERROR (-20010, 'No name entered for ' || p_empno);
    END IF;

    IF v_dept_code IS NULL THEN
        RAISE_APPLICATION_ERROR (-20020, 'No department entered for' || p_empno);
    END IF;

    IF v_mgr IS NULL THEN
        RAISE_APPLICATION_ERROR (-20030, 'No manager entered for ' || p_empno);
    END IF;

    IF v_hiredate IS NULL THEN
        RAISE_APPLICATION_ERROR (-20040, 'No hire date entered for ' || p_empno);
    END IF;

    DBMS_OUTPUT.PUT_LINE ('Employee ' || p_empno || ' validated without errors');

EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE ('SQLCODE: ' || SQLCODE);
        DBMS_OUTPUT.PUT_LINE ('SQLERRM: ' || SQLERRM);
END;
```

```
Output:
SQL> set serveroutput on
SQL> execute verify_employee (1);
```

```
SQLCODE: -438
SQLERRM: SQL0438N Application raised error or warning with diagnostic text:
"No manager entered for 1".
```

SQLSTATE=UD999
DB2500001: The command completed successfully.

For help with mapping PL/SQL error codes and exception names to DB2 error codes and SQLSTATE values, refer to the DB2 Information Center, found at:

<http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/topic/com.ibm.db2.luw.apdv.plsql.doc/doc/r0055262.html>

2.1.6 Cursors data type

DB2 supports Oracle PL/SQL cursors and associated data types. A cursor is a built-in data type that defines a result set of rows that you can process with an application or PL/SQL logic. A cursor can be used in the following contexts: user-defined cursors types, global variables, parameters to procedures and functions, local variables, and return types of functions.

The general process for using cursors is:

1. Define the cursor with the associated SELECT statement when appropriate.
2. Open the cursor with optional input parameters.
3. Fetch one record at a time in a loop of some kind, moving the columns values of every record into application host variables or a PL/SQL variable (sometimes multiple fetches can be done at once to batch the fetches).
4. Close the cursor to free up database resources.

An application (PL/SQL program or client program) consumes the records produced by the cursors by fetching the records from the cursor one by one. Cursors are record-at-a-time lengthy operations, so they are less efficient than a SET-at-a-time operation. Cursors hold resources while opened. When a cursor is the main construct used for passing back information from RDBMSs to client applications, it is important to manage them properly. It is a best practice to properly close them as soon as they are no longer in use or in general to minimize the cursors' usage.

Depending on the cursor type, the operations you can do on cursors may be slightly different.

REF CURSORS

REF CURSOR is a type in PL/SQL that allows you to define cursor variables that can hold the pointers to the cursors. Cursor variables are frequently used to pass the result sets from the queries between various PL/SQL objects.

REF CURSORS are types, and in DB2 they must be created inside a package. Example 2-35 shows a usage of REF CURSOR and cursor variable Cur0.

Example 2-35 Using REF CURSOR

```
CREATE OR REPLACE PACKAGE ref_cursor_pack1
IS
    TYPE rcursor IS REF CURSOR;
END;
/
CREATE OR REPLACE PROCEDURE ref_cursor1(table_in IN VARCHAR2,
    table_out OUT VARCHAR2)
AS
    Cur0 ref_cursor_pack1.rcursor;
BEGIN
    OPEN Cur0 for
        SELECT table_name
        FROM syspublic.user_tables
        WHERE table_name >= table_in order by 1;
    LOOP
        FETCH Cur0 INTO table_out;
        EXIT WHEN (Cur0%FOUND or SQL%NOTFOUND);
    END LOOP;
    CLOSE Cur0;
    RETURN;
END;
/
```

Weakly typed cursors

Weakly typed cursors are the one that are not bound to a particular result set or type. When we need such cursor, we could utilize the SYS_REFCURSOR type. A weakly typed cursor is supported by DB2.

Weakly typed cursors are used when the definition of the result set is created dynamically or at run time. There is limited or no type checking on weakly typed cursors; therefore, their row structures can only be discovered at run time by applications. This is often used to flow cursors around (static, strongly typed, or weakly typed ones), passing cursors back and forth as parameters between procedures and functions. Weakly typed cursor variables are ideal for holding different cursors type at different points of time. For example, a procedure could be a “cursor factory”, returning different cursor types into the same OUT parameter based on various values or logic, such as a case statement.

Example 2-36 on page 57 shows a procedure using a weakly typed cursor.

Example 2-36 Using a weakly typed cursor

```
CREATE OR REPLACE PROCEDURE cursor_factory (  
    action IN INTEGER,  
    curs OUT sys_refcursor  
)  
AS  
BEGIN  
    IF action=1 THEN  
        OPEN curs FOR SELECT * FROM dept;  
    ELSIF action=2 THEN  
        OPEN curs FOR SELECT * FROM emp;  
    ELSE  
        NULL;  
    END IF;  
END;  
/  
  
DECLARE  
    v_ref_cur SYS_REFCURSOR;  
    r_dept dept%ROWTYPE;  
    r_emp emp%ROWTYPE;  
BEGIN  
    cursor_factory(1, v_ref_cur);  
    LOOP  
        FETCH v_ref_cur INTO r_dept;  
        EXIT WHEN v_ref_cur%NOTFOUND;  
        DBMS_OUTPUT.PUT_LINE(r_dept.dname);  
    END LOOP;  
    CLOSE v_ref_cur;  
END;  
/
```

Strongly typed cursors

Strongly typed cursors are cursors with a RETURN clause defining the number of columns and their types the cursor returns. These cursor data types are called strongly typed, because when result set values are assigned to them, the data types of the result sets can be checked.

Strongly typed cursors are used when the result set definition can be defined statically by a row, a record description, or an SQL SELECT statement; they are mainly used for static business logic. The strongly typed cursors provide the advantages where their structures and result sets type can be verified or checked by the compilation process or at assignment times, and errors are raised when data types or row mismatches occur. A strongly typed cursor of one type cannot be assigned to a cursor variable of another type.

Example 2-36 on page 57 cannot use a strongly typed cursor, since the strongly typed cursor by definition is tied only to single table or view. In Example 2-37, the reference cursor is anchored to the entire EMPLOYEE table, and it retrieves all the columns from that table.

Example 2-37 Using a strongly typed cursor

```
CREATE OR REPLACE PACKAGE cursor_package
IS
    TYPE scursor IS REF CURSOR RETURN employee%ROWTYPE;
END;
/

CREATE OR REPLACE PROCEDURE strongcursor ( name_var IN VARCHAR2, curs IN OUT
cursor_package.scursor)
AS
    emp_rows employee%ROWTYPE;
BEGIN
    OPEN curs FOR
        SELECT *
        FROM employee
        WHERE lastname >= name_var
        ORDER BY lastname;
    FETCH curs INTO emp_rows;
    DBMS_OUTPUT.PUT_LINE(emp_rows.lastname || ', ' || emp_rows.firstnme);
    CLOSE curs;
END;
/
```

Implicit cursors

Implicit cursors are cursors automatically declared, opened, and closed by certain PL/SQL constructs, such as a cursor declared in the FOR LOOP structure, or for a single row fetch of an SQL SELECT INTO statement.

Example 2-38 on page 59 demonstrates a usage of the implicit cursor. There is no explicit cursor declaration here.

Example 2-38 Using implicit cursor

```
DECLARE
BEGIN
    FOR rec IN (SELECT lastname, firstnme
                FROM employee ORDER BY lastname)
    LOOP
        DBMS_OUTPUT.PUT_LINE(rec.lastname || ', ' || rec.firstnme);
    END LOOP;
END;
/
```

Parameterized cursors

Parameterized cursors are strongly typed cursors where parameters are associated with the cursor in the definition. Example 2-39 shows a usage example. Note that DB2 requires that you specify the length of the parameter in the parameterized cursor, name0 VARCHAR2(30).

Example 2-39 Using a parameterized cursor

```
CREATE OR REPLACE PROCEDURE param_cursor(table_in IN VARCHAR2, table_out OUT
VARCHAR2)
AS
    CURSOR Cur0 (name0 VARCHAR2(30)) IS
        SELECT table_name
        FROM syspublic.user_tables
        WHERE table_name >= name0 order by 1;
BEGIN
    OPEN Cur0(table_in);
    FETCH Cur0 INTO table_out;
    CLOSE Cur0;
    RETURN;
END;
/
```

Static cursors

Static cursors are cursors that are associated with one query that is known at compile time. In Example 2-40, we modify the parameterized cursor example (Example 2-39 on page 59) to illustrate the static cursor.

Example 2-40 Using static cursor

```
CREATE OR REPLACE PROCEDURE static_cursor(table_in in varchar2, table_out out
varchar2)
AS
    CURSOR Cur0 IS
        SELECT table_name
        FROM syspublic.user_tables
        WHERE table_name >= table_in order by 1;
BEGIN
    OPEN Cur0;
    FETCH Cur0 INTO table_out;
    CLOSE Cur0;
    RETURN;
END;
/
```

2.1.7 Static and dynamic SQL support

PL/SQL supports both static and dynamic SQL execution. A Static SQL statement is compiled once, and then stored in the database for future runtime execution, while dynamic SQL is compiled at run time unless the compiled package happens to be cached in memory when the SQL is invoked.

The DB2 compatibility feature for PL/SQL allows both static and dynamic SQL. The static SQL statements supported can be a data definition language (DDL), data manipulation language (DML), or transaction control (COMMIT or ROLLBACK). EXECUTE IMMEDIATE is the primary mechanism for executing dynamic statement where an SQL text may not be fully defined until run time or specific static DDL SQLs that cannot be executed otherwise. In addition, EXECUTE IMMEDIATE can also execute a PL/SQL block.

It is sometimes convenient to execute data definition language (DDL) statements from within PL/SQL. Example 2-41 on page 61 shows how to use EXECUTE IMMEDIATE to make dynamically changes to an existing table.

Example 2-41 EXECUTE IMMEDIATE

```
CREATE TABLE empltest (empid VARCHAR2(6), emplastname VARCHAR2(30),
empfirstname VARCHAR2(30))
/
CREATE OR REPLACE PROCEDURE alter_table IS
BEGIN
    EXECUTE IMMEDIATE 'ALTER TABLE empltest RENAME COLUMN empid TO emp_id ';
    EXECUTE IMMEDIATE 'ALTER TABLE empltest ALTER COLUMN emp_id SET DATA TYPE
INT';
END;
/
```

When manipulating data with DML, it is always better to use static execution whenever possible, as this compiles and verifies the statements immediately. Example 2-42 shows a simple procedure with a few static SQL statements for manipulating data.

Example 2-42 Data manipulation

```
CREATE OR REPLACE PROCEDURE add_emp1 IS
BEGIN
    INSERT INTO empltest VALUES('111111', 'X', 'Y');
    INSERT INTO empltest VALUES('222222', 'X', 'Y');
    UPDATE empltest SET empid=999999 where empid=111111;
    DELETE empltest WHERE empid='222222';
END;
/
```

Using the EXECUTE IMMEDIATE style (Example 2-43) to achieve the same purpose provides no advantage. Besides, if a syntax error is present in one of these EXECUTE IMMEDIATE statements, it will only be discovered (too late) at their execution time instead of being caught at compilation time, early in the development process.

Example 2-43 EXECUTE IMMEDIATE is not required

```
CREATE OR REPLACE PROCEDURE add_emp2 IS
BEGIN
    EXECUTE IMMEDIATE 'insert into empltest values(''111111'', ''X'', ''Y'')';
    EXECUTE IMMEDIATE 'insert into empltest values(''222222'', ''X'', ''Y'')';
    EXECUTE IMMEDIATE 'update empltest set empid=''999999''
                        where empid=''111111''';
    EXECUTE IMMEDIATE 'delete empltest where empid=''222222''';
END;
/
```

On the other hand, EXECUTE IMMEDIATE is often used to execute a DML that is constructed, as shown in Example 2-44. It also demonstrates how variables can be passed between procedures.

Example 2-44 A common usage of EXECUTE IMMEDIATE

```
CREATE OR REPLACE PROCEDURE add_emp3 IS
    v_var1 VARCHAR2(6);
    v_var2 VARCHAR2(30);
    v_var3 VARCHAR2(30);
    v_var4 VARCHAR2(30);
    v_str VARCHAR2(300);
BEGIN
    v_var1 := '111111';
    v_var2 := 'L1';
    v_var3 := 'F1';
    v_var4 := '999999';
    v_str := 'insert into empltest values(' || '''' || v_var1 || '''' || ',' ||
        '''' || v_var2 || '''' || ',' || '''' || v_var3 || '''' || ')';

    EXECUTE IMMEDIATE v_str;
    EXECUTE IMMEDIATE 'update empltest set empid=' || '''' || v_var4 || '''' || ' '
        where empid=' || '''' || v_var1 || '''';
    EXECUTE IMMEDIATE 'delete empltest where empid=' || '''' || v_var4 || '''';
END;
/
```

Example 2-45 on page 63 shows an equivalent form of Example 2-44.

Example 2-45 A common usage of EXECUTE IMMEDIATE

```
CREATE OR REPLACE PROCEDURE add_emp4 IS
    v_var1 VARCHAR2(6);
    v_var2 VARCHAR2(30);
    v_var3 VARCHAR2(30);
    v_var4 VARCHAR2(30);
BEGIN
    v_var1 := '111111';
    v_var2 := 'L1';
    v_var3 := 'F1';
    v_var4 := '999999';

    EXECUTE IMMEDIATE 'insert into empltest values(:1, :2, :3)'
        using v_var1, v_var2, v_var3;
    EXECUTE IMMEDIATE 'update empltest set empid=:1
        where empid=:2' using v_var4, v_var1;
    EXECUTE IMMEDIATE 'delete empltest where empid=:1' using v_var4;
END;
/
```

Example 2-46 shows another dynamic SQL execution where the SQL statement is constructed dynamically and executed using EXECUTE IMMEDIATE.

Example 2-46 Using EXECUTE IMMEDIATE

```
CREATE OR REPLACE PROCEDURE del_emp1(name VARCHAR2, whereclause VARCHAR2)
IS
    str varchar2(30);
BEGIN
    str := 'delete ' || name || ' ' || whereclause ;
    EXECUTE IMMEDIATE str;
END;
/
```

EXECUTE IMMEDIATE can also execute a PL/SQL block, in addition to the dynamic or static SQL statements. In this case, we need to ensure that a complete block is specified. An anonymous block example that shows the execution of a CALL statement wrapped in a block is shown in Example 2-47.

Example 2-47 Using EXECUTE IMMEDIATE to execute PL/SQL block

```
DECLARE

id VARCHAR2(6);
LName VARCHAR2(6);
FName VARCHAR2(6);

BEGIN
    EXECUTE IMMEDIATE 'BEGIN add_emp3; END' ;
    --- if the procedue has arguments, use the following syntax
    --- EXECUTE IMMEDIATE 'BEGIN add_emp3(:1, :2 ,:3); END'
    --- using id, LName, FName;
END;
/
```

In DB2, EXECUTE IMMEDIATE does not currently support SELECT INTO or VALUES for retrieving data. You can use a cursor operation to achieve the direct equivalent.

2.1.8 Support for built-in scalar functions

DB2 provides many new built-in functions that increases the compatibility of applications originally written for Oracle. In general, functions can be classified by the actions they perform, such as character and string functions, conversion functions, error functions, table functions, and so on. In this section, we summarize some of the new and existing scalar functions available in DB2 that are commonly seen in Oracle applications.

Character and string functions

The character and string functions primarily deal with manipulation and processing of strings to produce a modified output based on the specific action performed on the input string. We list some of the commonly seen character and string functions in Table 2-5 on page 65.

Table 2-5 Some character and string functions supported in DB2

ASCII	LENGTH	RTRIM
CHR	LOWER	SOUNDEX
CONCAT	LPAD (new)	SUBSTR (updated)
CONCAT with	LTRIM	TRANSLATE
INITCAP (new)	REPLACE	TRIM
INSTR (new)	RPAD (new)	UPPER

Conversion functions

DB2 has extended TO_CHAR, TO_NUMBER, TO_DATE, and TO_TIMESTAMP conversion functions. These four functions are identical with their respective Oracle functions, except for the support of the third NLS parameter. Table 2-6 lists a few conversion functions supported in DB2.

Table 2-6 Some conversion functions supported by DB2

CAST	TO_CHAR (extended)	TO_TIMESTAMP (extended)	TO_NUMBER (extended)
CONVERT	TO_DATE (extended)	TO_CLOB (new)	

Date calculation functions

The date calculation functions help you manipulate the date and time. TO_DATE and TO_CHAR (especially useful for date manipulations) are also listed under Conversion functions. Table 2-7 presents some of the most widely used data manipulation functions.

Table 2-7 Some date calculation functions in DB2

ADD_MONTHS (new)	LOCALTIMESTAMP	SYSDATE (new)
CURRENT_DATE (new)	MONTHS_BETWEEN	TO_CHAR (extended)
CURRENT_TIMESTAMP (new)	NEXT_DAY (new)	TO_DATE (extended)
LAST_DAY (new)	ROUND (date)	TRUNC (date) (new)

Time zone functions

Oracle supports several functions that provide time zone information. In DB2, this functionality could be implemented with custom functions utilizing DB2 Timestamp data type. In the next few examples, we demonstrate how to utilize the power of DB2's date scalar functions to obtain the same output.

Example 2-48 shows how the From_TZ function can be implemented in DB2 to provide time zone information.

Example 2-48 From_TZ implementation

```
CREATE OR REPLACE FUNCTION from_tz_db2(ts TIMESTAMP, tz VARCHAR2)
  RETURN VARCHAR2(39)
IS
  am_pm VARCHAR2(10);
  tz1 VARCHAR2(10);
  tz2 VARCHAR2(10);
  indplus NUMBER;
  indminus NUMBER;
  colind NUMBER;
  syntax_function EXCEPTION;
BEGIN
  indminus := LOCATE('-', tz);
  indplus := LOCATE('+', tz);
  colind := LOCATE(':', tz);
  IF ( colind = 0 ) THEN
    raise syntax_function;
  END IF;
  IF ( colind = 2 ) THEN
    BEGIN
      IF ( indplus = 0 ) OR ( indminus = 0 ) THEN
        tz1 := '+0'||tz;
      END IF;
    END;
  ELSIF ( colind = 3 ) THEN
    BEGIN
      IF ( indminus = 0 ) AND ( indplus = 0 ) THEN
        tz1 := '+'||tz;
      ELSIF ( indplus != 0 ) THEN
        tz1 := '+0'||substr(tz,2, length(tz)-1);
      ELSE -- negative
        tz1 := '-0'||substr(tz,2, length(tz)-1);
      END IF;
    END;
  ELSIF ( colind = 4 ) THEN
    BEGIN
      IF ( indminus != 0 ) OR ( indplus != 0 ) THEN
        tz1 := tz;
      END IF;
    END;
  ELSE
    RAISE syntax_function;
  END IF;

  IF (length(tz1) = 5 ) AND ( locate(':', tz1) = 4 ) THEN
    tz2 := tz1||'0';
  ELSIF (length(tz1) = 4 ) AND ( locate(':', tz1) = 4 ) THEN
    tz2 := tz1||'00';
  ELSE
    tz2 := tz1;
```

```

END IF;
am_pm := CASE WHEN (hour(ts) <= 12) THEN ' AM ' ELSE ' PM ' END;
RETURN CAST(ts AS TIMESTAMP(9)) || am_pm || tz2;
EXCEPTION
WHEN syntax_function THEN
    RAISE_APPLICATION_ERROR(-20001, 'incorrect syntax format for '||tz);
END;
/

```

Example 2-49 shows a simple implementation for Oracle TZ_OFFSET function that returns the zone offset for a given time zone. The exception checking is not included for simplicity. Also, not all the zones are shown, as there are more than 400 zones.

Example 2-49 TZ_OFFSET implementation

```

CREATE OR REPLACE FUNCTION tz_offset_db2(zone VARCHAR2)
    RETURN VARCHAR2(6)
IS
    offset VARCHAR2(6);
BEGIN
    offset := CASE
        WHEN zone = 'Africa/Johannesburg' THEN      '+02:00'
        WHEN zone = 'Africa/Khartoum' THEN          '+02:00'
        WHEN zone = 'Africa/Mogadishu' THEN          '+03:00'
        WHEN zone = 'Africa/Nairobi' THEN            '+03:00'
        WHEN zone = 'Africa/Nouakchott' THEN         '+00:00'
        WHEN zone = 'Africa/Tripoli' THEN             '+02:00'
        WHEN zone = 'Africa/Tunis' THEN               '+01:00'
        WHEN zone = 'Africa/Windhoek' THEN            '+01:00'
        WHEN zone = 'America/Adak' THEN               '-09:00'
        WHEN zone = 'America/Anchorage' THEN          '-08:00'
        WHEN zone = 'America/Anguilla' THEN           '-04:00'
        WHEN zone = 'America/Araguaina' THEN          '-03:00'
        WHEN zone = 'America/Aruba' THEN              '-04:00'
        .....
        WHEN zone = '+00:00' THEN                      '+00:00'
        WHEN zone = '+01:00' THEN                      '+01:00'
        WHEN zone = '+02:00' THEN                      '+02:00'
        WHEN zone = '+03:00' THEN                      '+03:00'
        .....
    ELSE
        '+99:99'
    END;

    RETURN offset;

END;
/

```

From TZ_OFFSET_ZONE, it is possible to develop a DB2 version of NEW_TIME scalar function, which takes a time in one time zone and converts it to a time in another time zone, for example, NEW_TIME (time, zone1, zone2). Note that this function will return the right output only if the database is in Oracle compatibility mode.

Example 2-50 shows a simple implementation (assuming that you use the correct inputs).

Example 2-50 NEW_TIME implementation

```

CREATE OR REPLACE FUNCTION new_time_db2(date1 DATE, zone1 VARCHAR, zone2 varChar)
  RETURN DATE
IS
    offset1 NUMBER;
    offset2 NUMBER;

BEGIN
    offset1 := CASE
        WHEN zone1 = 'AST' THEN          -4
        WHEN zone1 = 'ADT' THEN          -3
        WHEN zone1 = 'BST' THEN         -11
        WHEN zone1 = 'BDT' THEN         -10
        WHEN zone1 = 'CST' THEN          -6
        WHEN zone1 = 'CDT' THEN          -5
        WHEN zone1 = 'EST' THEN          -5
        WHEN zone1 = 'EDT' THEN          -4
        WHEN zone1 = 'GMT' THEN           0
        WHEN zone1 = 'HST' THEN         -10
        WHEN zone1 = 'HDT' THEN          -9
        WHEN zone1 = 'MST' THEN          -7
        WHEN zone1 = 'MDT' THEN          -6
        WHEN zone1 = 'NST' THEN         -3.5
        WHEN zone1 = 'PST' THEN          -8
        WHEN zone1 = 'PDT' THEN          -7
        WHEN zone1 = 'YST' THEN          -9
        WHEN zone1 = 'YDT' THEN          -8
    ELSE
        99
    END;

    offset2 := CASE
        WHEN zone2 = 'AST' THEN          -4
        WHEN zone2 = 'ADT' THEN          -3
        WHEN zone2 = 'BST' THEN         -11
        WHEN zone2 = 'BDT' THEN         -10
        WHEN zone2 = 'CST' THEN          -6
        WHEN zone2 = 'CDT' THEN          -5
        WHEN zone2 = 'EST' THEN          -5
        WHEN zone2 = 'EDT' THEN          -4

```

```

        WHEN zone2 = 'GMT' THEN          0
        WHEN zone2 = 'HST' THEN        -10
        WHEN zone2 = 'HDT' THEN         -9
        WHEN zone2 = 'MST' THEN         -7
        WHEN zone2 = 'MDT' THEN         -6
        WHEN zone2 = 'NST' THEN        -3.5
        WHEN zone2 = 'PST' THEN         -8
        WHEN zone2 = 'PDT' THEN         -7
        WHEN zone2 = 'YST' THEN         -9
        WHEN zone2 = 'YDT' THEN         -8
    ELSE
        99
    END;

    RETURN date1 + (offset2 - offset1)/24;
END;
/

```

Oracle interval functions that utilize Interval Year To Date and Interval Day To Second data types, could be implemented in DB2 by utilizing the DB2 timestamp data type, which components will be considered as intervals, not as a point in time. At this point, a group of DB2 scalar functions such as YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, and MICROSECOND, could be employed to manipulate the data. These functions have no equivalent in Oracle.

For more examples, refer to Appendix C, “Function Mapping”, of *Oracle to DB2 Conversion Guide for Linux, UNIX, and Windows*, SG24-7048.

Mathematical functions

DB2 also provides a rich set of mathematical functions, as shown in Table 2-8.

Table 2-8 Mathematical functions

ABS	CEIL	MAX	STDDEV
ACOS	COS	MIN	SUM
ASIN	COSH	MOD	TAN
ATAN	COUNT	LN	SQRT
ATAN2	DENSE_RANK	POWER@	TANH
AVG	EXP	RANK	TRUNC (numbers)
BITAND	EXTRACT (new)	ROUND (numbers)	VAR_POP
BITANDNOT	FLOOR	SIGN	VAR_SAMP
BITOR	GREATEST	SIN	VARIANCE
BITXOR	LEAST	SINH	

Error functions

Error functions, `SQLCODE` and `SQLERRM`, are used to raise exception or retrieve error codes. They can only be used in the `EXCEPTION` section of PL/SQL blocks.

SQLCODE

The `SQLCODE` function returns the `SQLCODE` value associated with the raised exception. Example 2-51 shows a usage of the `SQLCODE` function.

Example 2-51 SQLCODE function

```
EXCEPTION WHEN OTHERS THEN
    raise_application_error(-20001,'SQLCODE is ' || SQLCODE);
END;
```

SQLERRM

The `SQLERRM` function returns the error message associated with the raised exception.

Example 2-52 shows a usage of the `SQLERRM` function.

Example 2-52 SQLERRM function

```
EXCEPTION WHEN OTHERS THEN
    raise_application_error(-20001,'SQLERRM is ' || SQLERRM);
END;
```

Miscellaneous functions

Oracle offers some specific functions that are not available on other database platforms. To ensure a seamless transition from Oracle to DB2, DB2 provides support for some of the most widely used Oracle functions, as shown in Table 2-9.

Table 2-9 Miscellaneous functions

CARDINALITY	DECODE (DB2 9.5)	NULLIF
COALESCE	LAG (DB2 9.5)	NVL (DB2 9.5)
CURRENT_USER	LEAD (DB2 9.5)	

Here we provide examples of `DECODE` and `NVL` functions for reference to show that there is no difference in how these functions operate on DB2 and Oracle. We also demonstrate how the function `USERENV` could be implemented in DB2.

DECODE

DB2 supports the DECODE function in the same way Oracle does, which is also very similar to the CASE statement. In Example 2-53, the query selects the number of current projects per employee and displays a conditional message based on this number.

Example 2-53 DECODE function

```
SELECT emp_id, last_name, current_projects,  
       DECODE(current_projects,  
              0, 'Attention - No projects',  
              1, 'Attention - Single project',  
              2, 'Attention - Need to assign projects',  
              3, 'Good job - Working on 3 projects',  
              4, 'Great job - Working on 4 projects',  
              5, 'Excellent - Do not assign more projects',  
              NULL, 'Verify project assignments',  
              'Error with project assignments')  
FROM employees  
ORDER BY current_projects;
```

Example 2-54 demonstrates part of the result set returned by the query in DECODE.

Example 2-54 DECODE query output

29 PARKER	0 Attention - No projects
3 KWAN	1 Attention - Single project
7 PULASKI	1 Attention - Single project
2 THOMPSON	2 Attention - Need to assign projects
5 GEYER	2 Attention - Need to assign projects
9 HENDERSON	3 Good job - Working on 3 projects
18 SCOUTTEN	3 Good job - Working on 3 projects
14 NICHOLLS	4 Great job - Working on 4 projects
1 HAAS	5 Excellent - Do not assign more projects
19 WALKER	5 Excellent - Do not assign more projects

NVL

In a similar fashion to the DECODE function, the NVL function handles conditional NULL values and is fully supported in DB2. The query in Example 2-55 returns a specific message when department information is not yet assigned to the employee.

Example 2-55 Query with NVL function

```
SELECT e.emp_id, e.first_name, e.last_name, e.dept_code,  
NVL(d.dept_name, 'Unassigned or unknown Department') as department  
FROM  
    employees e,  
    departments d  
WHERE  
    e.dept_code =d.dept_code (+)  
    and emp_id between 30 and 35  
ORDER BY department DESC, emp_id asc;
```

Example 2-56 presents the output of this query. Note the department description for the NEW department code.

Example 2-56 Output of query with NVL function

31	MAUDE	SETRIGHT	NEW	Unassigned or unknown Department
34	JASON	GOUNOT	NEW	Unassigned or unknown Department
32	RAMLAL	MEHTA	E21	SUPPORT SERVICES
33	WING	LEE	E21	SUPPORT SERVICES
30	PHILIP	SMITH	E11	SPIFFY COMPUTER SERVICE DIV

Today, there is a wide range of scalar functions used in the database industry and each RDBMS provides specific functions designed to satisfy the requirements of their users. New functions are released by the database vendors with every new release. For any function that is not provided in DB2, you could easily develop a corresponding DB2 equivalent by using either PL/SQL or SQL PL language. In Oracle, a raw type stores character data and is byte-oriented. In DB2, the equivalent type for raw type is VARCHAR FOR BIT DATA. Example 2-57 shows an example of RawToHex implementation in DB2

Example 2-57 RawToHex implementation

```
CREATE OR REPLACE FUNCTION RAW2HEX  
    (x1 VARCHAR(100) FOR BIT DATA)  
RETURNS VARCHAR2(100)  
SOURCE HEX(VARCHAR(100) FOR BIT DATA)
```

USERENV

USERENV function is specific to Oracle and can be used to retrieve information about the current Oracle session. The function accepts several different input parameters, as shown in Figure 2-2.

Parameter	Returned Value
CLIENT_INFO	The user session information inserted using the DBMS_APPLICATION_INFO package
ENTRYID	Current Session ID for auditing session
SESSIONID	Current Session ID
INSTANCE	Oracle instance ID
ISDBA	TRUE or FALSE depending if the user has been granted DBA privileges or not
LANG	The ISO abbreviation for the language
LANGUAGE	The language, territory, and Character Set of the current session
TERMINAL	The Operating system ID of the current session

Figure 2-2 Parameters that could be passed to the USERENV function

To enforce our language statement, we provide two examples in Example 2-58 that shows how to implement the USERENV function in DB2 in both DB2 SQL PL and PL/SQL styles. Both functions return the same result set. Depending on the value of the input parameter, the function returns specific information for the current session identifier. In the example, only a few parameters are used.

Example 2-58 USERENV function

```
--SQL PL style:
-----
CREATE OR REPLACE FUNCTION userenv( p VARCHAR(250))
  RETURNS VARCHAR(128)
  DETERMINISTIC
  RETURN ( CASE upper(p)
            WHEN 'SCHEMAID'      then current schema
            WHEN 'CURRENT_USER'  then current user
            WHEN 'SESSIONID'     then APPLICATION_ID()
            ELSE 'other_user_info'
          END) ;
-----

-- PL/SQL style:
-----
CREATE OR REPLACE FUNCTION userenv(p VARCHAR2)
  RETURN VARCHAR2 IS
  v_str      VARCHAR2(128);
  v_app_id   VARCHAR2(50);
  BEGIN
    v_str:=( case upper(p)
              WHEN 'SCHEMAID'      then current schema
              WHEN 'CURRENT_USER'  then current user
              WHEN 'SESSIONID'     then APPLICATION_ID()
              else 'other_user_info'
            END );
    RETURN v_str;
  END;
```

2.1.9 Routines, procedures, and functions compatibility

In this section, we look at moving the routines, procedures, and functions originally written for execution against an Oracle database to DB2.

PL/SQL procedures and functions are named blocks that persist in the database through the CREATE PROCEDURE or CREATE FUNCTION statements. Both a procedure and function contain executable statements, but differ in certain characteristics. The procedure takes zero or more inputs/outputs and may not return a value, but a function takes zero or more inputs/outputs and will always return an output value. During the run time, the procedure is invoked as a PL/SQL statement from the command line, another procedure, or function, trigger, or anonymous block. A function is always invoked as part of an

expression, in an SQL, or an assignment statement, where the contexts are valid in the command line, another procedure, or function, trigger, or anonymous block.

DB2 also provides support for anonymous blocks, that is, unnamed PL/SQL blocks that are not stored persistently in the database catalog.

Anonymous blocks

Anonymous blocks are PL/SQL constructs that contain unnamed blocks of code, which will not be stored persistently in the database, but are primarily intended for a single time execution. Unlike named blocks, which are persistently stored in the database, the compilation and execution of an anonymous block are combined in one step, which offers the flexibility to make immediate changes and execute them in the same time. For comparison, a stored procedure must be recompiled in a separate step every time its definition changes.

Anonymous blocks are often used to test, troubleshoot, and develop stored procedures, simulate application runs, and build complex, *ad hoc* queries on the fly. During the execution of anonymous blocks, if an exception occurs and is caught, the transaction control can be handled accordingly in the exception section. If the exception is not caught, all statements prior to the exception are rolled back to the previous commit point.

There are many examples throughout the book that illustrate the use of anonymous blocks. Example 2-59 shows a simple anonymous block to illustrate the basic construct.

Example 2-59 Simple anonymous block

```
DECLARE
    current_date DATE := SYSDATE;
BEGIN
    DBMS_OUTPUT.PUT_LINE( current_date );
END;
/
```

For more details about anonymous blocks, refer to the article “DB2 9.7: Using PL/SQL anonymous blocks in DB2 9.7”, found at:

<http://www.ibm.com/developerworks/data/library/techarticle/dm-0908anonymousblocks/index.html>

You can also refer to the DB2 Information Center, found at:

<http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/topic/com.ibm.db2.luw.apdv.plsql.doc/doc/c0053848.html>

Procedures compatibility

DB2's PL/SQL support covers a wide range of Oracle PL/SQL features, syntactically and semantically. Within this compatible context, an Oracle PL/SQL procedure can be compiled and invoked directly on DB2. Throughout this chapter, we cover many of the supported features in detail and demonstrate procedure examples.

Example 2-60 shows a PL/SQL procedure. With DB2's PL/SQL support, you can directly compile and run this procedure in DB2 without any modification.

Example 2-60 PL/SQL procedure

```
CREATE OR REPLACE PROCEDURE ADD_NEW_EMPLOYEE (
    p_FirstName EMPLOYEES.first_name%TYPE,
    p_LastName EMPLOYEES.last_name%TYPE,
    p_EmpMgrId EMPLOYEES.emp_mgr_id%TYPE,
    p_DeptCode EMPLOYEES.dept_code%TYPE,
    p_Account EMPLOYEES.acct_id%TYPE,
    p_CreateDate EMPLOYEES.create_date%TYPE DEFAULT SYSDATE,
    p_OfficeId EMPLOYEES.office_id%TYPE DEFAULT 2
) AS

-- variable and cursor declaration
v_EmployeeId EMPLOYEES.emp_id%TYPE :=1;
v_EmployeeIdTemp EMPLOYEES.emp_id%TYPE;
CURSOR c_CheckEmployeeId IS SELECT 1 FROM EMPLOYEES WHERE emp_id=v_EmployeeId;

BEGIN
    -- Find Next available employee id from the employee sequence
    LOOP
        SELECT employee_sequence.NEXTVAL INTO v_EmployeeId FROM DUAL;
        OPEN c_CheckEmployeeId;
        FETCH c_CheckEmployeeId INTO v_EmployeeIdTemp;
        EXIT WHEN c_CheckEmployeeId%NOTFOUND;
    END LOOP;

    CLOSE c_CheckEmployeeId;
    SELECT employee_sequence.CURRVAL INTO v_EmployeeId FROM DUAL;
    INSERT INTO EMPLOYEES(emp_id, first_name, last_name, current_projects,
        emp_mgr_id, dept_code, acct_id, office_id, band,
        create_date)
    VALUES (v_EmployeeId , INITCAP(p_FirstName), INITCAP(p_LastName), 0,
        p_EmpMgrId ,p_DeptCode, p_Account, p_OfficeId, 1, p_CreateDate;

    DBMS_OUTPUT.PUT_LINE('Employee record id ' || v_EmployeeId ||
        ' was created successfully.');
```

```
EXCEPTION
    WHEN others THEN
        DBMS_OUTPUT.PUT_LINE('Employee record was not created.');
```

```
RAISE;
END ADD_NEW_EMPLOYEE;
```

/

DB2's implementation of PL/SQL covers the most commonly used language elements. Many applications can move to DB2 with no code changes at all, but it is not uncommon for a database migration to require a small number changes. For those few unavailable language elements you encounter, the PL/SQL code will require modification to provide the equivalent results in DB2.

Example 2-61 shows an Oracle procedure using TYPE.

Example 2-61 Using TYPE in Oracle

```
CREATE OR REPLACE PROCEDURE ref_cursor1(table_in IN VARCHAR2,
                                         table_out OUT VARCHAR2)
AS
TYPE rcursor IS REF CURSOR;
Cur0 rcursor;
BEGIN
    OPEN Cur0 for
        SELECT table_name
        FROM syspublic.user_tables
        WHERE table_name >= table_in order by 1;
    LOOP
        FETCH Cur0 INTO table_out;
        EXIT WHEN (Cur0%FOUND or SQL%NOTFOUND);
    END LOOP;
    CLOSE Cur0;
    RETURN;
END;
/
```

In DB2, the TYPE construct declarations are implemented either by using CREATE TYPE statement (to define the TYPE as separate database object) or in a PL/SQL package, as shown in Example 2-62. Note how a proper TYPE reference is created in the objects where the type is used.

Example 2-62 Using REF CURSOR

```
CREATE OR REPLACE PACKAGE ref_cursor_pack1
IS
    TYPE rcursor IS REF CURSOR;
END;
/
CREATE OR REPLACE PROCEDURE ref_cursor1(table_in IN VARCHAR2,
    table_out OUT VARCHAR2)
AS
    Cur0 ref_cursor_pack1.rcursor;
BEGIN
    OPEN Cur0 for
        SELECT table_name
        FROM syspublic.user_tables
        WHERE table_name >= table_in order by 1;
    LOOP
        FETCH Cur0 INTO table_out;
        EXIT WHEN (Cur0%FOUND or SQL%NOTFOUND);
    END LOOP;
    CLOSE Cur0;
    RETURN;
END;
/
```

Nested routines (procedures or functions) should be implemented in DB2 as stand-alone database objects or inside PL/SQL packages. In Example 2-63, the nestedproc1 procedure contains the declaration of another procedure and one function.

Example 2-63 Oracle nested procedure

```
CREATE OR REPLACE PROCEDURE nestedproc1 (p_arg1 IN VARCHAR2, p_arg2 OUT
VARCHAR2)
IS
    var1 VARCHAR2(30);

    PROCEDURE localProc1(p_arg3 IN VARCHAR2, p_arg3o OUT VARCHAR2)
    IS
        BEGIN
            p_arg3o := ' ' || p_arg3;
        END;

    FUNCTION localFunc2(p_arg4 IN VARCHAR2) RETURN VARCHAR2
```

```

    IS
    BEGIN
        return INITCAP(p_arg4);
    END;

BEGIN
    localProc1(p_arg1, var1);
    p_arg2 := localFunc2(p_arg1||var1);
END;
/

```

Example 2-64 illustrates how to implement the nested routines from Example 2-82 in PL/SQL package in a way that works on both DB2 and Oracle. Note that the localProc1 and localFunc2 are defined only inside the package body, which makes them private to the package and makes them correspond to the way they were nested in nestedproc1 in Example 2-63 on page 78.

Example 2-64 DB2 Nested procedure

```

CREATE OR REPLACE PACKAGE nestedpack1
IS
    var1 VARCHAR2(30);
    PROCEDURE nestedproc1(p_arg1 IN VARCHAR2, p_arg2 OUT VARCHAR2);
END nestedpack1;
/

CREATE OR REPLACE PACKAGE BODY nestedpack1
IS
    PROCEDURE nestedproc1(p_arg1 IN VARCHAR2, p_arg2 OUT VARCHAR2)
    IS
        BEGIN
            localProc1(p_arg1, var1);
            p_arg2 := localFunc2(p_arg1||var1);
        END;

    PROCEDURE localProc1(p_arg3 IN VARCHAR2, p_arg3o OUT VARCHAR2)
    IS
        BEGIN
            p_arg3o := ' ' || p_arg3;
        END;

    FUNCTION localFunc2(p_arg4 IN VARCHAR2) RETURN VARCHAR2
    IS
        BEGIN
            return INITCAP(p_arg4);
        END;

END nestedpack1;
/

```

Functions compatibility

The building blocks for PL/SQL functions are similar to the PL/SQL procedures, except, as noted earlier, with respect to the returning value and the invocation method. In general, the supported features for procedures are applied to functions except for the autonomous transaction features. However, as DB2 implementation details of procedures and functions are a bit different, in some exceptional cases a handful of constructs that work in procedures may not be applicable to functions.

Recursion is a powerful technique in programming language. Just as Oracle does, DB2 supports recursive SQL calls through mechanisms, such as the CONNECT BY/PRIOR constructs or by using COMMON TABLE EXPRESSION, examples of which are available in Hierarchical queries.

DB2 also supports recursion through function calls in PL/SQL, where a function or a procedure calls itself with different arguments. A recursive function or procedure should have the logical conditions to ensure that the routine does not loop forever. DB2 recursion support in PL/SQL is semantically similar with slightly syntax differences due to implementation differences.

Example 2-65 shows a simple Oracle recursive function (REPORT_CHAIN) that calls itself.

Example 2-65 Oracle recursive function

```
CREATE OR REPLACE FUNCTION report_chain(p_emp_id IN NUMBER) RETURN VARCHAR2

AS
  p VARCHAR2(30);
  empmgrid NUMBER(10);

BEGIN

  SELECT emp_mgr_id
    INTO empmgrid
    FROM employees
   WHERE emp_id = p_emp_id;

  p := report_chain(empmgrid);

  IF empmgrid is not NULL THEN
    RETURN '|||empmgrid || '#' || p || '*';

  END IF;

  EXCEPTION WHEN OTHERS THEN RETURN ' ';

END;
```


/

To implement recursive functions in DB2 and achieve the same result, you can use the EXECUTE IMMEDIATE statement to call the recursive function, as shown in Example 2-66.

Example 2-66 DB2 recursive function

```
CREATE OR REPLACE FUNCTION report_chain(p_emp_id IN NUMBER) RETURN VARCHAR2
AS
  p VARCHAR2(30);
  empmgrid NUMBER(10);
  v_stmt VARCHAR2(512);
BEGIN

  SELECT emp_mgr_id
    INTO empmgrid
    FROM employees
   WHERE emp_id = p_emp_id;

  v_stmt := 'set ? = report_chain(:1)';

  execute immediate v_stmt into p using empmgrid;

  IF empmgrid is not NULL THEN
    RETURN '||' || empmgrid || '# ' || p || '*';
  END IF;

  EXCEPTION WHEN OTHERS THEN RETURN ' ';
END report_chain;
/
```

Moving external routines

External procedures and functions written in Java and C/C++ are frequently found in both Oracle and DB2 applications. We provide two examples of building routines using C and Java. Although in most cases there will be no changes to the code, it is important to review the code and ensure that it is compatible with parameter style handling in DB2.

For complete information about building and running external procedures and functions, consult the following IBM DB2 documents:

- ▶ *Getting Started with Database Application Development*, SC10-4252
- ▶ *Developing SQL and External Routines*, SC10-4373

Building C/C++ routines

We now show an example of creating a stored procedure written in C. Here are some basic steps that must be performed when creating any C procedure (or function):

1. Create the external procedure or function and save it on your file system. If the procedure or function contains embedded SQL, then it should be saved with the extension `.sqc`; if not, save it with the extension `.c`.
2. Create the export file (AIX only)

AIX requires you to provide an export file that specifies which global functions in the library are callable from outside it. This file must include the names of all routines in the library. Other UNIX platforms simply export all global functions in the library. Here is an example of an AIX export file for the procedure outlanguage that exists in the file `spserver.sqc`:

```
#!/spserver export file
outlanguage
```

The export file `spserver.exp` lists the stored procedure outlanguage. The linker uses `spserver.exp` to create the shared library `spserver` that contains the outlanguage stored procedure.

3. On Windows, a DEF file is required, which has a similar purpose. See `sqllib/samples/c/spserver.def` for an example.
4. Run the `bldrtn` script, which creates the shared library:

```
bldrtn my_routine my_database_name
```

The script copies the shared library to the server in the path `sqllib/function`.

5. Catalog the routines by running the `catalog_my_routine` script on the server.

After a shared library is built, it is typically copied into a directory from which DB2 will access it. When attempting to replace a routine shared library, you should either run `/usr/sbin/slibclean` to flush the AIX shared library cache, or remove the library from the target directory and then copy the library from the source directory to the target directory. Otherwise, the copy operation may fail because AIX keeps a cache of referenced libraries and does not allow the library to be overwritten.

DB2 provides build scripts for precompiling, compiling, and linking C stored procedures. These are located in the `sqllib/samples/c` directory, along with sample programs that can be built with these files. This directory also contains the `embprep` script that is used within the build script to precompile a `*.sqc` file. The build scripts have the `.bat` (batch) extension on Windows, and have no extension on UNIX platforms. For example, `bldrtn.bat` is a script to build C/C++ stored procedure on a Windows platform; `bldrtn` is the equivalent on UNIX.

Here is a simple example of creating and cataloging a stored procedure written in C. This procedure queries the SYSPROCEDURES table from the DB2 System Catalog to determine in which language (JAVA, C, SQL, and so on) that the BONUS_INCREASE procedure is written. Perform these steps:

1. Create and save the C source file (Example 2-67), with embedded SQL, as outlanguage.sqc.

Example 2-67 Stored procedure in C

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlca.h>
#include <sqlc.h>
#include <sqludf.h>
#include <sql.h>
#include <memory.h>
SQL_API_RC SQL_API_FN outlanguage(char language[9]){
    struct sqlca sqlca;
    EXEC SQL BEGIN DECLARE SECTION;
    char out_lang[9];
    EXEC SQL END DECLARE SECTION;
    /* Initialize strings used for output parameters to NULL */
    memset(language, '\0', 9);
    EXEC SQL SELECT language INTO :out_lang
    FROM sysibm.sysprocedures
    WHERE procname = 'BONUS_INCREASE';
    strcpy(language, out_lang);
    return 0;
} /* outlanguage function */
```

2. Create and save the .exp file as outlanguage.exp. Here are the contents of that file:

```
outlanguage
```

3. Create and save the outlanguage_crt.db2 file, which catalogs the procedure. Here are its contents:

```
CREATE PROCEDURE outlanguage (OUT language CHAR(8))
DYNAMIC RESULT SETS 0
LANGUAGE C
PARAMETER STYLE SQL
NO DBINFO
FENCED NOT THREADSAFE
MODIFIES SQL DATA
PROGRAM TYPE SUB
EXTERNAL NAME 'outlanguage!outlanguage'!
```

4. Execute the build script bldrtn for the outlanguage.sqc file using the db2_emp database:

```
bldrtn outlanguage db2_emp
```

5. Make a connection to the database by running this command:

```
db2 connect to db2_emp
```

6. Execute the script to catalog the procedure by running this command:

```
db2 -td! -vf outlanguage_crt.db2 > message.out
```

DB2 responds with the message:

```
DB20000I The SQL command completed successfully.
```

The message.out file should be viewed for messages, especially if any other message than The SQL command completed successfully is returned.

7. Test the procedure by running this command:

```
db2 "call outlanguage(?)"
```

The results are:

```
Value of output parameters
```

```
-----
```

```
Parameter Name : LANGUAGE
```

```
Parameter Value : SQL
```

```
Return Status = 0
```

Building Java routines

Here are the basic steps to create an external Java user defined function (UDF) from the DB2 command window:

1. Compile your_javaFileName.java to produce the file your_javaFileName.class by running this command:

```
javac your_javaFileName.java
```
2. Copy your_javaFileName.class to the sqllib\function directory on Windows operating systems, or to the sqllib/FUNCTION directory on UNIX.
3. Connect to the database by running this command:

```
db2 connect to your_database_name
```
4. Register the your_javaFileName library in the database by using the CREATE FUNCTION SQL statement:

```
db2 -td! -vf <your_create_function_statement.db2>
```

Here is an example of a procedure that will create a Java UDF that will retrieve the system name from the DB2 Registry variable DB2SYSTEM:

1. The Java source file shown in Example 2-68 is saved as db2system_nameUDF.java.

Example 2-68 UDF Java source

```
import java.io.*;
public class db2system_nameUDF {
    public static String db2system_name() {
        Runtime rt = Runtime.getRuntime();
        Process p=null;
        String s = null;
        String returnString = "";
        try {
            // WINDOWS: **** uncomment and compile the following for Windows
            // p = rt.exec("cmd /C db2set DB2SYSTEM");
            // UNIX: **** uncomment and compile the following for UNIX
            p = rt.exec("db2set DB2SYSTEM");
            BufferedInputStream buffer =
                new BufferedInputStream(p.getInputStream());
            BufferedReader commandResult =
                new BufferedReader(new InputStreamReader(buffer));
            try {
                while ((s = commandResult.readLine()) != null)
                    returnString = returnString.trim() + s.trim() ;
                // MAX number of chars for the DB2SYSTEM variable is 209
                // characters
                commandResult.close();
                // Ignore read errors; they mean process is done
            } catch (Exception e) {
            }
        } catch (IOException e) {
            returnString = "failure!";
        }
        return(returnString);
    }
}
```

2. Compile the Java source. The compile command is:
javac db2system_nameUDF.java
3. Copy the .class file to the /sqllib/function directory using this command:
\$ cp db2system_nameUDF.java /home/db2inst1/sqllib/function

4. Construct the CREATE FUNCTION file and save it as db2system_name.db2:

```
DROP FUNCTION DB2SYSTEM_NAME !
CREATE FUNCTION DB2SYSTEM_NAME ()
RETURNS VARCHAR(209)
EXTERNAL NAME 'db2system_nameUDF!db2system_name'
LANGUAGE JAVA
PARAMETER STYLE JAVA
NOT DETERMINISTIC
NO SQL
EXTERNAL ACTION!
```

5. Connect to the database using this command:

```
db2 connect to db2_emp
```

6. Execute the script to register the UDF with the database using this command:

```
db -td! -vf db2system_name.db2
```

7. Test the UDF using this command:

```
db2 "values db2system_name()"
```

The results are:

```
-----
smpoaix

1 record(s) selected.
```

2.1.10 Moving PL/SQL packages

Both Oracle and DB2 support the concept of grouping database related objects in a single unit known in Oracle as a package and as a module in DB2. In addition to its native modules, DB2 provides support for the full set of PL/SQL modular features extended to both user and system defined (built-in) packages. Therefore, the definition of a package/module as “*a database object that is a collection of other database objects, such as functions, procedures, types, and variables*” is valid for both database systems. In this book, the terms module and package are used interchangeably.

Packages have multiple features that make them very useful as database objects. Package features allow you to:

- ▶ Extend schema support by allowing you to group together, in a named set, a collection of related data type definitions, database object definitions, and other logic elements, including:
 - PL/SQL procedures and functions
 - Public prototype of procedures and functions
 - User-defined data type definitions, including distinct type, array type, associative array type, row type, and cursor type
- ▶ Define a name space such that objects defined within the package can refer to other objects defined in the package without providing an explicit qualifier.
- ▶ Add object definitions that are private to the package. These objects can only be referenced by other objects within the package.
- ▶ Add object definitions that are published. Published objects can be referenced from within the package or from outside of the package.
- ▶ Define published prototypes of routines without routine bodies in packages and later implement the routine bodies using the routine prototype.
- ▶ Initialize the package by executing the package initialization procedure for the package. This procedure can include SQL statements and SQL PL statements, and can be used to set default values for global variables or to open cursors.
- ▶ Reference objects defined in the package from within the package and from outside of the package by using the package name as a qualifier (two-part name support) or a combination of the package name and schema name as qualifiers (three-part name support).
- ▶ Drop objects defined within the package.
- ▶ Drop the package.
- ▶ Manage who can reference objects in a package by allowing you to grant and revoke the EXECUTE privilege for the package.

User defined packages

You can create user defined packages in DB2 with the same syntax you would use in Oracle. As expected, these packages will have the same structure and requirements. Similar to Oracle, DB2 user defined packages have extensions of schemas that provide name space support for the objects that they reference. They are repositories in which executable code can be defined. Using a package involves referencing or executing objects that are defined in the package specification and implemented within the package.

You can use the CREATE OR REPLACE PACKAGE syntax to create a package specification, which defines the interface to a package. Creating a package specification enables you to encapsulate related database objects, such as type, procedure, and function definitions, within a single context in the database. A package specification establishes which package objects can be referenced from outside of the package (known as public elements of that package). Similar to Oracle, you can refer to any of the public objects defined in the package specification (variable, constant, exception, function, and procedure) with the three part name qualifier:

```
<schema_name>.<package_name>.<object_name>
```

A package body contains the implementation of all of the procedures and functions that are declared within the package specification. The CREATE PACKAGE BODY statement creates a package body that contains the implementation of all of the procedures and functions that are declared within the package specification, as well as any declaration of private types, variables, and cursors.

Synonyms can be created in packages the same way as in any other database object.

An example of basic package, called c, is available in Appendix C, “Built-in packages” on page 249. This PACKAGE example demonstrates some of the traditional PL/SQL functionality you frequently see in Oracle user defined packages, such as creating a package specification and body, a procedures definition within package, a definition of associative array, anchor data types %TYPE and %ROWTYPE, exception handling, cursor definition, DBMS_OUTPUT built-in package, and others.

Support for built-in packages

DB2 provides support for some of Oracle's most widely used built-in packages through the new system-defined modules (built-in packages). These packages include routines and procedures that simplify the procedural and application logic by providing enhanced capabilities for communicating through messages and alerts, for creating, scheduling, and managing jobs, for operating on large objects, for executing dynamic SQL, for working with files on the database server file system, and for sending e-mail.

Table 2-10 on page 89 provides a list of these packages and their descriptions.

Table 2-10 New built-in packages

Package (module)	Description
DBMS_ALERT	Provides a set of procedures for registering alerts, sending alerts, and receiving alerts.
DBMS_JOB	Provides a set of procedures for creating, scheduling, and managing jobs. DBMS_JOB is an alternate interface for the Administrative Task Scheduler (ATS).
DBMS_LOB	Provides a set of routines for operating on large objects.
DBMS_OUTPUT	Provides a set of procedures for putting messages (lines of text) in a message buffer and getting messages from the message buffer within a single session. These procedures are useful during application debugging when you need to write messages to standard output.
DBMS_PIPE	Provides a set of routines for sending messages through a pipe within or between sessions that are connected to the same database.
DBMS_SQL	Provides a set of procedures for executing dynamic SQL.
DBMS_UTILITY	Provides a set of utility routines.
UTL_DIR	Provides a set of routines for maintaining directory aliases that are used with the UTL_FILE package.
UTL_FILE	Provides a set of routines for reading from and writing to files on the database server file system.
UTL_MAIL	Provides a set of procedures for sending e-mail.
UTL_SMTP	Provides a set of routines for sending e-mail using the Simple Mail Transfer Protocol (SMTP).

Detailed references about these packages, their method, and some examples are provided in Appendix C, “Built-in packages” on page 249. More detailed information about the built-in packages (system-defined module) is available in the DB2 Information Center, found at:

<http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/topic/com.ibm.db2.luw.sql.rtn.doc/doc/c0053670.html>

2.1.11 Moving triggers

A trigger is a database object that consists of set of SQL statements, automatically executed when a specified action occurs. A trigger is associated with a specific table and defines a set of actions within the trigger construct (consisting of SQL or PL/SQL statements) that are triggered in response to an SQL INSERT, DELETE, or UPDATE operation on the specified table.

The trigger functionality is supported in both Oracle and DB2. In both databases, triggers can be used for wide variety of actions, for example, updates to other tables, automatically generating or transforming values for inserted or updated rows, or invoking functions to perform tasks, such as issuing alerts.

Triggers can be fired once FOR EACH ROW or once FOR EACH STATEMENT and BEFORE, INSTEAD OF, or AFTER the triggered operation occurs. In PL/SQL, DB2 supports FOR EACH ROW BEFORE and AFTER triggers. DB2 does not allow updates table to be performed from within a BEFORE trigger. In most cases, AFTER triggers could be used instead to perform these actions.

INSTEAD OF as well as FOR EACH STATEMENT triggers are available in DB2 when written with the SQL PL syntax.

Oracle allows you to define a single trigger that can contain triggered actions for INSERT/DELETE/UPDATE on the table. This is known as multi-action trigger. In DB2, this multi-action trigger has to be separated into separate triggers (one for each action).

Example 2-69 on page 91 shows an Oracle multi-action trigger with INSERT, DELETE, and UPDATE actions that synchronize the entries in the EMPLOYEES and DEPARTMENTS table when the data changes in the EMPLOYEES table. Note that DB2 supports the SELECT on the same table on which the trigger is defined, which Oracle does not.

Example 2-69 Multi-action trigger

```
CREATE OR REPLACE TRIGGER Update_Departments
  AFTER INSERT OR DELETE OR UPDATE ON employees FOR EACH ROW

DECLARE
  INS integer:=0;
  DEL integer:=0;
  UPD integer:=0;

BEGIN
  IF DELETING THEN
    UPDATE departments
      SET (total_projects, total_employees)=
        (SELECT count(1), SUM(current_projects) FROM employees)
      WHERE dept_code=:old.dept_code;
    DEL := 1;
  ELSIF INSERTING THEN
    UPDATE departments
      SET (total_projects, total_employees)=
        (SELECT count(1), SUM(current_projects) FROM employees)
      WHERE dept_code=:new.dept_code;
    INS := 1;
  ELSIF UPDATING THEN
    UPDATE departments
      SET (total_projects, total_employees)=
        (SELECT count(1), SUM(current_projects) FROM employees)
      WHERE dept_code IN (:old.dept_code, :new.dept_code);
    UPD := 1;
  END IF;

  IF ( DEL = 1 ) THEN
    INSERT INTO logged_table VALUES(SYSDATE, 'table row delete');
  ELSIF ( INS = 1 ) then
    INSERT INTO logged_table VALUES(SYSDATE, 'table row insert');
  ELSIF ( UPD = 1 ) then
    INSERT INTO logged_table VALUES(SYSDATE, 'table row update');
  END IF;

END Update_Departments;
/
```

Example 2-70 demonstrates how to separate the multi-action trigger from Example 2-104 into three separate triggers (one for each action of INSERT, DELETE, and UPDATE).

Often, inside of the trigger, there are common statements located outside of the DELETING, INSERTING, and UPDATING blocks. In this case, it is necessary to add these common statements to each of the separated triggers. As a best practice, we recommend extracting complex logic from a trigger, placing it in a procedure, and invoking the procedure from the trigger.

Example 2-70 Three DB2 triggers corresponding to a multi-action trigger

```
CREATE OR REPLACE TRIGGER Update_Departments_I
  AFTER INSERT ON employees FOR EACH ROW

DECLARE
BEGIN
  UPDATE departments
    SET (total_projects, total_employees)=
      (SELECT count(1), SUM(current_projects) FROM employees)
    WHERE dept_code=:new.dept_code;
  INSERT INTO logged_table VALUES(SYSDATE, 'table row insert');
END Update_Departments_I;
/

CREATE OR REPLACE TRIGGER Update_Departments_D
  AFTER DELETE ON employees FOR EACH ROW

DECLARE
BEGIN
  UPDATE departments
    SET (total_projects, total_employees)=
      (SELECT count(1), SUM(current_projects) FROM employees)
    WHERE dept_code=:old.dept_code;
  INSERT INTO logged_table VALUES(SYSDATE, 'table row delete');
END Update_Departments_D;
/

CREATE OR REPLACE TRIGGER Update_Departments_U
  AFTER UPDATE ON employees FOR EACH ROW

DECLARE
BEGIN
  UPDATE departments
    SET (total_projects, total_employees)=
      (SELECT count(1), SUM(current_projects) FROM employees)
    WHERE dept_code IN (:old.dept_code, :new.dept_code);
  INSERT INTO logged_table VALUES(SYSDATE, 'table row update');
END Update_Departments_U;
```

Oracle supports enabling and disabling of triggers globally using the ALTER TRIGGER statement. In DB2, triggers can either be dropped and recreated instead, or global or package variables can be used to control whether the triggers is to be fired in a specific context.

2.1.12 Moving SQL statements

DB2 provides native support for most of the Oracle SQL syntax. This allows existing Oracle based applications to run on a DB2 database with no or very minimal code changes. In this section, we provide some examples of the most often used SQL syntax.

Truncate table SQL statement

DB2 9.7 now includes a new TRUNCATE statement for quickly deleting all rows from a database table. The difference between the TRUNCATE statement and the DELETE statement is that the TRUNCATE statement cannot be rolled back and the keyword IMMEDIATE is mandatory to indicate this fact. When the DB2_COMPATIBILITY_VECTOR registry variable is set, the IMMEDIATE keyword is no longer required. Example 2-71 shows the usage of TRUNCATE.

Example 2-71 Truncate table

```
CREATE TABLE trunc1(c1 int);
INSERT INTO trunc1 VALUES (1), (2), (3);
SELECT * FROM trunc1;
C1
-----
1
2
3
3 record(s) selected.
TRUNCATE trunc1;
SELECT * FROM trunc1;
C1
-----
0 record(s) selected.
```

Autonomous transaction

DB2 provides support for autonomous transaction, a mechanism that allows you to run a block of statements (or a separate transaction) independently of the outcome of the invoking transaction. This feature is particularly useful when moving applications with autonomous transactions supported by Oracle. DB2 supports `PRAGMA AUTONOMOUS_TRANSACTION` for the outer block of a stored procedure. A procedure that you define with this clause runs within its own session, meaning that the procedure is independent of the calling transaction. If you need separate blocks of a procedure, a trigger, or a function to run autonomously, you should wrap the statements into an autonomous procedure.

Hierarchical queries

Hierarchical queries are a form of recursive query that provides support for retrieving a hierarchy from relational data using a `CONNECT BY` clause, pseudo-columns (`LEVEL`), unary operators (`CONNECT_BY_ROOT`, `PRIOR`), and the `SYS_CONNECT_BY_PATH` scalar function.

`CONNECT BY` recursion uses the same subquery for the seed (start) and the recursive step (connect). This combination provides a concise method of representing recursions, such as reports-to-chains presented in `CONNECT BY` recursion.

The query in Example 2-72 relies on the child-parent relationship between `emp_id` and `emp_mgr_id` in `Employees` table. It returns the ID and the last name for each employee together with their manager's employee ID and the number of people in the reporting chain. This syntax is supported on both Oracle and DB2.

Example 2-72 CONNECT BY recursion

```
SELECT substr(lpad('', level * 2) || emp_id,1,20) AS emp_id,  
       last_name, emp_mgr_id,  
       level as number_in_report_chain  
FROM employees  
START WITH emp_mgr_id IS NULL  
CONNECT BY PRIOR emp_id = emp_mgr_id  
ORDER BY emp_id;
```

Figure 2-3 on page 95 shows the output.

EMP_ID	LAST_NAME	EMP_MGR_ID	NUMBER_IN_REPORT_CHAIN
1	HAAS		1
2	THOMPSON	1	2
3	KWAN	1	2
4	GEYER	1	2
5	STERN	2	3
6	PULASKI	2	3
7	HENDERSON	11	3
8	SPENSER	11	3
9	LUCCHESI	1	2
10	O'CONNELL	2	3
11	QUINTANA	2	3
12	NICHOLLS	5	3
13	ADAMSON	5	3
14	PIANKA	5	3
15	YOSHIMURA	1	2
16	SCOUTTEN	17	3
17	WALKER	17	3
18	BROWN	2	3
19	JONES	6	4
20	LUTZ	6	4
21	JEFFERSON	6	4
22	MARINO	19	4
23	SMITH	19	4
24	JOHNSON	6	4
25	PEREZ	5	3
26	SCHNEIDER	1	2
27	PARKER	2	3
28	SMITH	29	4
29	SETRIGHT	1	2
30	MEHTA	2	3
31	LEE	2	3
32	GOUNOT	7	4

Figure 2-3 Output of a CONNECT BY recursive SQL statement

Example 2-73 presents a hierarchical query that features different hierarchical syntax constructions, such as CONNECT_BY_ROOT and SYS_CONNECT_BY_PATH in combination with START WITH... CONNECT BY PRIOR, which we could execute successfully on both Oracle and DB2.

Example 2-73 Demonstrating CONNECT_BY_ROOT and SYS_CONNECT_BY_PATH

```

SELECT (INITCAP(last_name) || ', ' || INITCAP(first_name)) as employee,
       CONNECT_BY_ROOT (INITCAP( first_name) || ' ' || INITCAP(last_name))
       as top_manager, SYS_CONNECT_BY_PATH ((INITCAP(last_name) || ', ' ||
       INITCAP(first_name)),',' > ') as report_chain
FROM employees
START WITH band = '5'
CONNECT BY PRIOR emp_id = emp_mgr_id
ORDER BY employee;

```

The query returns the report chain of the employees in the EMPLOYEES table, as shown in Figure 2-4.

EMPLOYEE	TOP_MANAGER	REPORT_CHAIN
Adamson, Bruce	Christine Haas	> Haas, Christine > Geyer, John > Adamson, Bruce
Brown, David	Christine Haas	> Haas, Christine > Thompson, Michael > Brown, David
Geyer, John	Christine Haas	> Haas, Christine > Geyer, John
Gounot, Jason	Christine Haas	> Haas, Christine > Thompson, Michael > Pulaski, Eva > Gounot, Jason
Haas, Christine	Christine Haas	> Haas, Christine
Henderson, Eileen	Christine Haas	> Haas, Christine > Lucchessi, Vincenzo > Henderson, Eileen
Jefferson, James	Christine Haas	> Haas, Christine > Thompson, Michael > Stern, Irving > Jefferson, James
Johnson, Sybil	Christine Haas	> Haas, Christine > Thompson, Michael > Stern, Irving > Johnson, Sybil
Jones, William	Christine Haas	> Haas, Christine > Thompson, Michael > Stern, Irving > Jones, William
Kwan, Sally	Christine Haas	> Haas, Christine > Kwan, Sally
Lee, Wing	Christine Haas	> Haas, Christine > Thompson, Michael > Lee, Wing
Lucchessi, Vincenzo	Christine Haas	> Haas, Christine > Lucchessi, Vincenzo
Lutz, Jennifer	Christine Haas	> Haas, Christine > Thompson, Michael > Stern, Irving > Lutz, Jennifer
Marino, Salvatore	Christine Haas	> Haas, Christine > Yoshimura, Masatoshi > Walker, James > Marino, Salvatore
Mehta, Ramlal	Christine Haas	> Haas, Christine > Thompson, Michael > Mehta, Ramlal
Nicholls, Heather	Christine Haas	> Haas, Christine > Geyer, John > Nicholls, Heather
O'Connell, Sean	Christine Haas	> Haas, Christine > Thompson, Michael > O'Connell, Sean
Parker, John	Christine Haas	> Haas, Christine > Thompson, Michael > Parker, John
Perez, Maria	Christine Haas	> Haas, Christine > Geyer, John > Perez, Maria
Pianka, Elizabeth	Christine Haas	> Haas, Christine > Geyer, John > Pianka, Elizabeth
Pulaski, Eva	Christine Haas	> Haas, Christine > Thompson, Michael > Pulaski, Eva
Quintana, Delores	Christine Haas	> Haas, Christine > Thompson, Michael > Quintana, Delores
Schneider, Ethel	Christine Haas	> Haas, Christine > Schneider, Ethel
Scoutten, Marilyn	Christine Haas	> Haas, Christine > Yoshimura, Masatoshi > Scoutten, Marilyn
Setright, Maude	Christine Haas	> Haas, Christine > Setright, Maude
Smith, Daniel	Christine Haas	> Haas, Christine > Yoshimura, Masatoshi > Walker, James > Smith, Daniel
Smith, Philip	Christine Haas	> Haas, Christine > Thompson, Michael > Parker, John > Smith, Philip
Spenser, Theodore	Christine Haas	> Haas, Christine > Lucchessi, Vincenzo > Spenser, Theodore
Stern, Irving	Christine Haas	> Haas, Christine > Thompson, Michael > Stern, Irving
Thompson, Michael	Christine Haas	> Haas, Christine > Thompson, Michael
Walker, James	Christine Haas	> Haas, Christine > Yoshimura, Masatoshi > Walker, James
Yoshimura, Masatoshi	Christine Haas	> Haas, Christine > Yoshimura, Masatoshi

Figure 2-4 Output of query with CONNECT_BY_ROOT and SYS_CONNECT_BY_PATH

ROWNUM

Oracle uses the ROWNUM pseudo-column to control the number of rows returned from an SQL statement.

DB2 supports the same exact syntax. The following statement runs in both Oracle and DB2:

```
SELECT * FROM employees WHERE ROWNUM < 10 ;
```

Additionally, DB2 has its own syntax to achieve the same task: the number of rows to read is determined with the FETCH FIRST n ROWS ONLY statement:

```
SELECT * FROM employees FETCH FIRST 9 ROWS ONLY;
```

This DB2 syntax is equivalent (synonym) to the ROWNUM example above.

UPDATE and DELETE statements can also be executed on both Oracle and DB2, as shown in Example 2-74. The example updates the office location for five employees' records from the EMPLOYEES table, which are older than one year and belong to department D11. The DELETE statement deletes five employees' records with office location 5.

Example 2-74 UPDATE and DELETE statements utilizing ROWNUM

```
UPDATE employees
SET office_id = 5
WHERE create_date < SYSDATE - 365
AND dept_code = 'D11'
AND ROWNUM <= 5 ;
```

```
DELETE FROM employees
WHERE office_id = 5
AND ROWNUM <= 5;
```

Row identifier (ROWID)

Each row in an Oracle database has a unique row identifier, or ROWID, that contains the physical address of a row in a database and uniquely identifies this row. This value is stored with the row and does not change over the life of the row in the table, until a table reorganization occurs, which can physically change the row location. ROWID is known as a pseudo-column and is used internally by the database to access the row; it is known to be the fastest way to access a single row in the database.

You can use the ROWID in the SELECT and WHERE clause of a query, but you cannot manipulate (insert, update, or delete) a value of the ROWID pseudo-column. When you describe a table using the DESCRIBE command, ROWID does not appear.

DB2 provides support for this qualifier. In Figure 2-5, we display the ROWID from the EMPLOYEE table retrieved with the following query:

Status	Result1	
	ROWID_VALUE	EMP_ID
1	040000000000000000000000185cfb0a0000	1
2	0500000000000000000000000185cfb0a0000	2
3	0600000000000000000000000185cfb0a0000	3
4	0700000000000000000000000185cfb0a0000	5
5	0800000000000000000000000185cfb0a0000	6
6	0900000000000000000000000185cfb0a0000	7
7	0a00000000000000000000000185cfb0a0000	9
8	0b00000000000000000000000185cfb0a0000	10
9	0c00000000000000000000000185cfb0a0000	11
10	0d00000000000000000000000185cfb0a0000	12
11	0e00000000000000000000000185cfb0a0000	13
12	0f00000000000000000000000185cfb0a0000	14
13	1000000000000000000000000185cfb0a0000	15
14	1100000000000000000000000185cfb0a0000	16
15	1200000000000000000000000185cfb0a0000	17
16	1300000000000000000000000185cfb0a0000	18
17	1400000000000000000000000185cfb0a0000	19
18	1500000000000000000000000185cfb0a0000	20
19	1600000000000000000000000185cfb0a0000	21
20	1700000000000000000000000185cfb0a0000	22
21	1800000000000000000000000185cfb0a0000	23
22	1900000000000000000000000185cfb0a0000	24
23	1a00000000000000000000000185cfb0a0000	25
24	1b00000000000000000000000185cfb0a0000	26
25	1c00000000000000000000000185cfb0a0000	27
26	1d00000000000000000000000185cfb0a0000	28

ROWID is often used in the procedural language to speed up the row access in high volume insert, update, and delete operations. With the current support of ROWID in DB2, the logic will work without changes. If you want to store a ROWID in a PL/SQL variable, you must create the following DB2 user distinct types:

Example 2-75 shows a way to store a ROWID in a PL/SQL variable.

```
CREATE DISTINCT TYPE ROWID AS VARCHAR(16) FOR BIT DATA WITH COMPARISONS;
```

```
END;  
/  

```

Outer join operator

Both Oracle and DB2 support the ANSI SQL syntax for three types of outer join (right, left, and full), as well as the left and right outer join (+) operator. When moving applications from Oracle to DB2, the outer join operator (+) could be used interchangeably with the ANSI SQL outer join syntax.

The outer join operator can only be specified within predicates of the WHERE clause on columns that are associated with table references specified in the FROM clause of the same sub-select.

In Example 2-76, the right outer join properly skips the records that have no corresponding entries, which demonstrates the outer join (+) operator syntax.

Example 2-76 Outer join

```
SELECT e.emp_id, e.first_name, e.last_name, e.dept_code,  
NVL(d.dept_name, 'Unassigned or unknown Department') as department  
FROM  
    employees e,  
    departments d  
WHERE  
    e.dept_code = d.dept_code (+)  
ORDER BY department DESC, emp_id asc;
```

Figure 2-6 shows the output.

EMP_ID	FIRST_NAME	LAST_NAME	DEPT_CODE	DEPARTMENT
31	MAUDE	SETRIGHT	NEW	Unassigned or unknown Department
34	JASON	GOUNOT	NEW	Unassigned or unknown Department
10	THEODORE	SPENSER	E21	SUPPORT SERVICES
32	RAMLAL	MEHTA	E21	SUPPORT SERVICES
33	WING	LEE	E21	SUPPORT SERVICES
20,033	HELENA	WONG	E21	SUPPORT SERVICES
20,034	ROY	ALONZO	E21	SUPPORT SERVICES
9	EILEEN	HENDERSON	E11	SPIFFY COMPUTER SERVICE DIV.
28	ETHEL	SCHNEIDER	E11	SPIFFY COMPUTER SERVICE DIV.
29	JOHN	PARKER	E11	SPIFFY COMPUTER SERVICE DIV.
30	PHILIP	SMITH	E11	SPIFFY COMPUTER SERVICE DIV.
20,028	EILEEN	SCHWARTZ	E11	SPIFFY COMPUTER SERVICE DIV.
20,031	MICHELLE	SPRINGER	E11	SPIFFY COMPUTER SERVICE DIV.
5	JOHN	GEYER	E01	SOFTWARE SUPPORT
7	EVA	PULASKI	D21	PLANNING
23	JAMES	JEFFERSON	D21	PLANNING
24	SALVATORE	MARINO	D21	PLANNING
25	DANIEL	SMITH	D21	PLANNING
26	SYBIL	JOHNSON	D21	PLANNING
27	MARIA	PEREZ	D21	PLANNING
20,024	ROBERT	MONTEVERDE	D21	PLANNING
6	IRVING	STERN	D11	OPERATIONS

Figure 2-6 Outer join query output

Be aware of an invalid cycle! A cycle is formed across multiple joins when the chain of predicates reference back to an earlier table reference. In Example 2-77, T1 is the outer table in the first predicate and later, in the third predicate, there is a circular reference back to T1. Although, T2 is referenced twice in both first and second predicates, this usage is not itself a cycle.

Example 2-77 Demonstration of valid and invalid cycles

```
SELECT * FROM T1,T2,T3
  WHERE T1.a1 = T2.b2(+)    -- T2 - OK
    AND T2.b2 = T3.c3(+)    -- T2 - OK
    AND T3.c3 = T1.a1(+)    -- Be aware of invalid cycle on T1 - Not allowed!
```

Select from DUAL

Oracle provides a dummy table called DUAL, which is frequently used to retrieve system information. Although DB2 has its own equivalent called SYSIBM.SYSDUMMY1, support for calls to DUAL has been provided. The following statement retrieves the same values on both Oracle and DB2:

```
SELECT SYSDATE AS CURRENT_DATE_TIME FROM DUAL;
```

Oracle optimizer hints

Oracle provides optimizer hints for controlling the optimizer's behavior. Due to the fundamental differences in the two optimizers, the Oracle optimizer hints given in the SQL queries are not applicable to DB2. To simplify the enablement process, DB2 will ignore the hints and no changes to the code are necessary.

DB2 optimizer is one of the most sophisticated cost-based optimizers in the industry. Directly influencing the optimizer is usually a rare case. For more information about providing explicit optimization guidelines to the DB2 optimizer, review the optimizer profiles and guidelines topic and related concepts found at:

<http://publib.boulder.ibm.com/infocenter/db2luw/v9/index.jsp?topic=/com.ibm.db2.udb.admin.doc/doc/c0024522.htm>

TRUNCATE as an SQL statement

As with Oracle, DB2 9.7 includes a new TRUNCATE statement that you can use to quickly delete all rows from a database table. Remember that unlike the DELETE statement, the TRUNCATE statement cannot be rolled back, which corresponds to the support provided by Oracle, Sybase, and Microsoft SQL Server. We provide a detailed discussion and an example in “Truncate table SQL statement” on page 93.

SELECT INTO statement with FOR UPDATE clause

DB2 supports the FOR UPDATE clause in the SELECT INTO statement to lock the selected row for later update, as in Oracle. The SELECT INTO statement produces a result table consisting of at most one row, and assigns the values in that row to host variables. The FOR UPDATE specifies that the selected row from the underlying table will be locked to facilitate updating the row later in the transaction, similar to the locking done for the SELECT statement of a cursor, which includes the FOR UPDATE clause. Example 2-78 shows an SELECT INTO with FOR UPDATE example.

Example 2-78 SELECT INTO FOR UPDATE

```
DECLARE
    empid_var employees.empid%TYPE;
    new_lastname employees.lastname%TYPE;
    name_var employees.lastname%TYPE;
BEGIN
    empid_var := 1000;
    new_lastname := 'NEW_NAME';
    --- changing the lastname of employee because of the marital status changed
    --- empid is a unique key

    SELECT lastname INTO name_var FROM employees
    WHERE empid = empid_var FOR UPDATE OF lastname;

    UPDATE employees SET lastname = new_lastname
    WHERE empid = empid_var;
END;
```

2.2 Schema compatibility features

In this section, we provide an overview of the new functionality introduced in DB2 to facilitate seamless schema enablement as well as some enablement references to help build a better understanding about the corresponding functionality.

2.2.1 Extended data type support

When providing extended data type support, DB2 allows us to simply create tables in DB2 using Oracle DDL without changing table structures. Now you can use the NUMBER and VARCHAR2 data types. You can also have the database manager interpret the DATE data type (normally comprised of year, month, and day) as a TIMESTAMP(0) data type (comprised of year, month, day, hour, minute, and second). Almost all of the corresponding Oracle-compatible functions for manipulating these data types and performing data type arithmetic on the DATE data type are also supported.

We can now run Oracle DDL to create tables directly in DB2, with the exception of only a few data types, and it still requires proper mapping to the appropriate DB2 data types. For example, depending on the application requirements for number (38) data type, we may chose to change it to a big integer, decfloat (34), decimal (14.0), or simply to number (31).

For more information about the new data types, refer to the DB2 Information Center for the following topics:

- ▶ VARCHAR2 data type, found at:

<http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/topic/com.ibm.db2.luw.apdv.porting.doc/doc/r0052880.html>

- ▶ NUMBER data type, found at:

<http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/topic/com.ibm.db2.luw.apdv.porting.doc/doc/r0052879.html>

- ▶ DATE data type based on TIMESTAMP(0), found at:

<http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/topic/com.ibm.db2.luw.apdv.porting.doc/doc/r0053667.html>

2.2.2 Flexible schema changes in DB2

DB2 introduces great flexibility when making schema changes. The new features are designed to increase the availability, minimize the database downtime, and simplify the administration tasks when schema changes are needed.

The new features include:

- ▶ Increasing maximum sizes for the length of DB2 identifier names for schema objects.
- ▶ Immediately altering table definitions (extended support for the ALTER TABLE statement and specifically for the ALTER COLUMN SET DATA TYPE option).

- ▶ On-the-fly replacement (CREATE OR REPLACE) for views, sequences, routines, and packages in any order.
- ▶ Introducing the CREATE WITH ERRORS option.
- ▶ Revalidate database schema objects and routines with automatic revalidation on first use or through single service routine invocation.

Maximum length of DB2 identifier names for schema objects

The identifier names of a DB2 schema object, such as index, constraints, and so on, have been increased to 128 characters. With this size increase, changing the object name during conversion is no longer required.

Alter table statement

By providing features, such as immediate alteration of the table definitions and extended support for the ALTER TABLE statement (specifically, the ALTER COLUMN SET DATA TYPE option), DB2 improves availability and simplifies administration by allowing you to make changes to the database with a minimal or no outage. This functionality, sometimes referenced as “schema evolution”, includes many features, such as the ability to change column data types, rename a column or index, add a default, and so on.

Alter objects

The alter objects options, such as CREATE OR REPLACE and REVALIDATE, greatly simplify the process of altering database objects.

CREATE OR REPLACE allows on-the-fly replacement for definitions of views, sequences, routines, and packages, which could be altered in any order and subsequently be revalidated automatically the first time they are used. This is especially important when there are dependencies of the database objects that were altered. For example, if a function selects data from a view that is built on a table we alter, both the view and the function definition will be invalidated when the change in the table definition occurs. DB2 will revalidate automatically both the function and the view the first time when they are called.

Of course, you could also revalidate the database objects manually by executing a single database procedure named ADMIN_REVALIDATE_DB_OBJECTS. In the example below, all objects in the schema called MY_SCHEMA will be revalidated:

```
CALL ADMIN_REVALIDATE_DB_OBJECTS ( NULL, 'MY_SCHEMA', NULL);
```

To see different options of this procedure, refer to DB2 Information Center, found at:

<http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/index.jsp?topic=/com.ibm.db2.luw.sql.rtn.doc/doc/r0053626.html>

The revalidation option depends on a database configuration parameter called `AUTO_REVAL`, which is described in 2.1.1, “SQL compatibility setup” on page 23.

2.2.3 Sequences

The sequences in Oracle and DB2 have the same definition and syntax. The enablement process requires no manual changes to the CREATE SEQUENCE statements. In addition to the sequence functionality, DB2 provides an option for the user to utilize the identity column functionality when there is a need to automatically generate values for the table.

Sequence characteristics

The characteristics of the sequence include:

- ▶ Sequences are not tied to any one table.
- ▶ Sequences generate sequential values that can be used in any SQL or XQuery statement.
- ▶ Because sequences can be used by any application, there are two expressions used to control the retrieval of the next value in the specified sequence and the value generated previous to the statement being executed. The PREVIOUS VALUE expression returns the most recently generated value for the specified sequence for a previous statement within the current session. The NEXT VALUE expression returns the next value for the specified sequence. The use of these expressions allows the same value to be used across several SQL and XQuery statements within several tables.

After creating a new sequence called `EMPLOYEE_SEQUENCE`, Example 2-79 shows a few typical PL/SQL code calls to this sequence. This code can be run in both Oracle and DB2.

Example 2-79 Sequence

[illegible]


```
EXECUTE IMMEDIATE 'INSERT INTO employees(emp_id, first_name, last_name,
current_projects, emp_mgr_id, dept_code, acct_id, office_id, band,
create_date) VALUES ( ' || v_EmployeeId || ', UPPER('' || p_FirstName ||
''), UPPER('' || p_LastName || ''), 0, ' || p_EmpMgrId || ', '' ||
p_DeptCode || ' ', ' || p_Account || ', ' || p_OfficeId || ', 1, '' ||
p_CreateDate || '')';

...
DBMS_OUTPUT.PUT_LINE('Employee record id ' || v_EmployeeId || '
was created successfully.');
```

For a complete sequence usage example, refer to the `ADD_NEW_EMPLOYEE` procedure in our test case procedure shown in Appendix C, “Built-in packages” on page 249.

2.2.4 Index enablement

Indexes play an important role in the efficient data retrieval alternative to the sequential table scan. Most of the indexes from the Oracle database can be deployed straight to DB2. Here we discuss some of the similarities and the differences in the index implementation in Oracle and DB2.

Include column index

DB2's INCLUDE column index is a concept that is also available on some other database systems. When creating a unique index, you have the option to include extra columns to the index using the INCLUDE clause. The INCLUDE columns are stored with the index, but will not be sorted and considered for uniqueness. Using the INCLUDE columns improves the performance of data retrieval when index access is involved, because DB2 can retrieve data directly from the index page instead of the data page.

Example 2-80 shows creating an index with INCLUDE columns.

Example 2-80 Creating an index with INCLUDE columns

```
CREATE UNIQUE INDEX ix1 ON employee
(name ASC) INCLUDE (dept, mgr, salary, years)
```

Clustered index support

Although both Oracle and DB2 have clustered indexes, the term “cluster” in relation to indexes has a different meaning in each database.

In Oracle, a cluster index means an index on a clustered or partitioned table.

In DB2, if an index is created with the CLUSTER option, the index provides the table clustering. In other words, with the clustering index, the data in the table is rearranged in the same order as that of the index.

The DB2 clustering index provides performance enhancements when a query scans most of the data in the same order as that of the index. When a new row is inserted, an attempt is made to keep the new row physically close to rows that have key values logically closed in the index-key sequence. For each table, there can only be one clustered index, because a table can only be in the same physical order as one index. Example 2-81 provides an example statement used to create a clustering index.

Example 2-81 How to create a clustered index in DB2

```
CREATE INDEX inxcls_emp_empno
ON employee (empno ASC)
CLUSTER
PCTFREE 10
MINPCTUSED 40;
```

Bitmap indexes

Support for the Oracle bitmap index is not available in DB2. This type of index is aimed at data warehousing and is suitable for an index where there are very few key values (low cardinality), for example, gender or state. In DB2, this type of index is not required, as the DB2 optimizer may create dynamic bitmap indexes during the execution of certain types of queries (if needed).

Functional based indexes

An Oracle function-based index computes the value of the function or expression and stores it in the index. DB2 provides corresponding functionality by creating a computed column to hold the generated values for the function expression, and then creates an index on this column. Example 2-82 demonstrates how to enable a search on UPPER names by adding new columns for first, last, and middle name and two composite indexes to speed up the search.

Example 2-82 Create generated columns in DB2

```
SET INTEGRITY FOR NAME OFF;

ALTER TABLE name ADD COLUMN upper_first_name
GENERATED ALWAYS AS ( UPPER(first_name) );
ALTER TABLE name ADD COLUMN upper_last_name
GENERATED ALWAYS AS ( UPPER(last_name) );
ALTER TABLE name ADD COLUMN upper_mid_name
GENERATED ALWAYS AS ( UPPER(mid_name) );
```

```
SET INTEGRITY FOR name IMMEDIATE CHECKED FORCE GENERATED;  
  
CREATE INDEX ix_up_name_1 ON name  
    (upper_first_name ASC,  
     upper_mid_name ASC) ;  
  
CREATE INDEX ix_up_name_2 ON name  
    (upper_last_name ASC,  
     upper_mid_name ASC) ;
```

Partitioned table indexes

In DB2, the indexes created on partitioned tables can be both local and global. For more details about these indexes, refer to “Indexes on partitioned tables” on page 123.

2.2.5 Constraints enablement

Both DB2 and Oracle support the same type of constraints, such as primary and foreign keys, unique, NOT NULL, and check constraints. Some of these constraints, such as primary keys and check constraints, are identical in both databases; others have specifics. For example, all columns specified in a unique constraint in DB2 must be defined as NOT NULL, while Oracle allows NULL values in a unique constraint. A unique index could be used in this case to provide corresponding functionality.

In Oracle, you are not required to add a NOT NULL attribute to a column in a table in order to define a primary key that includes the column. In DB2, you must explicitly specify the NOT NULL attribute if a primary key is to be defined on that column.

To enforce the referential integrity, both DB2 and Oracle support foreign key constraints, in which a parent table's primary key is referenced by the child table. DB2 additionally allows you to enforce the referential integrity for any unique constraints (not just for the primary key). This is especially useful when the unique constraint is a composite key and it could automatically apply more complex dependencies between the tables.

When we need to perform operations, such as loading data into a table, altering a table by adding constraints, or adding a generated column, we usually want to temporarily disable the constraints, complete the operation, and subsequently revalidate and enable the constraints. You can use the DB2 SET INTEGRITY mechanism to perform this manual integrity processing, as shown in Example 2-83.

Example 2-83 How to use SET INTEGRITY in DB2

```
db2 set integrity for Table1 off;  
-- perform the desired operation:  
-- DB2 Load, Alter Table to add constraints, etc.  
db2 set integrity for Table1 immediate checked;
```

There are many different options related to the SET INTEGRITY statement, which can be found in the DB2 Information Center, found at:

<http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/topic/com.ibm.db2.luw.admin.gui.doc/doc/t0023136.html>

DB2 offers one more type of constraint known as the informational constraint, which is a constraint attribute that can be used by the SQL compiler to improve access to data. Informational constraints are not enforced by the database manager, and are not used for additional verification of data; rather, they are used to improve query performance.

2.2.6 Created temporary tables

Oracle supports the concept of global temporary tables, which correspond to the created global temporary tables in DB2, thus providing straight compatibility between the two databases.

In addition, DB2 also provides declared global temporary tables, which differ from the created global temporary tables because of their persistent level.

Created global temporary tables are a new type of user-defined temporary table introduced in DB2 9.7. An application session can now use a created temporary table to store intermediate result sets for manipulation or repeated references without interfering with concurrently running applications. The definition of a created temporary table is stored persistently in the DB2 catalog, which allows this definition to be shared across all concurrent sessions. Any connection can refer to a created temporary table at any time without the need for a setup script to initialize the created temporary table. The content of a created temporary table is always kept private to each session; a connection can access only the rows that it inserts.

The persistent storage of the created global temporary table definition results in the following capabilities that are also common to Oracle:

- ▶ After an application session defines a created temporary table, concurrently running sessions do not have to redefine it.
- ▶ You can reference a created temporary table in SQL functions, triggers, and views in combinations of temporary and permanent tables.
- ▶ If the TRUNCATE statement is issued against a temporary table, it only truncates the session specific and has no effect on the data of other sessions.
- ▶ Indexes can be created on temporary tables.

Example 2-84 shows how to create a created global temporary table in DB2 with the same syntax as in Oracle.

Example 2-84 Create a created global temporary table

```
CREATE USER TEMPORARY TABLESPACE user_temp;
CREATE GLOBAL TEMPORARY TABLE Employees_Temp_table
  (Employee_number  NUMBER,
   Employee_name    VARCHAR2(250),
   Department       VARCHAR2(3))
ON COMMIT PRESERVE ROWS;
```

The same temporary table is created in Example 2-85 using the DB2 LIKE syntax. The results of inserting into and selecting from the global temporary table are the same in both examples.

Example 2-85 Using created global temporary tables

```
CREATE USER TEMPORARY TABLESPACE user_temp;

CREATE GLOBAL TEMPORARY TABLE Employees_Temp_table
LIKE employees
ON COMMIT PRESERVE ROWS;

INSERT INTO Employees_Temp_table
  SELECT * FROM employees
  WHERE dept_code = 'D21';

SELECT emp_id, first_name, last_name
FROM Employees_Temp_table;
```

The result of this script is shown in Example 2-86.

Example 2-86 Result set: select from a global temp table

EMP_ID	FIRST_NAME	LAST_NAME

7.	EVA	PULASKI
23.	JAMES	JEFFERSON
24.	SALVATORE	MARINO
25.	DANIEL	SMITH
26.	SYBIL	JOHNSON
27.	MARIA	PEREZ
20024.	ROBERT	MONTEVERDE

2.2.7 Synonyms

A synonym in Oracle is an alternative name for a database object, such as a table, view, sequence, procedure, stored function, package, Snapshot, or another synonym. In DB2, the support for this alternative name is also known as an *alias* (and vice versa; the aliases in DB2 are also known as *synonyms*). DB2 recognizes the same syntax as Oracle for creating synonyms on tables (Example 2-87). When creating synonym in DB2 for a package, nickname, sequence, view, or another synonym, you need to specify the type of the database object on which the synonym will be defined. Note the clause “FOR SEQUENCE” in the example below:

```
CREATE PUBLIC SYNONYM sequence_syn FOR SEQUENCE myschema.mysequence;
```

As with Oracle, the aliases (or synonyms) can be PUBLIC (with the schema SYSPUBLIC) or private (with the schema of the CURRENT USER, if the PUBIC keyword is not specified).

Example 2-87 creates a public synonym and an alias for the catalog view SYSCAT.TABLES using two different types of syntax. In the first case, the CREATE SYNONYM syntax is used in the same exact way as in Oracle. The second example demonstrates the CREATE PUBLIC ALIAS syntax that is specific to DB2. Although the syntax is different, the result is the same: two alternative names for SYSCAT.TABLES are created and the SYSCAT.TABLES could be referenced anywhere by either of them.

Example 2-87 Two different ways to create a synonym/alias in DB2

```
CREATE PUBLIC SYNONYM tabs_synonym FOR SYSCAT.TABLES
CREATE PUBLIC ALIAS tabs_alias FOR SYSCAT.TABLES
```

2.2.8 Views and Materialized Views

Views are quite similar in Oracle and DB2, and, depending on the SQL used in the view definition, most of the CREATE VIEW (CREATE OR REPLACE VIEW) statements could be run against DB2 without changes. The view named `organization_structure`, shown in Example 2-88, compiles on both Oracle and DB2. It utilizes several SQL syntaxes: CREATE OR REPLACE VIEW, recursive SQL statement START WITH ... CONNECT BY, LEVEL keyword, COALESCE, INITCAP, and NVL scalar functions, as well as an outer join syntax of (+). It also contains a CASE statement that conditionally calls two functions from a package named `project_package`.

Example 2-88 Create or replace view example with multiple SQL syntax structures

```
CREATE OR REPLACE VIEW organization_structure
("LEVEL", "FULL_NAME", "DEPARTMENT", "ASSIGNMENTS") AS
SELECT
    LEVEL,
    SUBSTR((LPAD(' ', 4 * LEVEL - 1) || INITCAP(e.last_name) || ' ', '
                                                || INITCAP(e.first_name)), 1, 40),
    NVL(d.dept_name, 'Unknown') ,
    (CASE COALESCE (d.DEPT_CODE, '001')
    WHEN 'L01' THEN project_package.fn_calc_dept_projects ( d.dept_code)
    ELSE project_package.fn_reset_dept_projects ( d.dept_code, d.total_employees)
    END ) as assignments
FROM
    employees e,
    departments d
WHERE
    e.dept_code=d.dept_code(+)
START WITH emp_id = 1 CONNECT BY NOCYCLE PRIOR emp_id = emp_mgr_id;
```

Materialized Views

The concept of a DB2 Materialized Query Table (MQT) is identical to the Materialized View in Oracle, and both are based on caching pre-computed query results as a table that can be later refreshed from the original base table(s) on demand or by schedule. The statement to create a MQT is semantically similar to that of creating an Oracle Materialized View with just a few syntax differences.

The significant performance gain is due to the ability of the optimizers to automatically recognize when an existing Materialized View or MQT could be used to satisfy an incoming query request more efficiently than going to the base tables. Although this mechanism enables more efficient access and saves processing power, the data could become out-of-date if not refreshed periodically. For this reason, both Oracle and DB2 provide a refresh clause with different scheduling options.

To learn more about Materialized Query Tables and see the enhancements in DB2, refer to the DB2 Information Center, found at:

<http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/topic/com.ibm.db2.luw.wn.doc/doc/c0054114.html>

2.2.9 Object types

Structured types are supported on both Oracle and DB2, and they are similar conceptually. A structured type is a user-defined data type containing one or more named attributes or set of method specifications. The attributes are properties that describe the specifics of the structured type and each has a specified data type.

In Example 2-89, the `address_type` consists of attributes such as street, number, city, and state. If methods are created, they allow you to define a behavior for structured types. The methods are routines that extend SQL and are integrated with a particular structured type. In this example, the methods `SAMEZIP` and `DISTANCE` calculate the information specific to `address_type`. To achieve identical functionality in the enablement process, you must use the DB2 syntax.

Using objects types in a PL/SQL context is limited in DB2. Refer to 2.1.2, “New PL/SQL and SQL types” on page 27 for details about what object types are supported in a PL/SQL context.

Example 2-89 Structured type with Oracle and DB2 syntax

```
-- DB2 style
CREATE TYPE address_type AS
(STREET    VARCHAR2(30),
 STREETNUMBER CHAR(15),
 CITY      VARCHAR2(30),
 STATE     VARCHAR2(10))
NOT FINAL
MODE DB2SQL

METHOD SAMEZIP (addr address_type)
RETURNS INTEGER
LANGUAGE SQL
DETERMINISTIC
CONTAINS SQL
NO EXTERNAL ACTION,

METHOD DISTANCE (addr address_type)
RETURNS FLOAT
LANGUAGE C
DETERMINISTIC
PARAMETER STYLE SQL
```



```

NO SQL
NO EXTERNAL ACTION

-- Oracle syntax
CREATE Or Replace TYPE address_type AS OBJECT (
    STREET      VARCHAR2(30),
    STREETNUMBER CHAR(15),
    CITY         VARCHAR2(30),
    STATE        VARCHAR2(10),

    MEMBER FUNCTION SAMEZIP (addr address_type) RETURN INTEGER,
    MEMBER FUNCTION DISTANCE (addr address_type) RETURN FLOAT
);

```

It is also common to see a type hierarchy or types that have nested structured type attributes supported in both Oracle and DB2. Due to the great similarity in concept and for simplicity, Example 2-90 only outlines the DB2 type declarations and does not provide a step by step comparison with Oracle.

Example 2-90 Type hierarchy and nesting of type attributes in DB2

```

-- Create a type hierarchy consisting of a type for employees and a subtype
-- for managers.
CREATE TYPE EMP AS
    (NAME      VARCHAR(32),
     SERIALNUM INT,
     SALARY     DECIMAL(10,2))
MODE DB2SQL

CREATE TYPE MGR UNDER EMP AS
    (BONUS     DECIMAL(10,2))
MODE DB2SQL

-- Create a type that has nested structured type attributes from
-- the type above

CREATE TYPE PROJECT AS
    (PROJ_NAME VARCHAR(20),
     PROJ_ID   INTEGER,
     PROJ_MGR  MGR,
     PROJ_LEAD EMP,
     AVAIL_DATE DATE)
MODE DB2SQL

```

DB2 also provides support for abstract data types similar to Oracle. Example 2-91 shows the creation of a type that is later used as a table column.

Example 2-91 Support of abstract data types in DB2 and Oracle

```
-- DB2 syntax
CREATE TYPE Address_type AS
(StreetNumber VARCHAR (10),
  Street VARCHAR(50),
  City VARCHAR(50),
  Zip VARCHAR (15)
) MODE DB2SQL
/

-- Oracle syntax
create type Address_type AS OBJECT
(  StreetNumber VARCHAR (10),
   street VARCHAR (50),
   city VARCHAR (50),
   zip VARCHAR (15)
);
/
--Create table (same syntax for both Oracle and DB2)

CREATE TABLE customer_with_type
(cust_ID integer,
first_name VARCHAR(50),
last_name VARCHAR(50),
Address Address_type
)
/
--
DESCRIBE TABLE customer_with_type;
```

Column name	Data type schema	Data type name	Column Length	Scale	Nulls
CUST_ID	SYSIBM	INTEGER	4	0	Yes
FIRST_NAME	SYSIBM	VARCHAR	50	0	Yes
LAST_NAME	SYSIBM	VARCHAR	50	0	Yes
ADDRESS	MyUser	ADDRESS_TYPE	0	0	Yes

4 record(s) selected.

2.2.10 Partitioning and MDC

If you are using partitioned tables on Oracle, as you convert to DB2, you must plan for similar table partitioning within DB2. In this section, we introduce various partitioning features provided by DB2 and compare them to the partitioning techniques on Oracle.

In a single database partition, DB2 automatically organizes data on disk by distributing data in a round robin fashion (by extentsize) across all containers of a table space. This method of data organization is the default behavior on DB2 and does not require any type of further definition. However, DB2 can be designed to organize data in other ways as well. These different data organization schemes can be specified at the database or table level.

The data organization methods available on DB2 are:

- ▶ Table partitioning
- ▶ Database partitioning
- ▶ Multidimensional clustering
- ▶ Combined organization schemes

Table partitioning

Table partitioning in DB2 is also referred to as *range partitioning* or *data partitioning*. This data organization scheme is one in which table data is divided across multiple storage objects called data partitions or ranges according to values in one or more table columns. Each data partition is stored separately and can be in different table spaces.

Example 2-92 shows creating table partitioning on DB2. The example demonstrates the use of a “shorthand” notation that automatically generates 24 partitions of uniform size, that is, one partition for each month over a two-year period. Note that the MINVALUE and MAXVALUE catch all values that fall below and above the defined ranges.

Example 2-92 DB2 table partitioning

```
CREATE TABLE orders
(
  l_orderkey DECIMAL(10,0) NOT NULL,
  l_partkey INTEGER,
  l_suppkey INTEGER,
  l_linenummer INTEGER,
  l_quantity DECIMAL(12,2),
  l_extendedprice DECIMAL(12,2),
  l_shipdate DATE
)
PARTITION BY RANGE(l_shipdate)
(STARTING MINVALUE,
 STARTING '1/1/1992' ENDING '12/31/1993' EVERY 1 MONTH,
 ENDING AT MAXVALUE);
```

Example 2-93 illustrates table partitioning using a longer syntax, which is required when the partitioning key is composed of a composite column.

Example 2-93 DB2 table partitioning using manual syntax

```
CREATE TABLE sales
(
  year INT,
  month INT
)
PARTITION BY RANGE (year, month)
(STARTING FROM (2001, 1)
ENDING (2001,3) IN tbsp1,
ENDING (2001,6) IN tbsp2,
ENDING (2001,9) IN tbsp3,
ENDING (2001,12) IN tbsp4,
ENDING (2002,3) IN tbsp5,
ENDING (2002,6) IN tbsp6,
ENDING (2002,9) IN tbsp7,
ENDING AT MAXVALUE );
```

Oracle's range partitioning is conceptually comparable to table partitioning in DB2. The differences between them lay mainly in the syntax used to define how the table is partitioned.

The equivalent code of Example 2-93 in Oracle is the range partition statement shown in Example 2-94.

Example 2-94 Oracle range partitioning

```
CREATE TABLE sales
(
  year int,
  month int
)
PARTITION BY RANGE (year, month)
(PARTITION p1 VALUES LESS THAN (2002,4) tablespace tbsp1,
PARTITION p2 VALUES LESS THAN (2002,7) tablespace tbsp2,
PARTITION p3 VALUES LESS THAN (2002,10) tablespace tbsp3,
PARTITION p4 VALUES LESS THAN (2002,13) tablespace tbsp4,
PARTITION p5 VALUES LESS THAN (2003,4) tablespace tbsp5,
PARTITION p6 VALUES LESS THAN (2003,7) tablespace tbsp6,
PARTITION p7 VALUES LESS THAN (2003,10) tablespace tbsp7,
PARTITION p8 VALUES LESS THAN (MAXVALUE, MAXVALUE) tablespace tbsp8 );
```

Note that a name is given to each Oracle partition in the example; however, naming the partitions is optional on both DB2 and Oracle. If the partition is not explicitly named, a system name is generated by default. To name a partition, use the PART or PARTITION keyword.

Also note that on Oracle each partition contains values less than, and not including, the value that defines that partition. On DB2, the values defined for each partition are included within that partition.

DB2 provides a method of table partitioning that is based on a generated expression of a column. Depending on the situation, table partitioning on a generated column may be used in a similar way as the list partitioning on Oracle.

Example 2-95 shows an Oracle list partitioning.

Example 2-95 Oracle list partitioning

```
CREATE TABLE customer
(
  cust_id int,
  cust_prov varchar2(2)
)
PARTITION BY LIST (cust_prov)
(PARTITION p1 VALUES ('AB', 'MB') tablespace tbsp_ab,
PARTITION p2 VALUES ('BC') tablespace tbsp_bc,
PARTITION p3 VALUES ('SA') tablespace tbsp_mb,
...
PARTITION p13 VALUES ('YT') tablespace tbsp_yt,
PARTITION p14 VALUES(DEFAULT) tablespace tbsp_remainder );
```

Example 2-96 shows how the Oracle list partitioning can be written as a DB2 table partitioning based on a generated column.

Example 2-96 DB2 conversion of Oracle list partition

```
CREATE TABLE customer
(
  cust_id INT,
  cust_prov CHAR(2),
  cust_prov_gen GENERATED ALWAYS AS
  (CASE
  WHEN cust_prov = 'AB' THEN 1
  WHEN cust_prov = 'BC' THEN 2
  WHEN cust_prov = 'MB' THEN 1
  WHEN cust_prov = 'SA' THEN 3
  ...
  WHEN cust_prov = 'YT' THEN 13
  ELSE 14
  END)
)
IN tbsp_ab, tbsp_bc, tbsp_mb, .... tbsp_remainder
PARTITION BY RANGE (cust_prov_gen)
(STARTING 1 ENDING 14 EVERY 1);
```

In Example 2-96, the numeric values are generated based on the values for CUST_PROV. The numeric values populate the generated column CUST_PROV_GEN, on which table partitioning is based.

Database partitioning

On DB2, database partitioning is one of the scalability features used to host a large size database in multiple partitions within and across different physical nodes. This is an optional DB2 feature known as the Database Partitioning Feature (DPF). DPF is mostly used for large, data warehousing applications, although it can be used in some types of OLTP applications as well. It implements a shared nothing architecture in which every partition has its set of resources. When database partitioning is used, the multiple database partitions appear and work together as a single unit. This architecture allows complex data access tasks to run on different parts of data in parallel.

When this feature is enabled, data organization is based on a hashing algorithm that distributes table data across multiple database partitions. Each database partition can reside on a separate partition in a physical or logical machine. Data is hashed according to a distribution key that is either explicitly defined in the table using the DISTRIBUTE BY HASH clause, or defaults to the first qualified column. Ideally, a distribution key is chosen that can hash the table data evenly across all database partitions.

Figure 2-7 shows a table that summarizes the differences between the Oracle and DB2 partitioning methods.

Oracle partitioning	DB2 data organization	Oracle 10g syntax	DB2 9 syntax
No equivalent	Round robin	None	Default: occurs automatically on single partition database
Range partitioning	Table partitioning	PARTITION BY RANGE	PARTITION BY RANGE
Hash partitioning	Database partitioning	PARTITION BY HASH	DISTRIBUTE BY HASH
List partitioning	Table partitioning with generated column	PARTITION BY LIST	PARTITION BY RANGE
Composite partitioning: hash-range hash-list	Combination of: database partitioning, table partitioning, multidimensional clustering	PARTITION BY RANGE, SUBPARTITION BY HASH SUBPARTITION BY LIST	DISTRIBUTE BY HASH PARTITION BY RANGE ORGANIZE BY DIMENSIONS
No equivalent	Multidimensional clustering	None	ORGANIZE BY DIMENSIONS

Figure 2-7 Mapping Oracle data organization schemes to DB2

Example 2-97 is an example of how a table is defined when database partitioning is used.

Example 2-97 Define a table in a partitioned database

```
CREATE TABLE partition_table
(partition_date date NOT NULL,
partition_data VARCHAR(20) NOT NULL
)
IN tbsp_parts
DISTRIBUTE BY HASH (partition_date);
```

The DISTRIBUTE BY HASH clause of a table is only used in a multiple partitioned database environment. To partition data in a single partitioned database environment, table partitioning or multidimensional clustering organization is used. Hash partitioning on Oracle is done in a single database environment.

Example 2-98 shows an example of how hash partitioning syntax on Oracle compares to DB2.

Example 2-98 Oracle hash partitioning

```
CREATE TABLE hash_table
(
  hash_part date,
  hash_data varchar2(20)
)
PARTITION BY HASH(hash_part)
(partition p1 tablespace tbsp1,
partition p2 tablespace tbsp2
);
```

Multidimensional clustering

Multidimensional clustering, also known as an MDC table, is a method of data organization that clusters data together on disk according to multiple dimension key values. A dimension is a key, such as product, time period, or geography, used to group factual data into a meaningful way for a particular application.

A dimension can consist of a composite of two or more columns. A desirable characteristic of dimension values is that they have low cardinality and consist of a minimal number of unique values.

Example 2-99 is an example of an MDC table definition.

Example 2-99 MDC table definition

```
CREATE TABLE sales
(
  store INT NOT NULL,
  sku INT NOT NULL,
  division INT NOT NULL,
  quantity INT NOT NULL
)
ORGANIZE BY DIMENSIONS (store, sku);
```

In Example 2-99, a table is created with the specification that division and quantity is to be organized by two dimensions, STORE and SKU. All data in the table are stored on disk in data blocks organized by STORE and SKU values. Each block on disk contains only rows of data based on a unique set of dimension values.

When dimension keys are used as predicates in the WHERE clause of a SELECT statement, query performance is usually greatly improved because many rows are retrieved with fewer I/Os. In addition, performance benefits are gained from the smaller block index that is used with MDC tables. Because all rows in a block are referenced by the same dimensions, only one index entry per dimension is required to locate all the rows in that block.

Oracle does not have an data organization scheme that is similar to the MDC table.

Combining methods of data organization

Just as Oracle has composite partitioning, a variety of data organization schemes can be combined on DB2.

These combinations are:

- ▶ Database partitioning with a sublevel of table partitioning
- ▶ Database partitioning with a sublevel of MDC data organization
- ▶ Database partitioning with a sublevel of table partitioning followed by a sublevel of MDC data organization
- ▶ Table partitioning with a sublevel of MDC data organization

Example 2-100 shows an example of combining database partitioning, table partitioning, and MDC organization.

Example 2-100 Combining database partitioning, table partitioning, and MDC

```
CREATE TABLE orders
(
  order_id INTEGER,
  ship_date DATE,
  region SMALLINT,
  category SMALLINT
)
IN tbsp1, tbsp2, tbsp3, tbsp4
DISTRIBUTE BY HASH (order_id)
PARTITION BY RANGE (ship_date)
(STARTING FROM ('01-01-2005') ENDING ('12-31-2006') EVERY (1 MONTH))
ORGANIZE BY DIMENSION (region, category);
```

In Example 2-100, the data is distributed over multiple database partitions using a hashed value of ORDER_ID. Within each database partition, the table is partitioned by the SHIP_DATE month, and within each table partition the data is organized in blocks by the dimensions REGION and CATEGORY.

On Oracle, composite partitioning is used to combine the following types of partitioning methods:

- ▶ Range partitioning with hash subpartitioning
- ▶ Range partitioning with list subpartitioning

The composite partitioning on Oracle is used when partitioning by a range alone does not provide enough granularity for managing a partition. On DB2, you can use a composite column as a range partitioning key to break down a table partition into smaller units. The range partition key is defined by the `PARTITION BY RANGE` clause, as shown in Example 2-101.

Example 2-101 Using `PARTITION BY RANGE` clause

```
CREATE TABLE sales
(
  year INT,
  month INT
)
IN tbsp1, tbsp2, tbsp3, tbsp4, tbsp5, tbsp6, tbsp7, tbsp8
PARTITION BY RANGE (year, month) ...
```

If adding a secondary column to the partitioning key is not possible, then use a generated column to complete the composite column.

Roll in and roll out of data

DB2 supports attaching a new partition to an existing partitioned table (roll in) and the detaching of a partitioned table into a single table (roll out). This functionality is achieved by using the `ATTACH PARTITION` and `DETACH PARTITION` clauses of the `ALTER TABLE` statement. By attaching a new partition to a table, you facilitate the adding of a new range of data to a partitioned table. A new partitioned range can be added anywhere in the table, and not only to the high end of the table.

To attach a partition, the data is loaded into a newly created table and then that table is attached to the existing partitioned table. Example 2-102 shows the newly created table `DEC03` that has been loaded with data rolled into the partitioned table `STOCK`.

Example 2-102 Roll in

```
ALTER TABLE stock ATTACH PARTITION dec03
STARTING FROM '12/01/2003' ENDING AT '12/31/2003'
FROM dec03;
COMMIT WORK ;
```

The new table that is attached must match the existing table in several ways, that is, the source and target tables must match in column order and definitions, default values, nullability, compression, and table space types used.

When a source is newly attached, it is offline and remains offline until the SET INTEGRITY statement is executed. The following example shows the SET INTEGRITY statement:

```
SET INTEGRITY FOR stock ALLOW WRITE ACCESS  
IMMEDIATE CHECKED FOR EXCEPTION IN stock USE stock_ex;  
COMMIT WORK;
```

SET INTEGRITY validates the data in the newly attached data partition. The COMMIT WORK is needed to end the transaction and to make the table available for use.

In a similar way, an existing table can have a partition detached into a separate table by using the ALTER statement:

```
ALTER TABLE stock DETACH PART dec01 INTO stock_drop;  
DROP TABLE stock_drop;
```

In addition, partitioned tables may be modified using the ADD PARTITION and DROP PARTITION options of the ALTER TABLE statement. The ADD PARTITION clause is used to add an empty partition with a new range to an existing partitioned table. After it is added, the partitioned table can be loaded with data.

Indexes on partitioned tables

Partitioned tables in both Oracle and DB2 can have indexes that are partitioned (local), non-partitioned (global), or a combination of these two options.

A non-partitioned index is a single index object that refers to all rows in a partitioned table. In DB2, non-partitioned indexes are always created as independent index objects in a single table space, even if the table data partitions span multiple table spaces.

A partitioned index is made up of a set of index partitions, each of which contains the index entries for a single data partition. Each index partition contains references only to data in its corresponding data partition. In DB2, both system-generated and user-generated indexes can be partitioned.

In some situations, such as when performing roll-in operations with partitioned tables, the partitioned indexes are more efficient (less time and resource consuming) than the non-partitioned indexes.

More details about the indexes on partitioned tables and the related topics are available in the DB2 Information Center, found at:

<http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/topic/com.ibm.db2.luw.admin.dbobj.doc/doc/c0055328.html>

2.2.11 Oracle database links and DB2 federation

A table that resides in another database can be accessed as a local table through the features delivered by the database management systems. In Oracle, the database links provide this capability, while in DB2, the Homogeneous Federation Feature delivers the ability to access database objects in different DB2 data servers.

You can have unified access to the data managed by multiple data servers, including DB2 (mainframe and distributed) and the Informix® with DB2 Homogeneous Federation Feature. This allows applications to access and integrate diverse data (mainframe and distributed) as though it were a DB2 table, regardless of where the information resides, while retaining the autonomy and integrity of the data sources. The InfoSphere™ Federation Server adds on the Homogeneous Federation Feature to significantly expand the choice of data sources to any kind of data, including database management systems on various platforms, flat files, Excel, rich media, e-mails, XML, and LDAP. This is an alternative method for moving data from an Oracle database to DB2.

For more information about InfoSphere Federation Server, refer to:

http://www.ibm.com/software/data/integration/support/federation_server/

You can also refer to the DB2 manual *Information Integration: Administration Guide for Federated Systems*, SC19-1020.

The DB2 Homogeneous Federation Feature enables you to:

- ▶ Gain virtualized real-time access to disparate data sources.
- ▶ Speed time to market for new projects.
- ▶ Access more data sources.
- ▶ Extend your data warehouse or data mart with remote data.

Setting up federated databases

Setting up federated databases is simple. We demonstrate the steps by using an example. In this example, we want to join the LOCAL_DEPARTMENT table in the DB2_EMP database with the EMPLOYEE table in database SAMPLE. Because the query will be executed on DB2_EMP, it will be the federated server.

Perform the following steps:

1. Enable the Federation feature.

The Federation feature is enabled by setting the DB2 database manager configuration (DBM CFG) parameter `FEDERATED` to `YES` on the federated server. You can check and set the value as shown in Example 2-103.

Example 2-103 Enabling the Federation feature

```
/WORK # db2 get dbm cfg |grep "Federated Database"
Federated Database System Support (FEDERATED) = NO
/WORK # db2 update dbm cfg using federated yes immediate
DB20000I The UPDATE DATABASE MANAGER CONFIGURATION command
completed
successfully.
/WORK # db2 get dbm cfg |grep "Federated Database"
Federated Database System Support (FEDERATED) = YES
```

Once the `FEDERATED` value in DBM CFG is changed, it is applied after restarting the database server, as shown in Example 2-104

Example 2-104 Restart the server

```
/WORK # db2stop force
02/14/2007 10:09:04 0 0 SQL1064N DB2STOP processing was
successful.
SQL1064N DB2STOP processing was successful.
/WORK # db2start
02/14/2007 10:09:08 0 0 SQL1063N DB2START processing was
successful.
244 Oracle to DB2 Conversion Guide for Linux, UNIX, and Windows
SQL1063N DB2START processing was successful.
```

2. Configure the component that is required to federate the table in another database.

- **WRAPPER:** A mechanism by which a federated server can interact with certain types of data sources. In our example, we create a wrapper for the `SAMPLE` database from `DB2_EMP`. You can check the data source from the catalog view `SYSCAT.SERVERS`.
- **SERVER:** A data source to a federated database. In our example, the data source is the `SAMPLE` database.
- **USER MAPPING:** Definition of mapping between an authorization ID that uses a federated database and the authorization ID and password to use at a specified data source.
- **NICKNAME:** Alias for a data source object.

In our example, we create a nickname, `REMOTE_EMPLOYEE`, for the `EMPLOYEE` table of the `SAMPLE` database. From `DB2_EMP`, we then use

this nickname in the query to join data. Example 2-105 shows the script `fed_config.db2`, which we used to set up the federated system in our lab.

Example 2-105 fed_config.db2 script

```
-----
-- create wrapper,user mapping, and nickname
-----
--Create wrapper;
CREATE WRAPPER drda;
SELECT * FROM syscat.wrappers;
CREATE SERVER fedserver type db2/udb version '9.1'
WRAPPER "DRDA"
AUTHID "db2inst1"
PASSWORD "db2inst1"
OPTIONS ( dbname 'SAMPLE' );
--
SELECT * FROM syscat.servers
-- Map user
CREATE USER MAPPING FOR USER
SERVER fedserver
OPTIONS( REMOTE_AUTHID 'db2inst1', REMOTE_PASSWORD 'db2inst1');
--
SELECT * FROM syscat.usermappings;
CREATE NICKNAME remote_employee FOR fedserver.db2inst1.employee;
```

Use the following command to execute the script:

```
/WORK # db2 -tf fed_config.db2
```

You can see the registered federation values in the result of the selection from the system catalog tables.

After you configure a federation server and link a remote table with a nickname in the local database, you can access the remote table because it is in the local database.

Example 2-106 on page 127 shows that you can delete and insert records on table `EMPLOYEE` of the `SAMPLE` database from `DB2_EMP` because it is one of the tables in this database.

Example 2-106 Access to a remote table

```
CONNECT TO db2_emp;
-----
-- insert a row in remote table
-----
DELETE FROM remote_employee WHERE empno='999999';
INSERT INTO remote_employee(empno,firstnme,lastname,workdept,edlevel)
VALUES ('999999','CARLOS','EDUARDO','E11',20);
SELECT empno,firstnme,workdept
FROM remote_employee WHERE empno='999999';
-----
-- Result of the SELECT
-----
EMPNO  FIRSTNME      WORKDEPT
-----
999999 CARLOS        E11
```

You also can join the remote table with the local table, as shown in Example 2-107.

Example 2-107 Join data in two tables

```
connect to db2_emp;
-----
-- Create a local table and insert 3 rows
-----
CREATE TABLE local_department
(deptno CHAR(3), deptname CHAR(20));
INSERT INTO local_department VALUES
('E01','Operation'),
('E10','Sales'),
('E11','Global Services');
SELECT * FROM local_department;
-----
-- Join data in two tables
-----
SELECT empno, firstnme, deptname
FROM remote_employee r, local_department d
WHERE r.workdept=d.deptno and empno='999999';
```

The result of joining a remote table (REMOTE_EMPLOYEE) and a local table (LOCAL_DEPARTMENT) is shown in Example 2-108.

Example 2-108 The result of joining two tables

```
-- Result of the first SELECT (SELECT FROM local_department)
DEPTNO DEPTNAME
-----
E01    Operation
E10    Sales
E11    Global Services

-- Result of the second SELECT ( Join local_department and remote_employee)
EMPNO  FIRSTNME      DEPTNAME
-----
999999 CARLOS Global Services
```

2.2.12 Oracle Data Dictionary compatible views

It is common for database administrators to have administrative scripts to retrieve information about Data Dictionary objects. It is also common to have application logic incorporated in PL/SQL code that retrieves Data Dictionary information.

DB2 provides a set of view definitions that mimic the most commonly used views from the Oracle Data Dictionary. More than 100 mapped views of Oracle, covering USER_*, ALL_*, and DBA_* views, are available with identical or close definitions.

In DB2, the system view DICTIONARY contains the names, schema, and description of Data Dictionary. Querying the system view DICT_COLUMNS returns the column names of each of the new dictionary views.

Table 2-11 lists the DB2 supported Data Dictionary views. The * notation applies to prefix DBA, USER, and ALL in front of each name.

Table 2-11 Oracle dictionary view supported in DB2

Category	View
General	<ul style="list-style-type: none">▶ *_CATALOG▶ *_DEPENDENCIES▶ *_OBJECTS▶ *_SEQUENCES▶ DBA/USER_TABLESPACES▶ DICTIONARY▶ DICT_COLUMNS

Category	View
Tables/View	<ul style="list-style-type: none"> ▶ *_COL_COMMENTS ▶ *_CONSTRAINTS ▶ *_CONS_COLUMNS ▶ *_INDEXES ▶ *_IND_COLUMNS ▶ *_PART_TABLES ▶ *_PART_KEY_COLUMNS ▶ *_SYNONYMS ▶ *_TABLES ▶ *_TAB_COL_STATISTICS ▶ *_TAB_COLUMNS ▶ *_TAB_COMMENTS ▶ *_TAB_PARTITIONS ▶ *_VIEWS ▶ *_VIEW_COLUMNS
Programming objects	<ul style="list-style-type: none"> ▶ *_PROCEDURES ▶ *_SOURCE ▶ *_TRIGGERS ▶ *_ERRORS
Security	<ul style="list-style-type: none"> ▶ DBA/USER_ROLE_PRIVS, ROLE_ROLE_PRIVS, SESSION_ROLES ▶ DBA/USER_SYS_PRIVS, ROLE_SYS_PRIVS, SESSION_PRIVS ▶ *_TAB_PRIVS, ROLE_TAB_PRIVS ▶ ALL/USER_TAB_PRIVS_MADE ▶ ALL/USER_TAB_PRIVS_RECD ▶ DBA_USERS ▶ DBA_ROLES

From the application's stand point, all these views can be referenced in the SQL and PL/SQL code, as shown in Example 2-109. The code snippet shows a part of a procedure that gathers information, such as owner, object name, and object type, about all invalid objects in a given schema. The string can be used later for recompiling, sending an e-mail notification, and other maintenance operations.

Example 2-109 Using Data Dictionary views in a PL/SQL procedure

```
CREATE OR REPLACE PROCEDURE Recompile_invalid_objects
  (existing_invalid_objects OUT VARCHAR2,
   in_owner                 IN VARCHAR2 DEFAULT NULL)
IS
  v_owner VARCHAR2(20) := NVL (UPPER (in_owner), 'APP');
BEGIN
  FOR invalid_objects_list IN (
    SELECT owner,
           object_name,
           object_type
    FROM all_objects
    WHERE owner = DECODE (v_owner, 'ALL', owner, v_owner)
          AND owner NOT IN ('SYSTEM', 'SYS')
          AND status = 'INVALID'
    ORDER BY owner, object_name, object_type)
  LOOP
    existing_invalid_objects := existing_invalid_objects
      || CHR(10)
      || RPAD (TRIM (invalid_objects_list.owner), 13)
      || RPAD (TRIM (invalid_objects_list.object_name), 31)
      || RPAD (TRIM (invalid_objects_list.object_type), 18);
  END LOOP;
END;
/
```

Example 2-110 loops through source code for database objects stored in the ALL_SOURCE Data Dictionary view. The application code prepares an SQL statement later for a dynamic execution based on object name and owner.

Example 2-110 Using ALL_SOURCE view

```
FOR get_source_info IN
  (SELECT owner, name, type, text
   FROM all_source
   WHERE owner = DECODE (v_owner, NULL, owner, v_owner)
        AND name = DECODE (v_name , NULL, name, v_name )
   ORDER BY owner, name, type
  )
LOOP
  ...
```

```
END LOOP;
```

Oracle also provides dynamic performance views that are updated dynamically by the Oracle instance with performance data. Dynamic performance views are prefixed with V_\$ and have public synonyms created with the V\$ prefix. These views are used by database administrators to monitor the database, and track cumulative information since startup. They are also sometime used in the application code to provide information about the database instance and contribute to the programming logic. Examples of such views are V\$INSTANCE, V\$DATABASE, V\$TABLESPACE, V\$DATAFILE, and V\$LOCK.

The Oracle dynamic performance views are part of the dictionary. It is a concept similar to the DB2 catalog views that are based on SYSIBM tables.

The DB2 SQL administrative views and routines provide an easy-to-use programmatic interface to the DB2 admin functionality through a construct that could be used in SQL PL. They encompass a collection of built-in views, table functions, procedures, and scalar functions for performing a variety of administrative tasks, such as reorganizing a table, capturing and retrieving monitor data, and retrieving the application ID of the current connection.

These routines and views can be invoked from an SQL-based application, a command line, or a command script.

The DB2 administrative views can be considered equivalent to the V\$ views in Oracle. Although the information provided by the V\$ views and the administrative views cannot be exactly the same, the information is common, and both return dynamic data.

For example, to obtain information about applications connected to the database from Oracle V\$ views, use the following query:

```
SELECT * FROM V$SESSION
```

An equivalent query to query data from DB2 administrative views is:

```
SELECT * FROM TABLE (SNAP_GET_APPL(CAST(NULL AS VARCHAR(128)),-1)) AS T
```

There are certain views that are particularly useful. One useful view is SYSENV_SYS_RESOURCES, which provides system information, such as memory, CPU, operating system, and host information. ENV_INST_INFO returns information about the current instance. The APPLICATIONS view returns information about connected applications. The DBCFG and DBMCFG views return information about database configuration and database manager configurations.

Table 2-12 shows some of the V\$ views and the equivalent administrative views or table functions.

Table 2-12 Oracle V\$ views and DB2 administrative views and table functions

Oracle	DB2
V\$INSTANCE	SNAPDBM administrative view or SNAP_GET_DBM table function
V\$DATABASE	SNAPDB administrative view or SNAP_GET_DB_V91 table function
V\$TABLESPACE	SNAPTBSP administrative view or SNAP_GET_TBSP_V91 table function
V\$DATAFILE	SNAPCONTAINER administrative view or SNAP_GET_CONTAINER_V91 table function
V\$SESSION	SNAPAPPL administrative view or SNAP_GET_APPL table function
V\$SQLTEXT	SNAPSTMT administrative view or SNAP_GET_STMT table function
V\$LOCK	SNAPLOCK administrative view or SNAP_GET_LOCK table function
V\$SYSSTAT (information for data buffer)	SNAPBP administrative view or SNAP_GET_BP table function
V\$SESSION_LONGOPS	LONG_RUNNING_SQL administrative view

Note: The administrative views are catalogued in the SYSIBMADM schema, and the table functions are catalogued in the SYSPROC schema. The SELECT privilege is required to access these objects.

The Snapshot administrative views or equivalent table functions have made monitoring simpler by providing access to monitoring information using SQL. In particular, from a database administrator's point of view, a common task is to gather continuous information about the system so that the overall state of the system is better known. In this regard, SYSIBMADM.SNAPDB,

SYSIBMADM.SNAPAPPL, and SYSIBMADM.SNAPSTMT are useful. A database administrator who previously administers the Oracle database on a continuous basis can do the same in DB2 by building a script that periodically refreshes a set of user tables from the three administrative views, and later query against these to generate trend usage, and use it to maintain complete control over the system.

In DB2 9.7, a new table function, MON_GET_PKG_CACHE_STMT, is available that can be used in a script to assist in monitoring.

For more information about DB2 administrative views and table functions, refer to *Administrative SQL Routines and Views*, SC10-4293.

2.3 DB2 command-line utilities

DB2 provides two flavors of command-line tools: the traditional Command Line Processor (CLP) and the complementary Command Line Processor Plus (CLPPlus). The DB2 CLP is the command-line interface that interacts with a DB2 server. You can use it to connect to databases, execute database utilities, issue SQL statements, execute scripts, or run the DB2 commands to manage your databases. Similar to CLP, CLPPlus offers support for many commands that are provided by the Oracle SQL*PLUS command-line utility.

2.3.1 Command Line Processor Plus

The Command Line Processor Plus (CLPPlus) is a command-line user interface that provides a complement to the functionality provided by the CLP. It offers advanced options, including developing and editing database objects using an SQL buffer, calling operating system and database management commands, compiling and running procedures, functions and packages, creating and formatting SQL type reports from the command line, and so on.

One of the important features of CLPPlus is its compatibility with Oracle's SQL*Plus utility. CLPPlus is designed as a quick, easy, and advanced command-line tool that provides compatibility for DBAs and application developers grown accustomed to Oracle's SQL*Plus interface. In this case, you may prefer using the DB2 CLPPlus interface when working with DB2 databases. You will find many familiar options, such as output formatting, developing in the SQL buffer, and so on. You can run a SQL*Plus script taken from Oracle in the DB2 CLPPlus with little or no modification.

To start the CLPPlus tool, just type **c1pp1us** from the Windows command prompt or UNIX shell prompt.

In a Windows environment, you can also start the CLPPLUS tool by selecting **Start → All Programs → IBM DB2 → your db2 copy → Command Line Tools → Command Line Processor Plus**.

Figure 2-8 shows DB2 CLPPlus in a Windows environment.

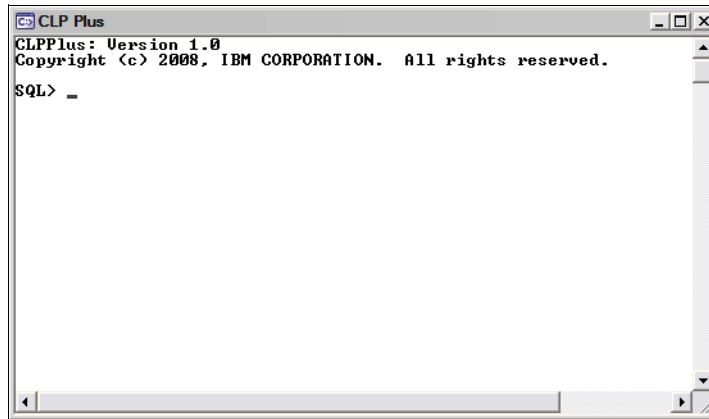


Figure 2-8 DB2 CLPPlus

Inside CLPPlus, you can run both operating system and database commands. To run the operating system commands, place the **HOST** operator in front of them. For example, if you were running on UNIX and you wanted to verify the existence of the `plsql.txt` file in your current directory, simply run:

```
SQL> HOST ls | grep plsql.txt
```

As an example on Windows, from within CLPPlus, we can run the **HOST** command followed by the **ipconfig** command to display the Windows IP configuration as follows:

```
SQL> HOST ipconfig
```

CLPPlus also gives you the ability to connect to any DB2 database without the need to catalog the database prior to connecting. Use the data server host name, port number, and database name along with your user credentials. In Figure 2-9 on page 135, user **DB2ADMIN** connects to a local database named **SAMPLE** that is listening on port 50000. To disconnect from the database, use the **disconnect** command.

```

CLP Plus
C:\Test>clpplus
CLPPlus: Version 1.0
Copyright (c) 2009. IBM CORPORATION. All rights reserved.

SQL> connect db2admin@localhost:50000/sample
Enter password:
Database Connection Information

Hostname = localhost
Database server = DB2/NT SQL09070
SQL authorization ID = db2admin
Local database alias = SAMPLE
Port = 50000

SQL> disconnect
SQL> _

```

Figure 2-9 Connecting to and disconnecting from a database in CLPPlus

Working with the SQL buffer is an essential part of the CLPPlus functionality. The SQL buffer is an “in memory” working area where CLPPlus keeps a copy of the most recently entered SQL statement or PL/SQL block. CLPPlus provides many commands to help manage the SQL buffer.

In Figure 2-10, we first load some PL/SQL code that is stored in a file using the GET command. Then we run this code straight from the buffer with the RUN command. For testing purposes, we have chosen a code sample that has a syntax error in it to show how CLPPlus can help manage debugging and execution of database code. In our case, the RUN command fails with a syntax error that is properly displayed in the CLPPlus editor (note that there is no space between the FROM clause and the table name).

```

CLP Plus
SQL>
SQL> get example.sql
1 SELECT  <INITCAP<last_name> || ', ' ||
2         INITCAP<first_name>> as employee,
3         CONNECT_BY_ROOT <INITCAP< first_name> || ', ' ||
4         INITCAP<last_name>> as top_manager,
5         SYS_CONNECT_BY_PATH (<INITCAP<last_name> || ', ' ||
6         INITCAP<first_name>>,'>') as report_chain
7 FROMemployee -- This is intentional error
8 START WITH band = '5'
9 CONNECT BY PRIOR emp_id = emp_mgr_id
10* ORDER BY employee;
SQL>
SQL>
SQL> run
1 SELECT  <INITCAP<last_name> || ', ' ||
2         INITCAP<first_name>> as employee,
3         CONNECT_BY_ROOT <INITCAP< first_name> || ', ' ||
4         INITCAP<last_name>> as top_manager,
5         SYS_CONNECT_BY_PATH (<INITCAP<last_name> || ', ' ||
6         INITCAP<first_name>>,'>') as report_chain
7 FROMemployee -- This is intentional error
8 START WITH band = '5'
9 CONNECT BY PRIOR emp_id = emp_mgr_id
10* ORDER BY employee;
ERROR near line 1:
DB2 SQL Error: SQLCODE=-104, SQLSTATE=42601, SQLERRMC=FROMemployee;'> as report_
chain
;FROM, DRIVER=3.57.78
SQL>
SQL> edit
DB250000I: The command completed successfully.

```

Figure 2-10 Working with the SQL buffer: GET, RUN, and EDIT commands

To fix the intentional error, we edit the code using the EDIT command. With the EDIT command, the buffer content is displayed in our preferred text editor (in this case, Notepad) and we could correct the error and save the changes, as shown in Figure 2-11.

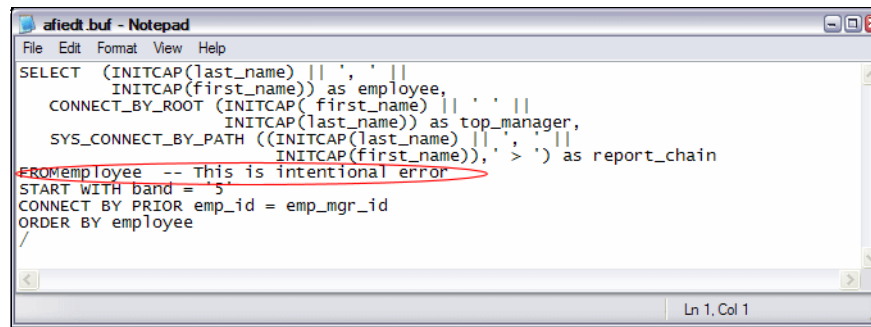


Figure 2-11 Editing the code

Once the error is corrected (we added space between the FROM clause and the table name), the SQL buffer gets updated with the new version of the SQL script and is ready to be used as soon as we close the text editor. If we run it again, the query will succeed and display the result set in the CLPPlus window.

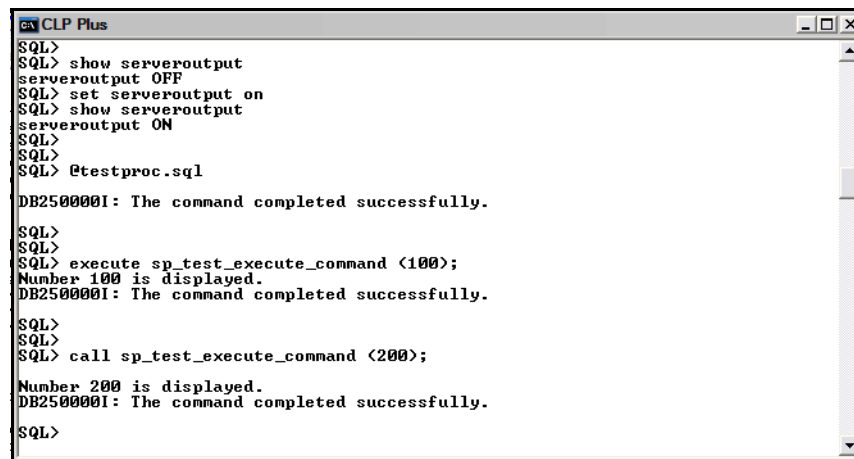
You can execute the PL/SQL procedures in CLPPlus by using the EXECUTE command. Example 2-111 shows a simple PL/SQL procedure saved in the example.sql file.

Example 2-111 Sample procedure

```
CREATE OR REPLACE PROCEDURE sp_test_execute_command
(p_first_id IN NUMBER)
IS
BEGIN
    IF p_first_id IS NOT NULL THEN
        DBMS_OUTPUT.PUT_LINE('Number ' || p_first_id || ' is displayed.');
```

Figure 2-12 on page 137 demonstrates two ways of running a PL/SQL procedure in CLPPLUS. Because the procedure uses the DBMS_OUTPUT built-in package to display messages to the user, before we run it, we have to set SERVEROUTPUT ON in order to see these messages. We first check for the current setting of the SERVEROUTPUT parameter using the SHOW command and then we set it to ON. To execute the procedure, we run the EXECUTE command followed by the procedure name and enter a parameter of 100.

Because CLPPlus provides high level compatibility, using the DB2 CALL statement to run the PL/SQL procedure produces the same result, as demonstrated in the second call to this procedure using a parameter of 200.



```

SQL>
SQL> show serveroutput
serveroutput OFF
SQL> set serveroutput on
SQL> show serveroutput
serveroutput ON
SQL>
SQL>
SQL> @testproc.sql
DB250000I: The command completed successfully.

SQL>
SQL>
SQL> execute sp_test_execute_command <100>;
Number 100 is displayed.
DB250000I: The command completed successfully.

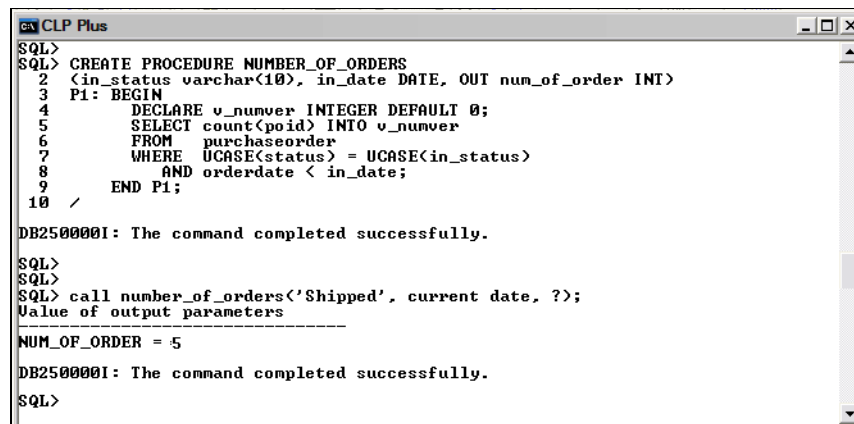
SQL>
SQL>
SQL> call sp_test_execute_command <200>;
Number 200 is displayed.
DB250000I: The command completed successfully.

SQL>

```

Figure 2-12 Executing PL/SQL procedure in CLPPlus

As shown in Figure 2-12, we can use the DB2 syntax to call PL/SQL procedures in CLPPlus, and we also can create and run SQL PL procedures in CLPPlus. Support for SQL PL is another big advantage of CLPPlus. In Figure 2-13, we create a SQL PL procedure using the OUT parameter in the SAMPLE database, call it, and display the output parameter in DB2 style. The result is exactly the same as the CLP provides.



```

SQL>
SQL> CREATE PROCEDURE NUMBER_OF_ORDERS
2  <in_status varchar(10), in_date DATE, OUT num_of_order INT>
3  Pi: BEGIN
4      DECLARE v_numver INTEGER DEFAULT 0;
5      SELECT count(poid) INTO v_numver
6      FROM   purchaseorder
7      WHERE  UCASE(status) = UCASE(in_status)
8      AND   orderdate < in_date;
9      END Pi;
10 /
DB250000I: The command completed successfully.

SQL>
SQL>
SQL> call number_of_orders('Shipped', current date, ?);
Value of output parameters
-----
NUM_OF_ORDER = 5
DB250000I: The command completed successfully.

SQL>

```

Figure 2-13 Executing the SQL PL procedure with OUT parameters in CLPPlus

CLPPlus also provides numerous options for producing well formatted reports on the fly. The query shown in Figure 2-14 displays information about five employees with selected employee numbers. The output of this query, also shown in Figure 2-14, is wrapped, not properly formatted (for example, the money fields do not have a currency sign) and, in general, is hard to read.

```

SQL>
SQL> SELECT firstnme, lastname, salary, bonus, empno emp_number,
2      DECODE(sex, 'F', 'Female', 'M', 'Male', 'Not Provided') sex
3 FROM employee
4 WHERE empno in ('000010', '000120', '200340', '000200', '200010')
5 ORDER BY empno;

```

FIRSTNME	LASTNAME	SALARY	BONUS
CHRISTINE	HAAS	152750.00	1000.00
000010	Female		
SEAN	O'CONNELL	49250.00	600.00
000120	Male		
DAVID	BROWN	57740.00	600.00
000200	Male		
DIAN	HEMMINGER	46500.00	1000.00
200010	Female		
ROY	ALONZO	31840.00	500.00
200340	Male		

```

SQL>

```

Figure 2-14 Formatting in CLPPlus: part 1

Using the rich pallet of formatting options offered by CLPPlus, we can greatly improve the appearance of this result set by changing the output settings and using the column formatting options. In Figure 2-15 on page 139, we first check for the current settings by running the SHOW command and then provide new values for the output related parameters with the SET command. We also show how long it took to execute this query by running the SET TIMING ON command. We increase the width of the output line and apply special formatting rules (dollar notation) to the salary and bonus columns by using the COLUMN FORMAT command. Additionally, using the HEADING command, we select new titles for the columns, such as lastname or firstname, to make them more meaningful to the user, and we add an alternative name to the salary column. The query now produces a well-formatted, more meaningful output and displays the execution time.

```

SQL>
SQL> show linesize
linesize 80
SQL> set linesize 100
SQL> show linesize
linesize 100
SQL>
SQL> show timing
timing OFF
SQL> set timing on
SQL> show timing
timing ON
SQL>
SQL> COLUMN lastname HEADING 'LAST NAME'
SQL> COLUMN firstnme HEADING 'FIRST NAME'
SQL> COLUMN salary HEADING 'COMPENSATION'
SQL>
SQL> COLUMN salary FORMAT $999,999.99
SQL> COLUMN bonus FORMAT $999,999.99
SQL>
SQL>
SQL> SELECT firstnme, lastname, salary, bonus, empno emp_number,
2 DECODE(sex, 'F', 'Female', 'M', 'Male', 'Not Provided') sex
3 FROM employee
4 WHERE empno in ('000010', '000120', '200340', '000200', '200010')
5 ORDER BY empno;

```

FIRST NAME	LAST NAME	COMPENSATION	BONUS	EMP_NUMBER	SEX
CHRISTINE	HAAS	\$152,750.00	\$1,000.00	000010	Female
SEAN	O'CONNELL	\$49,250.00	\$600.00	000120	Male
DAVID	BROWN	\$57,740.00	\$600.00	000200	Male
DIAN	HEMMINGER	\$46,500.00	\$1,000.00	200010	Female
ROY	ALONZO	\$31,840.00	\$500.00	200340	Male

```

Elapsed time: 17 millisecond(s)
SQL>

```

Figure 2-15 Formatting in CLPPlus: part 2

CLPPlus is a powerful and rich functionally command-line interface that provides a robust and simple environment for DBAs and application developers for *ad hoc* SQL and PL/SQL prototyping, development, scripting, and reporting. CLPPlus is also a first-class tool to assist in migration and ongoing maintenance that requires only a minimal investment in skill transfer. To discover more about the CLPPlus command utility, visit the DB2 Information Center, found at:

<http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/index.jsp?topic=/com.ibm.swg.im.dbclient.clpplus.doc/doc/c0056269.html>

2.3.2 Command Line Processor

CLP is a basic DB2 command tool with which most DB2 database administrators and developers are familiar. The CLP is used to execute database utilities, SQL statements, and online help. It offers a variety of command options and can be started in interactive input mode, command mode, and batch mode. From the enablement standpoint, you can utilize the CLP to run your conversion scripts. The scripts can contain the definition of database objects, such as DDL, DML, PLS/SQL, and so on. You can also include the data movement commands, such as IMPORT or LOAD.

To start the CLP in a Windows environment, select **Start → All Programs → IBM DB2 → your db2 copy → Command Line Tools → Command Line Processor**. Another way is to select **Start → Run** and enter `db2cmd`.

Figure 2-16 shows the DB2 CLP in a Windows environment.

```

DB2 CLP - DB2COPY1 - E:\SQLLIB\BIN\db2setcp.bat DB2SETCP.BAT DB2.EXE
(c) Copyright IBM Corporation 1993,2007
Command Line Processor for DB2 Client 9.7.0

You can issue database manager commands and SQL statements from the command
prompt. For example:
    db2 => connect to sample
    db2 => bind sample.bnd

For general help, type: ?.
For command help, type: ? command, where command can be
the first few keywords of a database manager command. For example:
    ? CATALOG DATABASE for help on the CATALOG DATABASE command
    ? CATALOG           for help on all of the CATALOG commands.

To exit db2 interactive mode, type QUIT at the command prompt. Outside
interactive mode, all commands must be prefixed with 'db2'.
To list the current command option settings, type LIST COMMAND OPTIONS.

For more detailed help, refer to the Online Reference Manual.

db2 =>

```

Figure 2-16 DB2 CLP

In a Linux or UNIX environment, you can start the CLP by running the **db2** command. You also can run a statement directly on the operating system prompt by adding the prefix `db2` to the statement.

The DB2 SQL scripts can be executed using the flags shown in this example:

```
db2 -tvf <scriptName>
```

In this example, the purpose of the `-t` flag is to tell the Command Line Processor to use a semicolon (;) as the statement termination character. The `-v` flag stands for verbose, so that the statements are display before being executed. The `-f` flag tells DB2 that the next parameter is a script file name to execute.

Example 2-112 shows how to run a script to create a table using CLP.

Example 2-112 Running a procedure with CLP

```
db2inst1> cat sample_script1.sql
```

```

CREATE TABLE sample1
( id    NUMBER(8),
  name  VARCHAR2(40)
)
;

```

```
db2inst1> db2 -tvf sample_script1.sql
CREATE TABLE simple1 ( id NUMBER(8) , name VARCHAR2(40) )
DB20000I The SQL command completed successfully.
```

SQLCOMPAT mode

When executing scripts using the **db2** command with the **-t** flag, the default statement terminator is a semi-colon (alternate terminators can be specified). Oracle uses the forward slash ("/) as the default statement terminator. In order to enhance compatibility and allow scripts that were written for Oracle and containing forward slashes to run seamlessly in DB2, you can use the **-td** option to tell the Command Line Processor to use a different statement terminator. For example, to use the forward-slash (/) as the statement terminator, run:

```
db2 -td/ -vf <scriptName>
```

Example 2-113 shows how to run a script with a forward slash using the **-td** command-line option.

Example 2-113 Script using forward-slash as the statement terminator

```
db2inst1> cat sample_create_proc.sql
CREATE TABLE "ACCOUNTS" (
    "ACCT_ID" NUMBER(31) NOT NULL,
    "DEPT_CODE" CHAR(3) NOT NULL,
    "ACCT_DESC" VARCHAR2(2000),
    "MAX_EMPLOYEES" NUMBER(3),
    "CURRENT_EMPLOYEES" NUMBER(3),
    "NUM_PROJECTS" NUMBER(1),
    "CREATE_DATE" DATE DEFAULT SYSDATE,
    "CLOSED_DATE" DATE DEFAULT SYSDATE+1 year)
/
CREATE OR REPLACE PACKAGE Account_Package AS
    TYPE customer_name_cache IS TABLE OF Employees%ROWTYPE INDEX BY PLS_INTEGER;
    PROCEDURE Account_List(p_dept_code IN accounts.dept_code%TYPE,
        p_acct_id IN accounts.acct_id%TYPE,
        p_Employees_Name_Cache OUT Customer_Name_Cache);
END Account_Package;
/
```

```
db2inst1> db2 -td/ -vf sample_create_proc.sql
CREATE TABLE "ACCOUNTS" ( "ACCT_ID" NUMBER(31) NOT NULL, "DEPT_CODE" CHAR(3) NOT
NULL, "ACCT_DESC" VARCHAR2(2000), "MAX_EMPLOYEES" NUMBER(3), "CURRENT_EMPLOYEES
" NUMBER(3), "NUM_PROJECTS" NUMBER(1), "CREATE_DATE" DATE DEFAULT SYSDATE, "CLOS
ED_DATE" DATE DEFAULT SYSDATE)
DB20000I The SQL command completed successfully.
```

```
CREATE OR REPLACE PACKAGE Account_Package AS
    TYPE customer_name_cache IS TABLE OF Employees%ROWTYPE INDEX BY PLS_INTEGER;
    PROCEDURE Account_List(p_dept_code IN accounts.dept_code%TYPE,
        p_acct_id IN accounts.acct_id%TYPE);
END Account_Package;
```

DB20000I The SQL command completed successfully.

You can achieve the same results by changing the SQLCOMPAT mode at the beginning of a script, or set it on the command line prior to running the script (allowing you to leave the script unchanged from its Oracle origins). This is shown in Example 2-114, which executes the script in Example 2-113 on page 141.

Example 2-114 Executing a script in SQLCOMPAT PLSQL mode

```
-- To execute this script, run: db2 -tvf <scriptname>
-- Uncomment the next line or make sure to set beforehand
-- SET SQLCOMPAT PLSQL;
db2inst1> db2 SET SQLCOMPAT PLSQL
DB20000I The SET SQLCOMPAT command completed successfully.

db2inst1> db2 -tvf sample_create_proc.sql
CREATE TABLE "ACCOUNTS" ( "ACCT_ID" NUMBER(31) NOT NULL, "DEPT_CODE" CHAR(3)
NOT
NULL, "ACCT_DESC" VARCHAR2(2000), "MAX_EMPLOYEES" NUMBER(3),
"CURRENT_EMPLOYEES
" NUMBER(3), "NUM_PROJECTS" NUMBER(1), "CREATE_DATE" DATE DEFAULT SYSDATE,
"CLOS
ED_DATE" DATE DEFAULT SYSDATE)
DB20000I The SQL command completed successfully.

CREATE OR REPLACE PACKAGE Account_Package AS
TYPE customer_name_cache IS TABLE OF Employees%ROWTYPE INDEX BY PLS_INTEGER;
PROCEDURE Account_List(p_dept_code IN accounts.dept_code%TYPE,
p_acct_id IN accounts.acct_id%TYPE);
END Account_Package;
DB20000I The SQL command completed successfully.
```



Enablement tools

Enabling your Oracle database and application to run with DB2 9.7 begins with the deployment of the table schema, followed by the movement of the table data and the deployment of procedural objects. Table schema objects include table, index, and constraint definitions, and procedural objects include PL/SQL packages, procedures, functions, and triggers.

IBM provides a set of tools that simplify and semi-automate the process. These tools make both the deployment of schema definitions and data movement manageable even in cases where tens of thousands of DDL and procedural objects exist.

In this chapter, we discuss three tools:

- ▶ The IBM Data Movement Tool and IBM Optim Development Studio can assist you with moving DDLs, table data, and PL/SQL objects from an Oracle database to a DB2 database.
- ▶ DB2 MEET helps you evaluate the PL/SQL compatibility ratio.

3.1 IBM Data Movement Tool

The IBM Data Movement Tool (DMT) is the tool of choice for moving schema and data efficiently between Oracle (Version, 8, 9, 10, and 11) as a source and DB2 for Linux, UNIX, and Windows (LUW) (Version 9.1, 9.5, and 9.7) as a target. This tool also supports many other databases as a source, including Sybase, Microsoft SQL Server, MySQL, PostgreSQL, and MS SQL Access. DMT also supports DB2 for z/OS and DB2 for iSeries® as the target RDBMS.

DMT has two modes of operation:

- ▶ An interactive mode, using a GUI-based interface that is easy to use.
- ▶ A command-line mode, which can provide prompts when necessary, but also can be used for scripting and launching the data movement process without user intervention.

The methodology used by the IBM Data Management Tool is straight forward. It uses a series of Java programs and shell scripts (Windows or UNIX) to access the source Oracle database and generate database scripts necessary to recreate the objects in DB2. It also extracts the data from the source Oracle tables using a multi-threaded technique and generates scripts to populate DB2 tables with the extracted data using the very fast DB2 LOAD utility.

An advantage of this method, where all of the necessary scripts are generated by the tool, is that they can be reviewed and modified before deploying them to the target DB2 database. In addition, much of the Oracle syntax that is extraneous or unsupported is automatically translated to the DB2 equivalent. As an example, the Oracle storage details of the CREATE TABLE command are removed when preparing the scripts for DB2.

Installation and configuration of the IBM Data Movement Tool

The IBM Data Movement Tool is a free of charge utility that can be downloaded from developerWorks at the following address:

<http://www.ibm.com/developerworks/data/library/techarticle/dm-0906datamovement/>

The forum for the tool can be found at:

<http://www.ibm.com/developerworks/forumsforum.jspa?forumID=1803>

Table 3-1 on page 145 shows that the prerequisites for the tool include Java Version 1.5 or higher and the JAR files to allow JDBC connections to the RDBMSs.

Table 3-1 DMT prerequisites

RDBMS	JDBC JAR files
DB2 for Linux, UNIX, and Windows	db2jcc.jar and db2jcc_license_cu.jar
Oracle	ojdbc14.jar, xdb.jar, xmlparserv2.jar, or classes12.jar or classes111.jar for Oracle 7 or 8i.
SQL Server	sqljdbc.jar
Sybase	jconn3.jar
MySQL	mysql-connector-java-5.0.8-bin.jar
PostgreSQL	postgresql-8.1-405.jdbc3.jar
Ingres	ijjdbc.jar
DB2 for z/Os	db2jcc.jar, db2jcc_license_cisuz.jar
DB2 for iSeries	jt400.jar
Microsoft Access	Optional Access_JDBC30.jar

The DMT can be hosted on the source Oracle Server, target DB2 server, or a machine that has connectivity to both servers. We recommend installing DMT on the target DB2 server.

To install the tool, simply unzip the product and you should have three files in your directory:

- ▶ `IBMDDataMovementTool.jar`: This is the Java JAR file containing the IBM DMT application.
- ▶ `IBMDDataMovementTool.cmd`: This is the Windows command script to run the tool.
- ▶ `IBMDDataMovementTool.sh`: This is the Linux/UNIX shell script to run the tool.

When you execute the `IBMDDataMovementTool.cmd` or `IBMDDataMovmenentTool.sh` file, the tool begins in GUI mode. If GUI support is not available, for example, if you are using a simple UNIX terminal emulator, you can start the command-line interface by using the `-console` option. We also recommend using a command-line interface for moving large amounts of data.

3.1.1 Moving DDLs and data with the IBM Data Movement Tool

In order to move the data from an Oracle source database to a DB2 target database, the DMT requires connection information for both Oracle and DB2. DMT also requires that you specify the path to JDBC drivers. These parameters can be directly entered into the GUI interface or through a command-line interface.

Figure 3-1 shows the DMT with the required fields by way of a GUI.

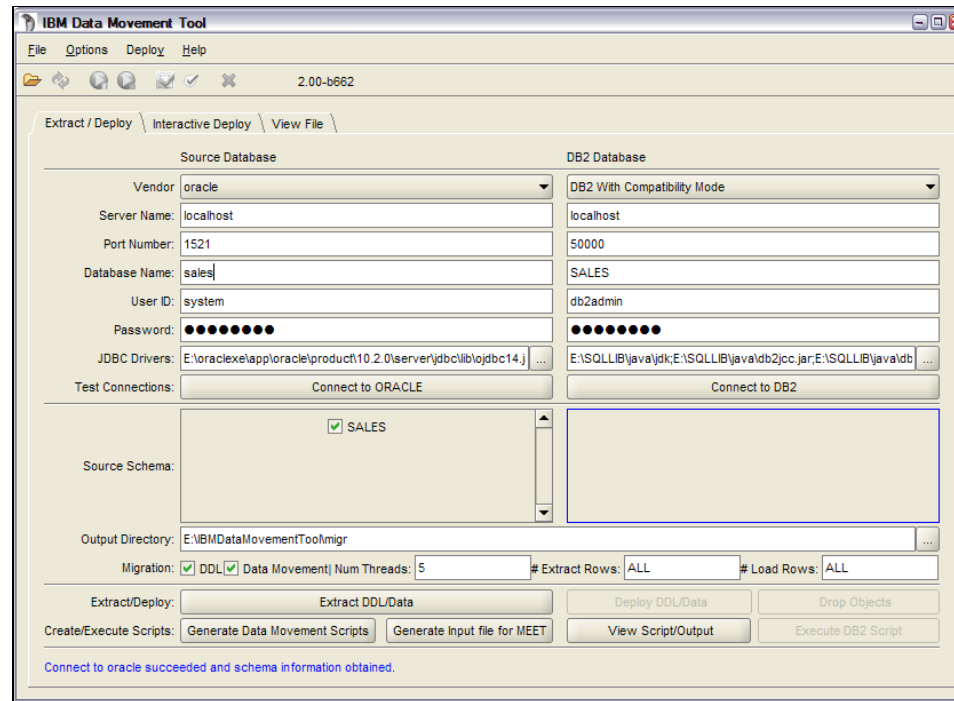


Figure 3-1 IBM DMT GUI

The extraction process retrieves the original Oracle data definitions (DDL), and can also automatically modify the code to make it more compatible with DB2. The options that affect the DB2 definitions can be selected in the Options menu bar, as shown in Figure 3-2 on page 147.

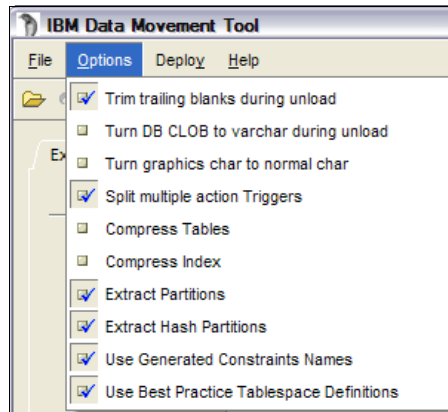


Figure 3-2 IBM DMT extraction options

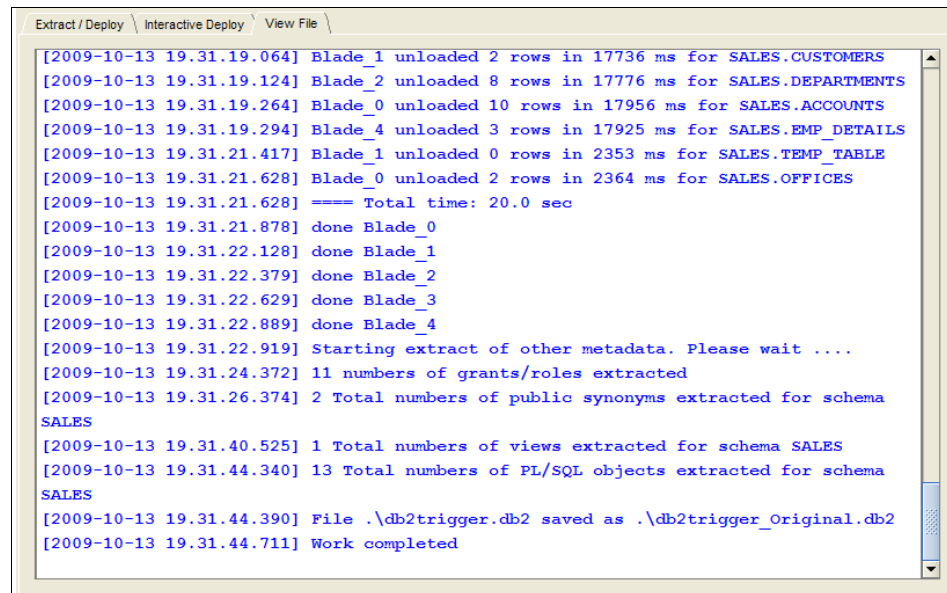
We recommend selecting the following options at a minimum:

- ▶ Trim trailing blanks during unload
- ▶ Split multiple action triggers
- ▶ Extract partitions
- ▶ Use generated constraints names
- ▶ Use best practice table space definitions

After the connection both to Oracle and to DB2 has been established, you can start the extraction of the DDL and data from Oracle by pressing the **Extract DDL/Data** button. You have the flexibility to choose particular schemas on the source Oracle database that you want to move. Note that the table data is temporarily stored by DMT in flat files, so you will need to provision extra space on your system.

Once the Extract DDL/Data process is completed, you can complete the data movement operation just by pressing the **Deploy DDL/Data** button.

You can check the progress of data movement by way of the interactive View File window (Figure 3-3).

The screenshot shows a window titled 'Extract / Deploy' with a sub-tab 'View File'. The window contains a text area with the following log output:

```
[2009-10-13 19.31.19.064] Blade_1 unloaded 2 rows in 17736 ms for SALES.CUSTOMERS
[2009-10-13 19.31.19.124] Blade_2 unloaded 8 rows in 17776 ms for SALES.DEPARTMENTS
[2009-10-13 19.31.19.264] Blade_0 unloaded 10 rows in 17956 ms for SALES.ACCOUNTS
[2009-10-13 19.31.19.294] Blade_4 unloaded 3 rows in 17925 ms for SALES.EMP_DETAILS
[2009-10-13 19.31.21.417] Blade_1 unloaded 0 rows in 2353 ms for SALES.TEMP_TABLE
[2009-10-13 19.31.21.628] Blade_0 unloaded 2 rows in 2364 ms for SALES.OFFICES
[2009-10-13 19.31.21.628] ==== Total time: 20.0 sec
[2009-10-13 19.31.21.878] done Blade_0
[2009-10-13 19.31.22.128] done Blade_1
[2009-10-13 19.31.22.379] done Blade_2
[2009-10-13 19.31.22.629] done Blade_3
[2009-10-13 19.31.22.889] done Blade_4
[2009-10-13 19.31.22.919] Starting extract of other metadata. Please wait ....
[2009-10-13 19.31.24.372] 11 numbers of grants/roles extracted
[2009-10-13 19.31.26.374] 2 Total numbers of public synonyms extracted for schema
SALES
[2009-10-13 19.31.40.525] 1 Total numbers of views extracted for schema SALES
[2009-10-13 19.31.44.340] 13 Total numbers of PL/SQL objects extracted for schema
SALES
[2009-10-13 19.31.44.390] File .\db2trigger.db2 saved as .\db2trigger_Original.db2
[2009-10-13 19.31.44.711] Work completed
```

Figure 3-3 IBM DMT view results of Extract DDL/Data

3.1.2 Deploying PL/SQL objects with the IBM Data Movement Tool

The IBM Data Movement Tool can also be used for deployment of PL/SQL objects, such as triggers, functions, procedures and packages. The PL/SQL source code is extracted as part of the Extract DDL/Data operation when the DDL check box is checked (which is the default, as shown in Figure 3-1 on page 146).

Once the Extract DDL operation is completed, the extracted DDL and PL/SQL become visible in the Interactive Deploy window (Figure 3-4 on page 149) after pressing the **Refresh** button.

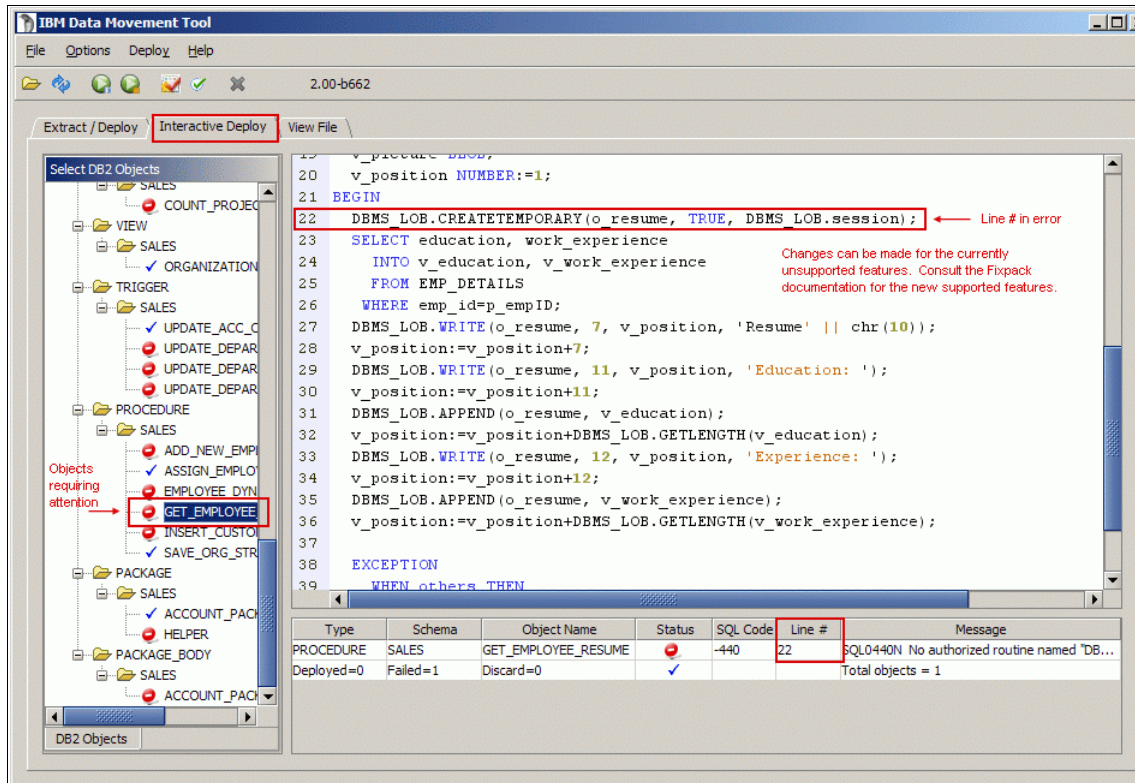


Figure 3-4 Interactive Deploy of the objects

The sequence of the deployment operation is as follows:

1. Ensure that you are connected to DB2 by using the Extract/Deploy tab.
2. Click the **Interactive Deploy** tab.
3. Click the **Open Directory** button to select the working directory containing the previously extracted objects, or just click the **Refresh** button. The objects are read and listed in a tree view.
4. You can deploy all objects by pressing the **Deploy All Objects** button on the toolbar. Most objects will deploy successfully; some may fail.
5. You can display the source of an object that failed to deploy (the object has a red icon next to it) either by selecting it from the tree view on the left, or by selecting it from the deployment log on the lower right. Right-click the error in the deployment log and select **See Detailed Error Message** to open the detailed message in the edit area in the upper right pane.

6. The Oracle compatibility mode generally allows deployment of objects as is. However, there may still be unsupported features that prevent successful deployment of some objects out of the box. Using the editor, you can adjust the source code of these objects to work around any issues. When you deploy the changed object, the new source is saved with a backup of the old source.
7. You can select one or more objects by pressing and holding the Ctrl key and clicking each object, and then click **Deploy Selected Objects** button on the toolbar to deploy the objects after they have been edited. Often, deployment failures occur in a cascade, which means that once one object is successfully deployed, others that depend on it will also deploy.
8. Repeat steps 5 through 7 until all objects have been successfully deployed.

3.1.3 Maintaining the current version of the tool

Because the tool is constantly being improved, we recommend that you check for updates periodically. You can see your current version by selecting **Help** → **About** or by using the -version option in command-line mode. The version is also displayed in the GUI, as shown in Figure 3-5.

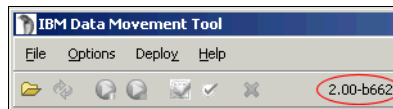


Figure 3-5 Current DMT version

DMT even provides an option to check if a newer version has been released by selecting **Help** → **Check Version** (Figure 3-6).

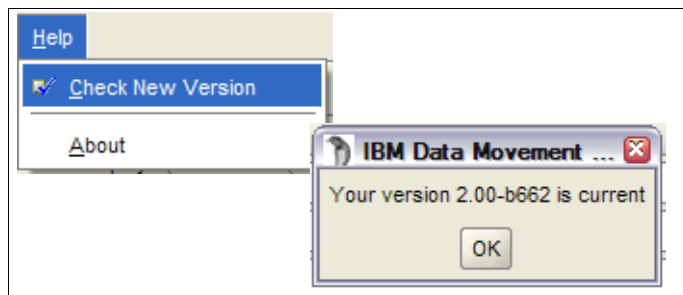


Figure 3-6 IBM DMT version information

3.2 IBM Optim Development Studio

IBM Optim Development Studio Version 2.2 (ODS) is an integrated database development environment that speeds application design, development, and deployment while increasing data access efficiency and performance. Using ODS, you can develop and test PL/SQL and SQL PL routines, generate and deploy data-centric Web services, create and run SQL and XQuery queries, and develop and optimize Java applications. ODS has been enhanced to provide native Oracle support and compatibility between DB2 and Oracle. You can utilize the functionality to move database objects from Oracle to DB2.

3.2.1 Moving DDLs and data with Optim Development Studio

Different from DMT, Optim Developer Studio is primarily a development environment and not a high speed data movement tool for a large amount of data. Optim Developer Studio focuses on portable ways of moving data (mainly SQL statements) between heterogeneous RDBMSs, and is useful during the development of a new database application or while extending an existing one. Application developers can work concurrently on the Oracle or DB2 database, developing for one and then subsequently move to the other for testing or production phases. The data movement feature utilizes logged INSERT-from-SELECT and thus limits the speed of the data transfer.

Optim Developer Studio is useful when the data to be transferred is small to medium size (a few GB). This is handy for test databases or when longer data transfer times are acceptable.

You can use the ODS Data Source Explorer window (Figure 3-7) to concurrently connect to both Oracle and DB2. After the connections are established, you can use standard *drag and drop* techniques to easily move database objects, such as tables (including data), procedures, and so on, from one database to the other. This technique does not allow you to make any changes to the definitions before deployment.

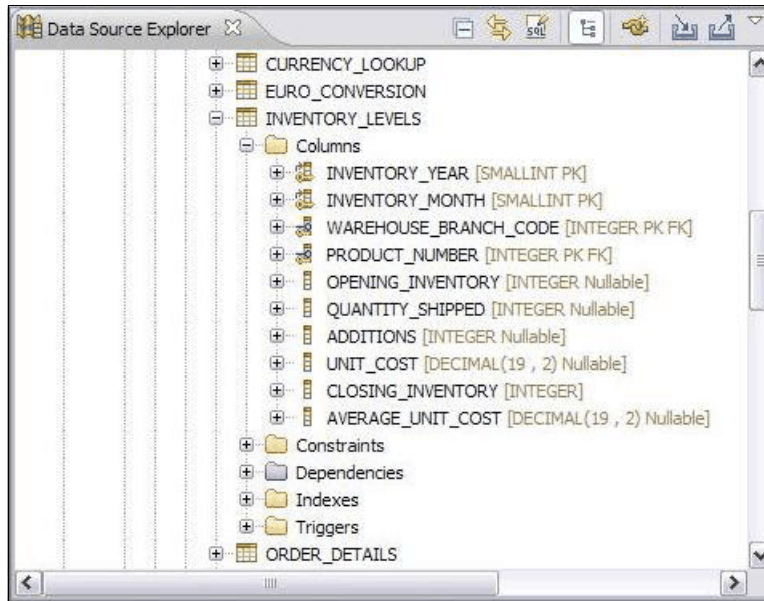


Figure 3-7 Data Source Explorer window

You can also use the *copy and paste* approach, which provides greater flexibility for moving tables and data from Oracle to DB2. Using this approach, you can change some data types in the table structure during the movement. This method allows you to move table definitions only (without data) or specify a particular error handling for the process.

In addition to the *drag and drop* and *copy and paste* approaches, the DDL script generation function of ODS is also very handy for object movement. You can save the generated DDL as a script in the SQL scripts folder of your data development project for immediate or future use.

3.2.2 Deploying PL/SQL objects with Optim Development Studio

Optim Development Studio also offers capabilities for deploying PL/SQL objects into DB2. The advantage of Optim is that you can deploy PL/SQL just by dragging and dropping the objects from Oracle to DB2 on the same window without any intermediate steps.

Follow these steps to drag a PL/SQL object from an Oracle to a DB2 project:

1. From the Data Source Explorer, expand your Oracle connection and find the PL/SQL objects you want to move.
2. Adjust your workbench so that the Data Source Explorer and the Data Project Explorer are both visible.
3. Drag the PL/SQL objects from the Data Source Explorer to the PL/SQL Packages folder of your DB2 project in the Data Project Explorer. If the Data Source Explorer and Data Project Explorer are not both visible, you can alternately copy from the source and paste to the target project.
4. When the drag and drop operation completes, you are notified that an incompatibility can exist, as shown in Figure 3-8. Click **OK**.

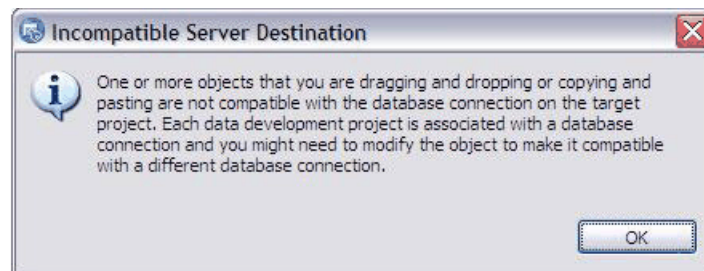


Figure 3-8 Accept that incompatibilities can exist between projects

5. Open the PL/SQL objects that you have moved to your DB2 project by double-clicking it or by right-clicking and selecting **Open** from the pop-up menu.
6. Make your changes to the source code in the editor.
7. Save your changes by pressing Ctrl+s or by selecting **File** → **Save**.

8. Right-click the PL/SQL object and select the **Deploy...** option (Figure 3-9).

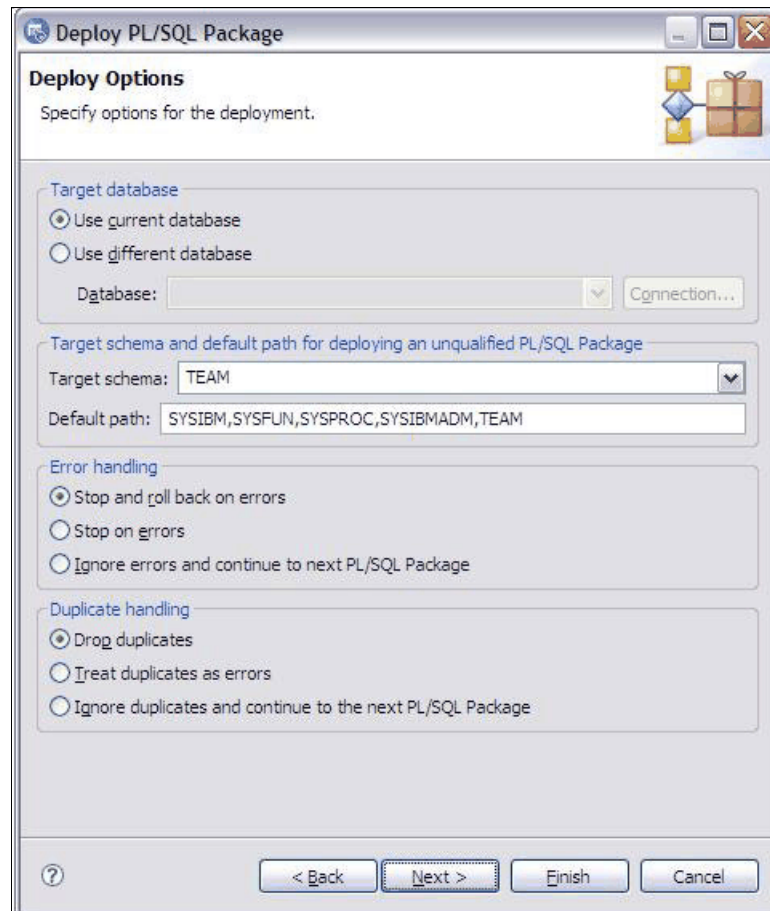


Figure 3-9 Specify deploy options

The third group (Error handling) appears in this window only when you have selected more than one PL/SQL package to deploy with this action.

9. When developing, it is a good practice to select the **Enable debugging** check box in the PL/SQL Package Options window of the Deploy wizard so that DB2 compiles the object with debugging information.
10. Click **Finish**. The selected objects are deployed sequentially in a background thread. The results are reported in the SQL Results view.

For the details about ODS, refer to:

<http://www-01.ibm.com/software/data/studio/>

3.3 MEET DB2 evaluation utility

IBM has created an internal utility named MEET DB2 that can analyze all the objects in your Oracle database and score them. It produces a report of what will work out of the box and where adjustments need to be made. Your IBM account representative or sales contact can run this utility to help quickly provide a compatibility assessment of your current Oracle database with DB2.

The utility is simple to run, accepting text as input, and rapidly produces an HTML report that summarizes the compatibility of the DDL and procedural objects in the input. The analysis provides detailed insight in moments about the level of compatibility that DB2 can provide for the database definitions and objects defined in the Oracle source database.

Figure 3-10 shows a sample MEET DB2 report.

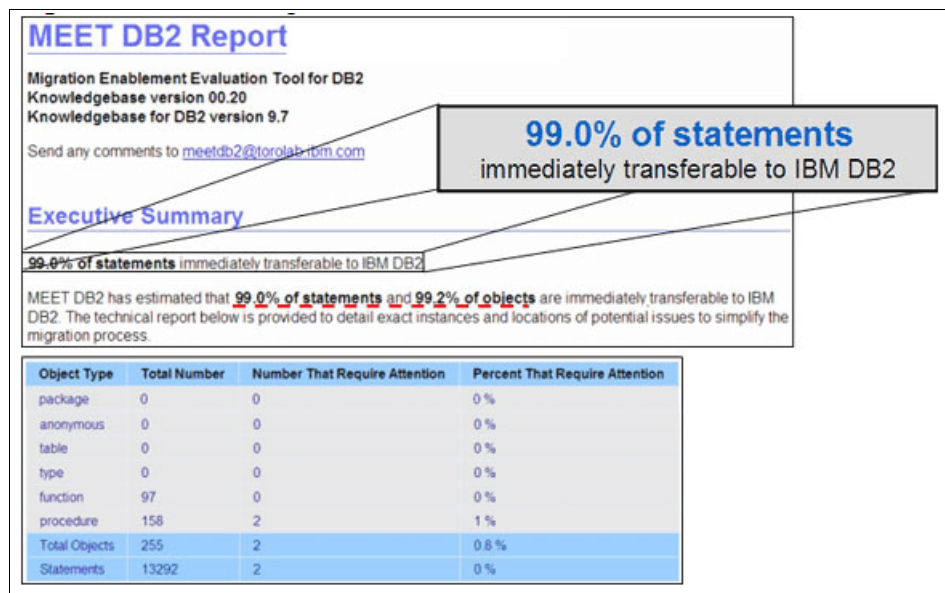


Figure 3-10 MEET DB2 report tool for assessment

3.4 Conclusion

IBM provides a rich set of tools to assist you with your enablement project. You can use IBM Data Movement Tool and Optim Development Studio to move the data from Oracle to DB2. You can also use IBM Data Movement Tool and Optim Development Studio to extract DDL and PL/SQL objects from Oracle and deploy them in DB2. Finally, you can contact your IBM Representative to run evaluation report on your Oracle source code to find how compatible it is with DB2.



Enablement scenario

Using the DB2 compatibility features of DB2 9.7, you can move your Oracle database application to DB2 with minimal or no changes. In the previous chapters, we present most of the compatibility features in DB2, including natively supported syntax and enablement tools. In this chapter, we walk you through a complete migration scenario to illustrate how easy the new compatibility features have made the conversion process.

As part of the scenario, we show how to install DB2 software and how to create a DB2 database in the Oracle compatibility mode. We then demonstrate moving an Oracle database, Sales, with a PL/SQL application to DB2 using IBM Data Movement Tool (DMT).

This chapter covers the following topics:

- ▶ Installing DB2 and creating an instance
- ▶ Enabling SQL compatibility
- ▶ Creating and configuring the target DB2 database
- ▶ Defining an additional database user with the Database Administrator authority
- ▶ Enabling using the IBM Data Movement Tool
- ▶ Verification of enablement

4.1 Installing DB2 and creating an instance

Regardless of the platform on which DB2 9.7 will be installed, you must consider and satisfy all the necessary hardware and software requirements. We recommend that you review the installation requirements and options for this task, which can be found in the DB2 Information Center at:

<http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/topic/com.ibm.db2.luw.qb.server.doc/doc/r0025127.html>

The following DB2 documents are also good sources of detailed information about this topic:

- ▶ *Getting Started with DB2 installation and administration on Linux and Windows*, GC10-4247
- ▶ *Quick Beginnings for DB2 Servers*, GC10-4246
- ▶ *Quick Beginnings for DB2 Clients*, GC10-4242

Note: For the most recent information about software requirements, refer to: <http://www.ibm.com/software/data/db2/udb/sysreqs.html>

The example scenario environment for the DB2 database server is on a machine running Red Hat Linux.

4.1.1 Install DB2 using db2setup

First, you must obtain the DB2 software. If you do not have a Passport Advantage® account, you can download DB2 in Try and Buy mode, which works for 90 days after installation. Use the following address to download the application:

<http://www-01.ibm.com/software/data/db2/9/download.html>

We recommend installing DB2 through the Setup Wizard GUI. However, you also can install DB2 using the command-line mode or through a silent (hands free) mode.

For more information about each installation method, please refer to the Information Center, found at:

<http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/index.jsp?topic=/com.ibm.db2.luw.qb.server.doc/doc/c0008711.html>

After you download and unpack DB2 Enterprise Server Edition 9.7 for Linux, UNIX, and Windows, run the `.\ESE\setup.exe` command using the Administrator user ID. This command launches DB2 Setup Launchpad, as shown in Figure 4-1.

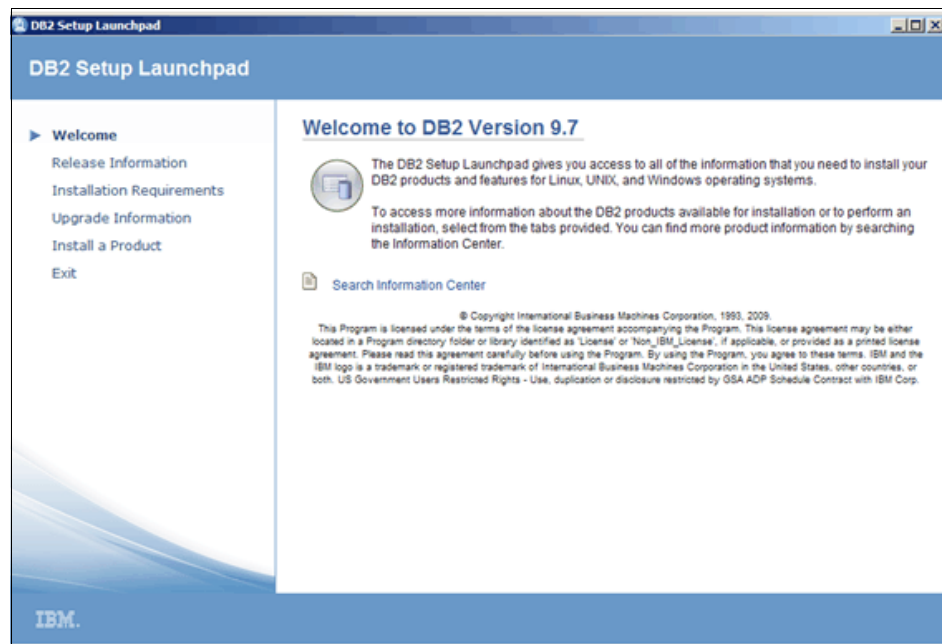


Figure 4-1 DB2 Setup Launchpad

Select the **Install a Product** option and proceed through the DB2 Setup wizard installation window, accepting the default values as you proceed. For the DB2 administrator, provide a new user ID (db2inst1), which will be created as part of the installation.

At the end of the installation, the DB2 setup wizard opens the window where you confirm that the installation is successful.

4.2 Enabling SQL compatibility

As discussed in Chapter 2, “Language compatibility features” on page 21, we must enable our DB2 instance in the Oracle Compatibility mode before we create the DB2 database. You do this task by setting two registry variables, DB2_COMPATIBILITY_VECTOR and DB2_DEFERRED_PREPARE_SEMANTICS. After the compatibility is set, restart the instance by running the **db2stop** and **db2start** commands, as shown in Example 4-1.

Example 4-1 Enabling Oracle Database compatibility features

```
/home/db2inst1>db2set DB2_COMPATIBILITY_VECTOR=ORA
/home/db2inst1>db2set DB2_DEFERRED_PREPARE_SEMANTICS=YES
/home/db2inst1>db2set -all
[i] DB2_COMPATIBILITY_VECTOR=ORA
[i] DB2_DEFERRED_PREPARE_SEMANTICS=YES
[i] DB2COMM=tcPIP
[g] DB2INSTDEF=db2inst1
/home/db2inst1>db2stop
SQL1064N  DB2STOP processing was successful.
/home/db2inst1> db2start
SQL1064N  DB2STOP processing was successful.
```

4.3 Creating and configuring the target DB2 database

Example 4-2 shows the script that contains the CREATE DATABASE command, as well as some post-creation commands. This script is run while logged in as user db2inst1 on the database server. As such, the CONNECT TO command does not require that you supply a user ID and password, because they are taken implicitly from the operating system.

Example 4-2 Script to create the example scenario database

```
CREATE DATABASE sales PAGESIZE 32 K AUTOMATIC STORAGE /home/db2inst1 restrict;

CONNECT TO sales;
CREATE USER TEMPORARY TABLESPACE user_temp;
UPDATE DB CFG USING auto_reval deferred_force;
UPDATE DB CFG USING decflt_rounding round_half_up;
TERMINATE;
```

To run this script, use the following command:

```
db2 -tvf <scriptname>
```


Important: By default, a DB2 database is created in Unicode. To ensure that the DB2 Oracle Compatibility feature functions correctly, you should create the database in Unicode.

The command parameters that we use for this enablement example scenario include some DB2 features that help simplify database design, future management, and initial tuning. There are also some automation features that we use that are enabled by default (no syntax is required to benefit from them). Here is a description of some of these features:

► **AUTOMATIC STORAGE**

When you use this feature (enabled by default), it is not necessary to configure the table space sizes or placement or concern ourselves with the maintenance of the table spaces as they grow in size. On Linux and AIX platforms, DB2 creates databases and their table spaces under the instance owner's home directory by default (for example, `/home/db2inst1`) on a path specified by the user.

► **PAGESIZE 32 K**

The default page size used by the `CREATE DATABASE` command is 4 KB. We override this default by selecting the largest page size available. This is in order to handle tables being migrated with both small and large row sizes.

► **RESTRICTIVE**

This parameter indicates that no privileges should be granted to the group `PUBLIC` by default. This provides for a higher level of security. We chose not to implement this option in our example scenario.

The following changes are made to the database after it is created:

► **GLOBAL TEMPORARY TABLESPACE**

We create user temporary table space in order to allow the future creation of a `GLOBAL TEMPORARY TABLE` in the example scenario. Note that because we use the IBM DMT for enablement, and that tool also creates a `GLOBAL TEMPORARY TABLE`, this command could have been skipped here.

► **AUTOMATIC REVALIDATION**

The database configuration parameter `AUTO_REVAL` is changed to `DEFERRED_FORCE`, as discussed in 2.1.1, "SQL compatibility setup" on page 23.

► **DECIMAL FLOAT POINT ROUNDING**

Another database configuration parameter, `DECFLT_ROUNDING`, is also set to more closely mimic Oracle behavior.

Some additional considerations:

- Memory management

In our example scenario environment, we make full use of the DB2 Self-Tuning Memory Manager (STMM) and autonomic features. By default, many DB2 memory heaps are self tuned by STMM, including buffer pool sizes, sort memory, hash memory, and lock memory. STMM dynamically adjusts both the total memory consumption of the database and the distribution of that memory to various purposes and needs. This means that the STMM is free to automatically adjust their sizes as needed, balancing between different requirements. These adjustments happen at run time every few minutes in response to changing workload demands.

- Physical database design and storage management

Physical database design practices for DB2 are similar to those in Oracle, although there are some differences. For example, DB2 provides Multi Dimensional Clustering that is unavailable in Oracle, while Oracle provides bitmap indexes and list partitioning. Similarly, DB2 provides automatic storage to simplify the management of data storage across several devices and file systems. While similar in principle to Oracle's storage models, the DB2 features have some different terminology and semantics that are worth knowing. For an overview of best practices for both physical database design and database storage management, we recommend you visit the DB2 Best Practices Web site, found at:

<http://www.ibm.com/developerworks/data/bestpractices/>

The best practices paper for physical database design covers best practices for designing indexes, materialized query tables (materialized views), data clustering and multi-dimensional clustering, range partitioning, hash partitioning, using sampling and counting in database design, and making use of EXPLAIN to improve design choices. The best practices for database storage covers guidelines and recommendations for spindles and logical unit numbers (LUNs), stripe and striping, transaction logs and data, file systems versus raw devices, Redundant Array of Independent Disks (RAID) devices, registry variable and configuration parameter settings, and automatic storage.

4.4 Defining an additional database user with the Database Administrator authority

Connections to the DB2 database for our enablement scenario environment are made with user *sales*, which has Database Administrator (DBADM) authority.

Although there are various authentication methods available, such as LDAP, Kerberos, or client authentication, the most common method is server authentication. For server authentication, the database user ID and password are authenticated using the database server's operating system mechanism. Therefore, before working with this new user in the database, we must first create the user *sales* in the operating system using this Linux command:

```
useradd -g db2iadm1 -m -d /db2home/sales sales
```

Once this user is created, we grant the DBADM authority by way of a db2inst1 Shell session as follows:

```
/home/db2inst1>db2 "CONNECT TO sales"  
/home/db2inst1>db2 "GRANT DBADM ON DATABASE TO USER sales"
```

4.5 Enabling using the IBM Data Movement Tool

In this section, we demonstrate an enablement scenario by using the IBM Data Movement Tool running on our DB2 database Linux server. We use DMT for moving DDL, table data, and deploying PL/SQL objects.

The DDLs and data used for this scenario are available for download at the IBM Redbooks Web site. Refer to Appendix F, “Additional material” on page 317 for the download details. We also include the DDLs in Appendix E, “Test cases” on page 279.

4.5.1 Getting started

Download the latest IBM Data Movement Tool from the link in the developerWork article and unpack the archive file into the `/home/db2inst1/IBMDDataMovementTool` directory. We also copy the DB2 and Oracle JDBC Drivers into this directory.

After we launch the GUI version of this tool by executing the `IBMDataMovementTool.sh` script, we go to the Extract/Deploy tab (Figure 4-2) and enter the database connection information and location of JDBC drivers both for the source Oracle source and the DB2 target databases.

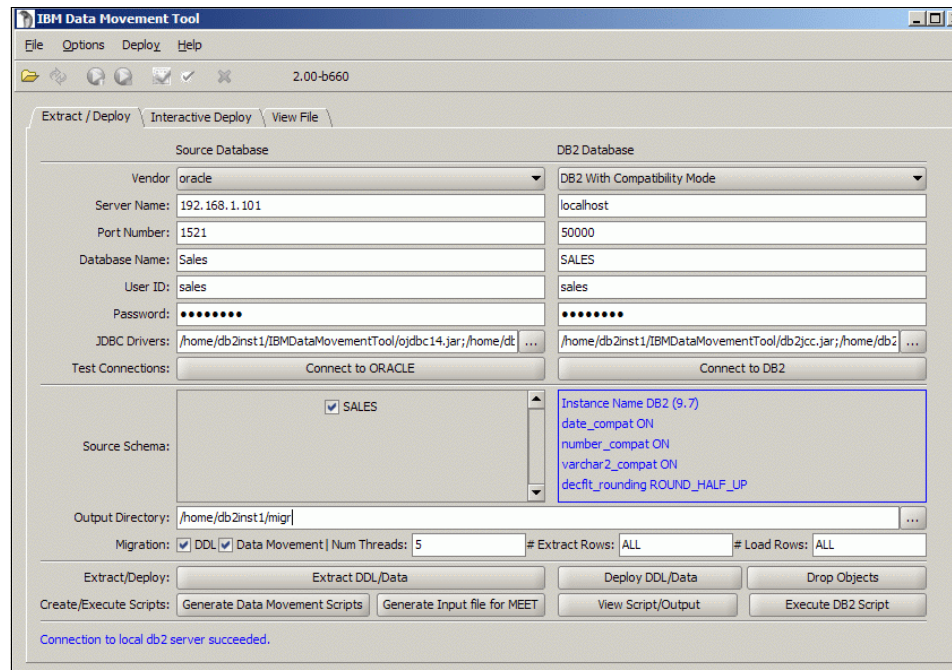


Figure 4-2 The IBM DMT with database connection information already entered

4.5.2 Extracting DDL, table data, and PL/SQL objects

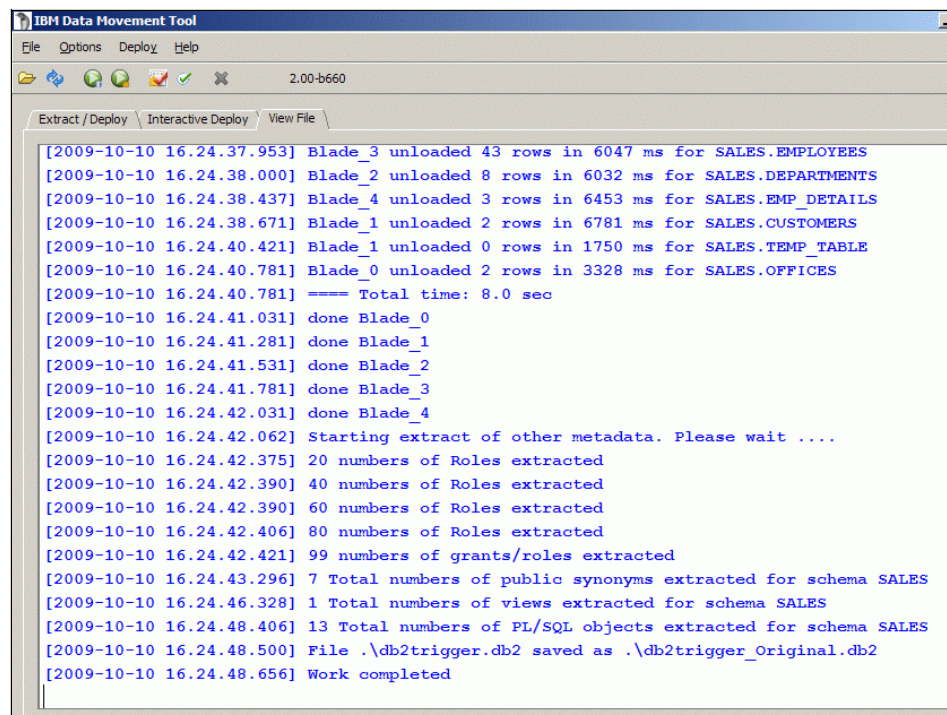
When the **Connect to ORACLE** button is clicked, two things happen: The list of the existing schemas in the Source Schema area appears and the Extract DDL/Data button becomes active. If there are any errors, we can find error details in the command window console that was opened by the tool (the window might be behind the GUI).

Our example database uses Oracle multi-action triggers that are not available in DB2. We select the **Split multiple action triggers** option in the Options menu along with the other recommended settings discussed in 3.1, “IBM Data Movement Tool” on page 144.

Before proceeding with the extraction from Oracle, we first connect to DB2 so that we can tailor the scripts that are built for the version of our target DB2 database. Notice that DB2 With Compatibility Mode had been selected in the target database.

Once the databases are connected, we are ready to extract the Oracle DDL and data. We click the **Extract DDL/Data** button to extract both the DDL and data.

We monitor the process through the View File tab (Figure 4-3). At the end of the process, the tool displays the message “Work completed”. We also see newly created scripts in the directory we specified (/home/db2inst/migr). The script contains the extracted and modified DDL statements, flat files with table data, and individual files for each type of PL/SQL objects.



The screenshot shows the IBM Data Movement Tool window. The 'View File' tab is active, displaying a log of the extraction process. The log includes timestamps, progress updates for various tables (Blade_0 to Blade_4), a total time of 8.0 seconds, and the completion of metadata extraction for schema SALES.

```
[2009-10-10 16.24.37.953] Blade_3 unloaded 43 rows in 6047 ms for SALES.EMPLOYEES
[2009-10-10 16.24.38.000] Blade_2 unloaded 8 rows in 6032 ms for SALES.DEPARTMENTS
[2009-10-10 16.24.38.437] Blade_4 unloaded 3 rows in 6453 ms for SALES.EMP_DETAILS
[2009-10-10 16.24.38.671] Blade_1 unloaded 2 rows in 6781 ms for SALES.CUSTOMERS
[2009-10-10 16.24.40.421] Blade_1 unloaded 0 rows in 1750 ms for SALES.TEMP_TABLE
[2009-10-10 16.24.40.781] Blade_0 unloaded 2 rows in 3328 ms for SALES.OFFICES
[2009-10-10 16.24.40.781] ==== Total time: 8.0 sec
[2009-10-10 16.24.41.031] done Blade_0
[2009-10-10 16.24.41.281] done Blade_1
[2009-10-10 16.24.41.531] done Blade_2
[2009-10-10 16.24.41.781] done Blade_3
[2009-10-10 16.24.42.031] done Blade_4
[2009-10-10 16.24.42.062] Starting extract of other metadata. Please wait ....
[2009-10-10 16.24.42.375] 20 numbers of Roles extracted
[2009-10-10 16.24.42.390] 40 numbers of Roles extracted
[2009-10-10 16.24.42.390] 60 numbers of Roles extracted
[2009-10-10 16.24.42.406] 80 numbers of Roles extracted
[2009-10-10 16.24.42.421] 99 numbers of grants/roles extracted
[2009-10-10 16.24.43.296] 7 Total numbers of public synonyms extracted for schema SALES
[2009-10-10 16.24.46.328] 1 Total numbers of views extracted for schema SALES
[2009-10-10 16.24.48.406] 13 Total numbers of PL/SQL objects extracted for schema SALES
[2009-10-10 16.24.48.500] File .\db2trigger.db2 saved as .\db2trigger_Original.db2
[2009-10-10 16.24.48.656] Work completed
```

Figure 4-3 Monitoring DDL/Data extraction

4.5.3 Deploying DDL and table data into a DB2 target database

Once the Oracle data is successfully extracted, we are now ready to deploy the Oracle objects to the target DB2 database.

We use the Deploy DDL/Data button in the same Extract/Deploy window to initiate this process. Note that as part of this operation, we only create tables, indexes, and sequences, and populate the tables with data. PL/SQL objects are not deployed as part of this operation.

In our case, all tables and data were moved successfully. We receive a success message, as shown in Figure 4-4.

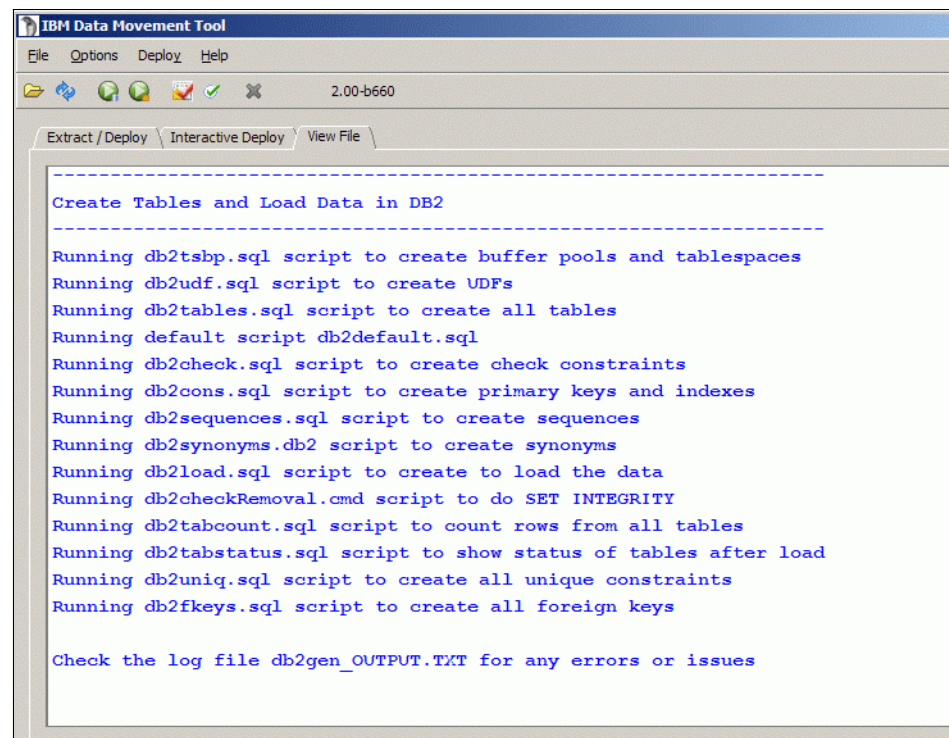



Figure 4-4 Table and data deployed

4.5.4 Deploying PL/SQL objects into a DB2 target database

The PL/SQL objects are extracted as part of the Extract DDL/Data process, but are not deployed automatically due to the manual changes that might be required. We use the Interactive Deploy window of the IBM Data Movement Tool to deploy PL/SQL objects.

To show the list of available objects, click the **Refresh** button . The DB2 Objects tree is populated with the different types of objects, as shown in Figure 4-5.

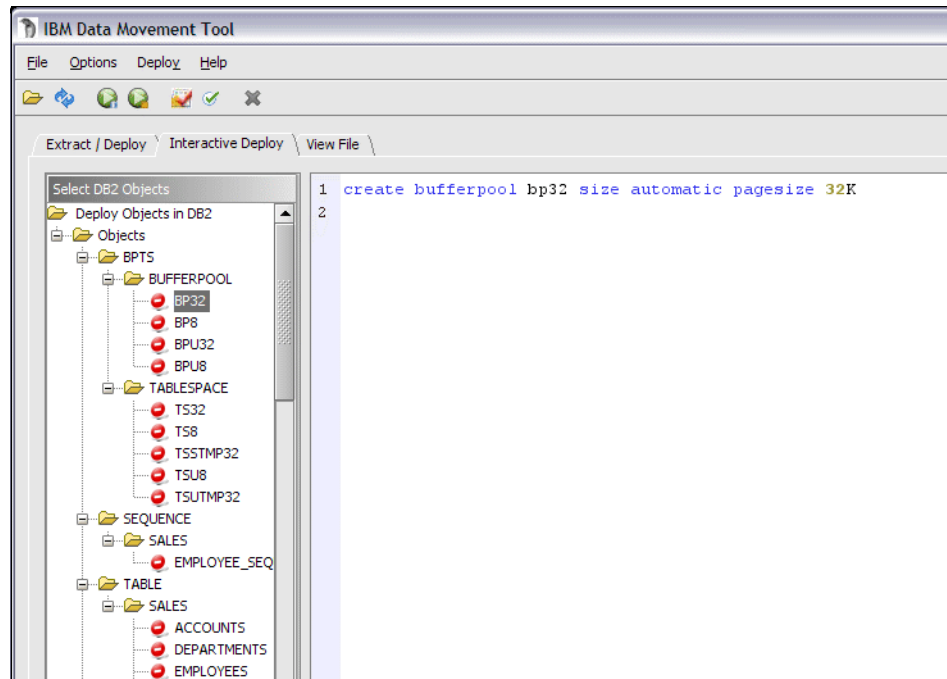


Figure 4-5 DMT Interactive Deploy window after refresh

To start the deployment, click the **Deploy All object** button, as shown in Figure 4-6.

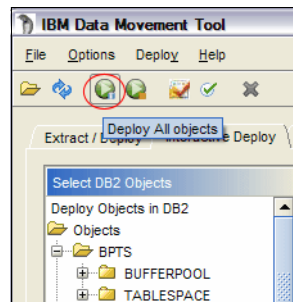


Figure 4-6 DMT Deploy All objects

During the deployment, the object tree will scroll down as each definition is executed. After only a few seconds, we see the results in the lower right bottom pane of the GUI, as shown in Figure 4-7.

Type	Schema	Object Name	Status	SQL ...	Line #	Message
TABLE	SALES	DEPARTMENTS	✓			Deployed
TABLE	SALES	EMPLOYEES	✓			Deployed
TABLE	SALES	EMP_DETAILS	✓			Deployed
TABLE	SALES	OFFICES	✓			Deployed
TABLE	SALES	TEMP_TABLE	✓			Deployed
DEFAULT	SALES	ACCOUNTS_CLOS...	✓			Deployed
DEFAULT	SALES	ACCOUNTS_CREA...	✓			Deployed
DEFAULT	SALES	EMPLOYEES_CREA...	✓			Deployed
CHECK_C...	SALES	CK1_EMPLOYEES	✓			Deployed
PRIMARY...	SALES	PK_ACCOUNTS	✓			Deployed
PRIMARY...	SALES	PK_DEPARTMENTS	✓			Deployed
PRIMARY...	SALES	PK_EMPLOYEES	✓			Deployed
PRIMARY...	SALES	PK_EMP_DETAILS	✓			Deployed
PRIMARY...	SALES	PK_OFFICES	✓			Deployed
INDEX	SALES	IX1_ACCOUNTS	✓			Deployed
INDEX	SALES	IX2_CUSTOMERS	✗	-205	0	SQL0205N Column or attribute "SYS_N...
INDEX	SALES	IX3_EMPLOYEES	✓			Deployed
FOREIGN...	SALES	FK1_EMP_DETAILS	✓			Deployed
FOREIGN...	SALES	FK2_ACCOUNTS	✓			Deployed
FOREIGN...	SALES	FK3_EMPLOYEES	✓			Deployed
FOREIGN...	SALES	FK4_EMPLOYEES	✓			Deployed
FOREIGN...	SALES	FK5_EMPLOYEES	✓			Deployed

Figure 4-7 DMT deployment results

4.5.5 Resolving incompatibilities with Interactive Deploy

We purposely introduced incompatibilities into our enablement scenario in order to illustrate the methods for solving them. For example, if we scroll through the deployment results list (lower right pane of the Interactive Deploy window) to the EMP_INFO_TYPE type, we see that it failed to deploy (SQLCODE -104, unexpected token). By right-clicking that line, we have the option to open the detailed error message in the edit area or to open the DDL source in the edit area and scroll down the object tree to that object (this is the default behavior when you double-click the line). This is shown in Figure 4-8 on page 169.

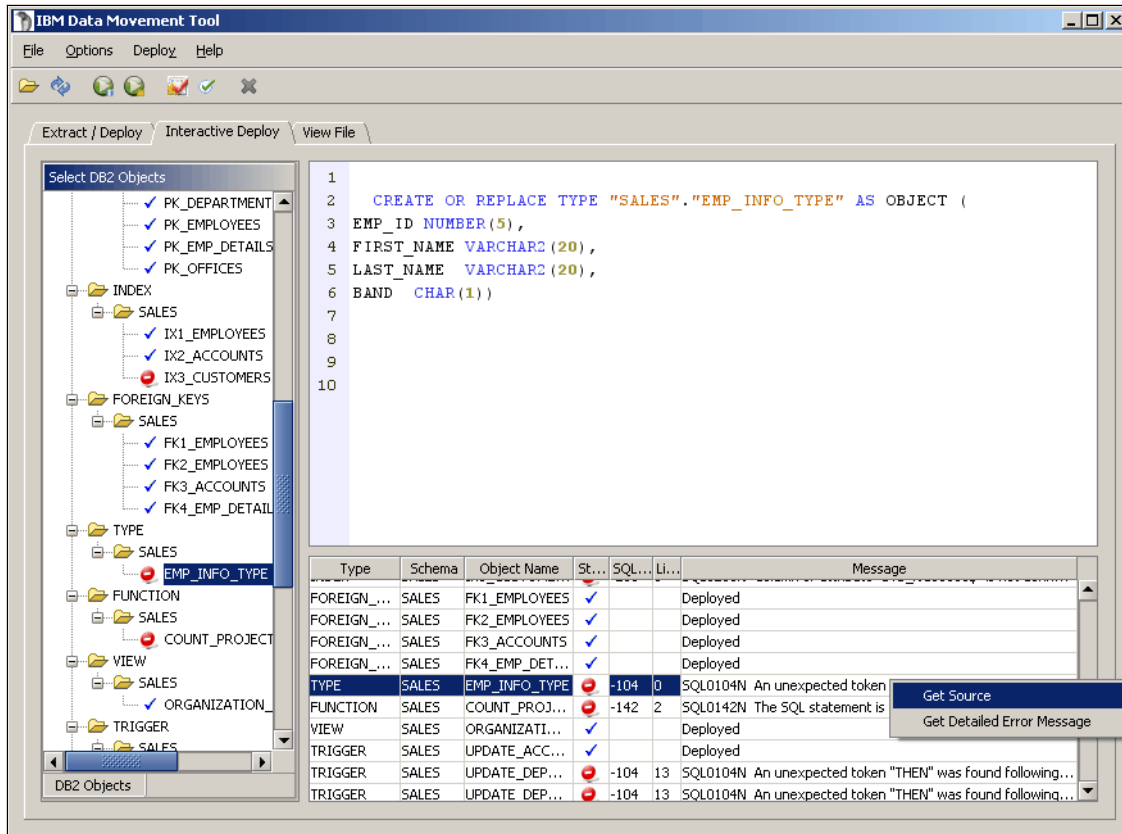


Figure 4-8 DMT: Right-clicking the deployment results gives two options

When we view the detailed error message (Figure 4-9), we see that the problem is related to the TYPE token that follows the CREATE OR REPLACE syntax.

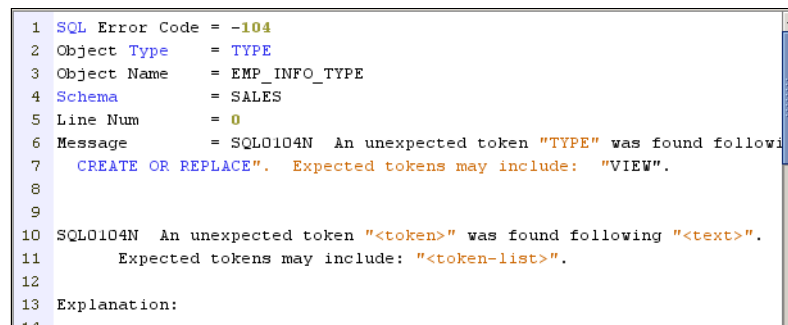


Figure 4-9 Detailed error message displayed in the edit pane

Because the OR REPLACE syntax is not currently available in DB2 for the CREATE TYPE statement, we must remove it. At the same time, we see that AS OBJECT is used. In DB2, this must be replaced with AS ROW. After making these corrections in the edit area, we click **Deploy Selected Objects**. Now the IBM DMT succeeds at deploying the object, as shown in Figure 4-10.

```
1
2 CREATE OR REPLACE PACKAGE "SALES"."HELPER" AS
3 /*
4 || -----
5 || DESCRIPTION: the purpose of this package is to create types that can be used
6 ||
7 ||
8 || DEMO PURPOSE: Definition of package for creating new types, Reference Cursor,
9 ||                 Record data type, Variable arrays
10 ||
11 || -----
12 */
13
14 -- type declaration
15 TYPE rct1 IS REF CURSOR;
16 TYPE EMP_INFO_TYPE IS RECORD (
17     EMP_ID      NUMBER(5),
18     FIRST_NAME  VARCHAR2(20),
19     LAST_NAME   VARCHAR2(20),
20     BAND        CHAR(1));
21 TYPE emp_array_type IS VARRAY(10) OF EMP_INFO_TYPE;
22
23
24 END HELPER;
25 /
```

Type	Schema	Object Name	Status	SQL Code	Line #
PACKAGE	SALES	HELPER	✓	Deployed	
Deployed=1	Failed=0	Discard=0	✓		

Figure 4-10 DMT deployment of the selected object after correcting the syntax

Definitions that have been modified are saved as scripts in the <database name>\savedobjects directory. For our example, the script was named sales_helper_package.sql. Other exceptions that have been fixed in the Interactive Deploy window are:

- ▶ Nested function AVERAGE_BAND was moved from the ACCOUNT_PACKAGE.DISPLAY_ACCOUNT_LIST procedure to the ACCOUNT_PACKAGE body
- ▶ The Oracle CREATE DIRECTORY command was replaced with the corresponding DB2 command UTL_DIR.CREATE_DIRECTORY.

- ▶ The Oracle DBMS_SQL.BIND_VARIABLE command was replaced with corresponding DB2 commands DBMS_SQL.BIND_VARCHAR and DBMS_SQL.BIND_CHAR.
- ▶ The Oracle DBMS_LOB.WRITE and DBMS_LOB.APPEND commands were replaced with the corresponding DB2' commands DBMS_LOB.WRITE_CLOB and DBMS_LOB.APPEND_CLOB.
- ▶ The Oracle XMLType and EXTRACT syntax to handle XML data were replaced with the DB2 native XML data type and XQUERY.

Once the above modifications were made, all PL/SQL objects were able to compile in DB2 successfully.

4.6 Verification of enablement

In order to verify that the database objects and PL/SQL code were enabled successfully, we run the anonymous block that simulates the application run. We execute the block in the DB2 CLPPlus interface.

To start CLPPlus and connect to DB2 database at the same time, we run the following command:

```
db2inst1>clpplus sales/password@localhost:50000/sales
Database Connection Information
Hostname = localhost
Database server = DB2/Linux  SQL09070
SQL authorization ID = sales
Local database alias = SALES
Port = 50000
CLPPlus: Version 1.0
Copyright (c) 2009, IBM CORPORATION. All rights reserved.
```

The same anonymous block can be executed from an Oracle database with one slight modification: Instead of using the stand-alone type EMP_INFO_TYPE, we use the type from the package HELPER.EMP_INFO_TYPE.

SQL*Plus' SERVEROUTPUT ON option, also supported in CLPPlus, can be used to display the output from an anonymous block. As you can see in Figure 4-11, our anonymous block executes normally in DB2. The results are consistent with the Oracle version of the same anonymous block, confirming that our application was enabled successfully in DB2.

```
DB250000I: The command completed successfully.
SQL> set serveroutput on
SQL> /
-----
----Account manipulation test--
Account 11 was successfully removed.
Account 11 successfully created .
Employee record id 37 was created successfully.
List of employees
-----
Record id           : 1
Employee            : TRENTON
Number of projects  : 0
Average Band in department : E
```

Figure 4-11 CLPPlus execution of an anonymous block

4.7 Summary

In this chapter, we used an example database running on Oracle and showed how it can be moved to DB2 with minimal effort. Over 98% of the source statements in the source database were compatible with DB2 without modification, while a small number were easily identified and required relatively minor modifications. After this short enablement process, the resulting DB2 database is fully functional.



Application conversion

In this chapter, we discuss some aspects of converting an application from an Oracle environment to a DB2 environment. The applications discussed here are client side applications that are not part of database objects. There are a variety of languages used in applications and each one can have its unique way of using APIs. In this chapter, we explain the necessary steps for converting client side applications from Oracle to DB2.

This chapter covers the following topics:

- ▶ Planning
- ▶ Conversion of self-built applications
 - Written with Oracle Pro*C
 - Written in Java
 - Based on Oracle OCI
 - Based on ODBC
 - Based on Perl
 - Based on PHP
 - Based on .NET

Note that the examples included in this chapter are excerpts from the actual programs, and cannot be compiled and executed by themselves.

5.1 DB2 application development introduction

To develop applications that access the DB2 database, embed the data access method of the high-level language into the application. IBM DB2 provides various programming interfaces for data access and manipulation.

There are various methods for performing data interaction from your application, including embedded static, dynamic SQL, native API calls, and methods provided by DB2 drivers for a specific application environment.

5.1.1 Driver support

DB2 supports numerous drivers for developing more complex applications. The driver manager defines a set of methods, variables, and conventions that provide a consistent database interface specifically for the DB2 database. Applications utilizing drivers are compiled and linked with the driver manager's libraries to invoke standardized APIs.

DB2 currently supports a large number of drivers, including CLI/ODBC, ADO and OLEDB, JDBC, SQLJ, PERL DBI, PHP, and the .NET Data Provider.

Perl database interface

To better understand how the interface works, let us examine the Perl database interface (DBI). A Perl program uses a standard API to communicate with the DBI module for Perl, which supports only dynamic SQL. DBI gives the API a consistent interface to any database that the programmer wishes to use. DBD::DB2 is a Perl module which, when used in conjunction with DBI, allows Perl to communicate with the DB2 database.

Figure 5-1 illustrates the Perl/ DB2 environment.

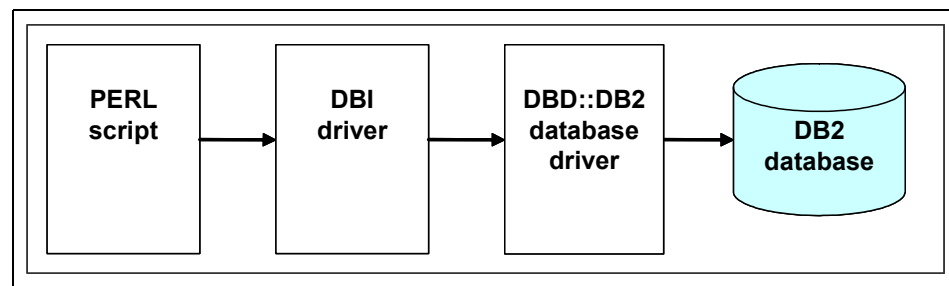


Figure 5-1 Perl/DB2 environment

Installation

You can acquire Perl as follows:

- ▶ The source can be downloaded from the following URL:

<http://www.perl.com>

You can then compile it.

- ▶ A binary version called ActivePerl, available for most operating systems, can be downloaded from ActiveState, found at:

<http://perl.about.com/gi/dynamic/offsite.htm?zi=1/XJ&sdn=perl&z=1/http%3A%2F%2Fwww.activestate.com%2F>

Note: The latest version of ActivePerl, at the time of this writing, is 5.10.0.1004

System requirements

The following requirement information is available at the ActiveState Web site:

<http://perl.about.com/gi/dynamic/offsite.htm?zi=1/XJ&sdn=perl&z=1/http%3A%2F%2Fwww.activestate.com%2F>

- ▶ General
 - Recommended 90 MB hard disk space for typical installation
 - Web browser for online help
- ▶ The following platforms are supported by DB2 9.7:
 - AIX 5L™ V5.1 or later (RS/6000®)
 - Linux: glibc 2.3 or later (x86 and x64)
 - Mac OS X 10.4 or later (x86 and PowerPc)
 - Solaris 2.8 or later (SPARC, 32-, and 64-bit)
 - Solaris 10 or later (x86)
 - Windows 2000 (x86)
 - Windows XP, 2003, and Vista (x86 and x64)

In addition to Perl, two additional modules need to be downloaded and installed in order to enable the Perl driver for DB2:

- ▶ DBI
- ▶ DBD::DB2

These modules can be downloaded from the Comprehensive Perl Archive Network (CPAN), found at:

<http://www.CPAN.org>

Information regarding the installation of these modules may also be found at this location.

Note: For the latest information about Perl and DB2, and related Perl modules, refer to the following Web site:

<http://www.ibm.com/software/data/db2/perl/>

PHP extensions

IBM supports access to DB2 databases from PHP applications through two extensions that offer distinct sets of features:

- ▶ **ibm_db2**

This is an extension written, maintained, and supported by IBM for access to DB2 databases and Cloudscape. It is an optimized driver built on top of the DB2 Call Level Interface driver. It offers a procedural API that, in addition to the normal create, read, update, and write database operations, also offers extensive access to the database meta data. This extension can be compiled with either PHP 4 or PHP 5.

- ▶ **PDO_IBM and PDO_ODBC**

These are drivers for the PHP Data Objects (PDO) extension that offers access to DB2 databases through the standard object-oriented database interface. PDO_IBM is an IBM database driver. Both PDO_IBM and PDO_ODBC extensions can be compiled directly against the DB2 libraries to avoid the communications impact and potential interference of an ODBC driver manager. This extension can be compiled with PHP 5.1.

Installation

An easy method of installing and configuring PHP on Linux, UNIX, or Windows operating systems is Zend Core for IBM, which provides an excellent out-of-the-box experience. Zend Core for IBM can be downloaded and installed for use in production systems from the following Web site:

<http://www.zend.com/downloads>

Additionally, precompiled binary versions of PHP are available for download from:

<http://www.php.net/>

Most Linux distributions include a precompiled version of PHP. On UNIX operating systems that do not include a precompiled version of PHP, your own version of PHP may be compiled.

Requirements

To learn how to set up the PHP environment on Linux, UNIX, or Windows operating systems, refer to *Developing Perl and PHP Applications*; SC10-4234.

5.1.2 Embedded SQL

An SQL statement can be embedded within a host language where SQL statements provide the database interface while the host programming language provides the remaining functionality. Embedded SQL applications require a specific precompiler for each language environment in order to preprocess (or translate) the embedded SQL calls into the host language.

Building embedded SQL applications involves two prerequisite steps prior to application compilation and linking. This is different from building applications with Oracle database access, as Oracle database applications do not have the concept of binding applications to a database prior to run time. An advantage of the static embedded SQL method is that it is more efficient and can yield better performance.

The two prerequisite steps for building DB2 embedded SQL applications are:

1. Preparing the source files containing the embedded SQL statements using the DB2 precompiler

The PREP (PRECOMPILE) command is used to invoke the DB2 precompiler. The precompiler reads the source code, parses and converts the embedded SQL statements to DB2 runtime services API calls, and writes the output to a new modified source file. The precompiler produces access plans for the SQL statements, which are stored together as a package within the database.

2. Binding the statements in the application to the target database

Binding is done, by default, during precompilation (the PREP command). If binding is to be deferred (for example, running the BIND command later), then the BINDFILE option needs to be specified at PREP time in order for a bind file to be generated.

Figure 5-2 shows the precompile-compile-bind process for creating a program with embedded SQL.

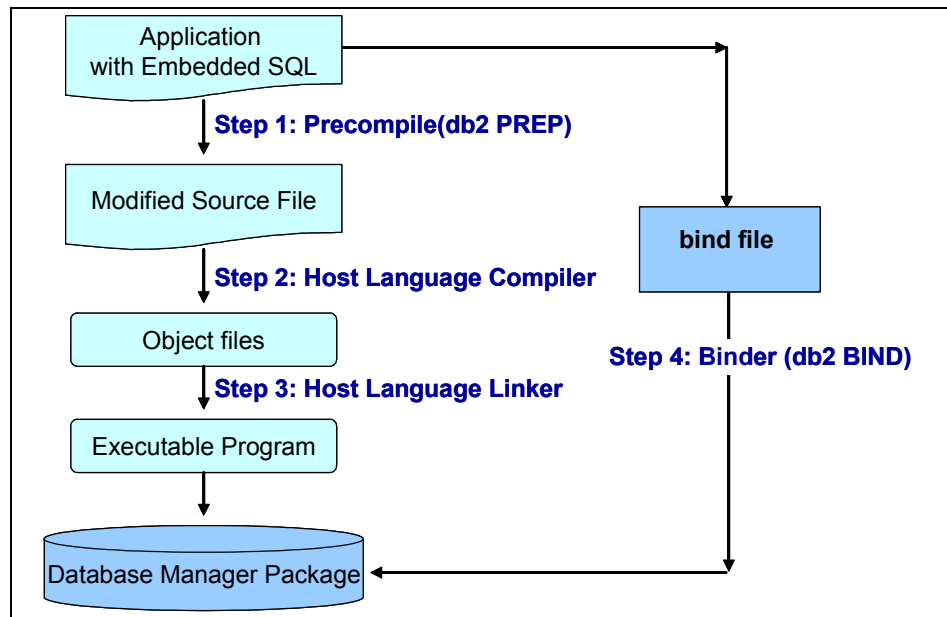


Figure 5-2 Precompile-compile-bind process for creating embedded SQL applications

DB2 supports the C/C++, FORTRAN, COBOL, and Java (SQLJ) programming languages for embedded SQL.

Embedded SQL applications can be categorized as follows:

- Static embedded SQL

In embedded SQL, you are required to specify the complete SQL statement structure. This means that all the database objects (including columns and table) must be fully known at the precompile time with the exception of objects referenced in the SQL WHERE clause. However, all the host variable data types still must be known at precompiler time. Note that host variables, which are declared variables that allow programs to communicate with the database, should be declared in a separate EXEC SQL DECLARE section and be compatible with DB2 data types.

When you use static SQL, you cannot change the form of the SQL statements, but using host variables increase the flexibility of the static SQL statements.

Example 5-1 shows a fragment of a COBOL program with static embedded SQL.

Example 5-1 A COBOL static embedded SQL program

```
move "Clerk" to job-new;
move "Mgr" to job-old
EXEC SQL UPDATE staff SET job=:job-new
        WHERE job=:job-old
END-EXEC.
move "UPDATE STAFF" to errloc.
```

► **Dynamic embedded SQL**

If not every database object in the SQL statement is known at precompile time, use dynamic embedded SQL. The dynamic embedded SQL statement accepts a character string host variable and a statement name as arguments. These character string host variables serve as placeholders for the SQL statements to be executed later. Note that dynamic SQL statements are prepared and executed during program run time.

In other words, the dynamic SQL is a good choice when you do not know the format of an SQL statement before you run a program.

Example 5-2 shows a fragment of a C program with a dynamic SQL statement.

Example 5-2 A dynamic SQL C program

```
EXEC SQL BEGIN DECLARE SECTION;
char st[80];
char parm_var[19];
EXEC SQL END DECLARE SECTION;

strcpy( st, "SELECT tablename FROM syscat.tables" );
strcat( st, " WHERE tablename <> ? ORDER BY 1" );

EXEC SQL PREPARE s1 FROM :st;
EXEC SQL DECLARE c1 CURSOR FOR s1;
strcpy( parm_var, "STAFF" );
EXEC SQL OPEN c1 USING :parm_var;
```

Note that host variable PARM_VAR still needs to be declared in EXEC SQL DECLARE SECTION.

5.2 Application enablement planning

Application enablement is another major step in the enablement project. This process includes:

- ▶ Checking software and hardware availability and compatibility
- ▶ Educating developers and administrators
- ▶ Analyzing application logic and source code
- ▶ Setting up the target environment
- ▶ Changing database specific items
- ▶ Application testing
- ▶ Application tuning
- ▶ Roll-out
- ▶ User education

Planning includes the creation of a project plan. Plan enough time and resources for each task. IBM and our Business Partners can help you with questions in order to define a well prepared project.

For applications developed in-house, the enablement effort for those applications belongs to the enablement team. For package applications, you can contact the vendor for a recommended enablement process.

Checking software and hardware availability and compatibility

The architecture profile is one of the outputs of the first task of enablement planning assessment. While preparing the architecture profile, you need to check the availability and compatibility of all involved software and hardware in the new environment.

Educating developers and administrators

Ensure that the staff has the skills for all products and the system environment you will use for the enablement project. Understanding the new product is essential for analyzing the source system.

Analyzing application logic and source code

In this analysis phase, you should identify all the affected sources. With the availability of DB2 9.7, the new Oracle Database compatibility features have closed most of the gaps. Some slight differences still exist with the Oracle Data Dictionary, Optimizer hints, Scalar functions, and certain SQLs. You also need to analyze the database calls within the application for the usage of database API.

Setting up the target environment

The target system, either the same or a different one, has to be set up for application development. The environment can include:

- ▶ The Integrated Development Environment (IDE)
- ▶ Database framework
- ▶ Repository
- ▶ Source code generator
- ▶ Configuration management tool
- ▶ Documentation tool

A complex system environment usually consists of products from different vendors. Check the availability and compatibility before starting the project.

Changing database-specific items

Regarding the use of the database API, you need to change the database calls in the applications. The changes include:

- ▶ Language syntax changes

The syntax of database calls varies in the different programming languages. In the next few sections, we discuss the varieties of C/C++, Java, and other applications. For information regarding other languages, contact IBM Technical Sales.

- ▶ SQL query changes

Oracle supports some nonstandard SQL extension queries, such as inclusion of optimizer hints or unnamed columns in a complex, inline SQL syntax. To convert such queries to standard SQL, refer to the DB2 documentation, such as the *SQL Reference, Volume 1*, SC27-2456 and *SQL Reference, Volume 2*, SC27-2457.

You have to modify the SQL queries to the Oracle Data Dictionary as well. The DB2 catalog simulation of Oracle Dictionary covers only the most common dictionary views, as the meta data between Oracle and DB2 is not the same.

- ▶ Changes in calling procedures and functions

Sometimes there is a need to change procedures to functions and vice versa. In such cases, you have to change all the calling commands and the logic belonging to the calls in other parts of the database and the applications.

- ▶ Logical changes

Because of architectural differences between Oracle and DB2, changes in the program flow might be necessary. Most of the changes are related to the different concurrency models.

Application testing

A complete application test is necessary after database conversion and application modification to ensure that the database conversion is correct, and all the application functions work properly.

It is prudent to run the enablement process several times in a development system to guarantee the process, run the same enablement process on a test system with existing test data, and then a copy or subset of productions data, before eventually running the process in production.

Application tuning

Tuning is a continuous activity for the database because data volume, the number of users, and applications change from time to time. After the enablement, application tuning should be applied with particular focus on the architectural differences between Oracle and DB2. For the details, see *DB2 Performance-tuning Guidelines*, SC10-4222 and *DB2 V7.1 Performance Tuning Guide*, SG24-6012.

Roll-out

The roll-out procedure varies and depends on the type of application and the kind of database connection you have. Prepare the workstations with the proper driver (for example, DB2 Runtime Client, ODBC, and JDBC) and server according to the DB2 version.

User education

In case of changes in the user interface, the business logic, or the application behavior because of system improvements, user education is required. Be sure to provide proper user education, because the acceptance of the target system is largely tied to the skills and satisfaction of the users.

5.3 Converting XML queries

Although both SQL/XML and XQuery are each defined by their own particular standards, there are still differences in how Oracle and DB2 have adhered to those standards, on an evolving basis, and as a result, there exists differences in the query, access, and generation of XML content in Oracle and DB2. DB2 has supported XML natively since DB2 9.1, and has continually enhanced the XML support to include non-Unicode database, XML Load utility, sub-document updates, new publishing, and data partitioned environment.

5.3.1 SQL/XML

SQL/XML is an extension of SQL that is part of the ISO SQL specification. The SQL/XML functions invoke XPath or XQuery expressions and are used in SQL statements to access portions of an XML document or to generate (publish) XML data from relational data. Without these functions, an SQL statement can only access a column of XML data at the row level and cannot query at the sub document level.

SQL/XML functions can be categorized into two groups:

- ▶ Those that query and access XML content
- ▶ Those that generate (publish) XML content from SQL data

For those SQL/XML functions that query and access XML content, Oracle provides a set of functions that use XPath to access XML content. Included in these functions are what are known as XMLType methods. The XMLType functions belong to the XMLType data type. Some examples are getStringVal(), getClobVal(), getNumberVal(), getNamespace(), and getBlobVal(). Besides the XMLType methods, Oracle also provides additional SQL/XML functions, such as extract(), existsNode(), and extractValue().

To query XML data, Oracle also supports the ISO/IEC standard SQL/XML functions, XMLQuery, XMLTable, and XMLEExists. These functions, known as the XQuery functions, are also supported on DB2, with the XMLEExists in the WHERE clause.

In addition to the SQL/XML querying functions, both DB2 and Oracle support several other types of functions, such as XMLCast, XMLParse, and XMLSerialize. Oracle also supports the casting function XMLType that converts an XML string value to an XMLType value. XMLType is most similar to DB2's XMLParse function. Oracle also supports XMLType methods (such as getStringVal) to cast XML values to scalar string values. The closest DB2 equivalent to these methods is the XMLCast function.

For those SQL/XML functions that are used to generate XML from relational data, Oracle supports many of the same standard SQL/XML functions that are supported by DB2. These functions are referred to as the “publishing” functions and include XMLElement, XMLAgg, XMLAttribute, XMLConcat, and XMLForest. Oracle also provides an additional set of functions that generate XML from SQL data.

Although none of the Oracle-provided SQL/XML functions and methods are found on DB2, their functionality can be mapped to DB2.

Table 5-1 lists some of the frequently used SQL/XML functions supported by Oracle and maps them to DB2 equivalents.

Table 5-1 SQL/XML function mapping

Oracle SQL/XML	SQL/XML category	Oracle specific	Closest DB2 equivalent	DB2 SQL/XML behavior
existsNode	Access	Yes	XMLEXISTS	Used in a WHERE clause to filter rows returned.
extract	Access	Yes	XMLQUERY	Returns an XML sequence.
extractValue	Access	Yes	XMLQUERY & XMLCAST	Returns an XML value and converts to a number or string scalar value.
getStringVal	Access	Yes	XMLCAST	Converts an XML value to a string or numeric scalar value.
getNumberVal	Access	Yes	XMLCAST	Converts an XML value to a string or numeric scalar value.
XMLCAST	Access	No	XMLCAST	Casts to the specified data type.
XMLEXISTS	Access	No	XMLEXISTS	Used in a WHERE clause to filter rows returned.
XMLQUERY	Access	No	XMLQUERY	Returns an XML sequence.
XMLTABLE	Access	No	XMLTABLE	Returns XML values as a table.
XMLPARSE	Casting	No	XMLPARSE	Casts an XML string into the XML data type.
XMLTYPE	Casting	Yes	XMLPARSE	Casts an XML string into the XML data type.
XMLSERIALIZE	Casting	No	XMLSERIALIZE	Casts an XML sequence into an XML string value.
XMLCONCAT	Generate	No	XMLCONCAT	Returns a sequence that concatenates XML values.
XMLELEMENT	Generate	No	XMLELEMENT	Returns an XML element.
XMLAGG	Generate	No	XMLAGG	Returns a sequence containing non-null XML values.
XMLATTRIBUTES	Generate	No	XMLATTRIBUTES	Generates attributes for an element.

Oracle SQL/XML	SQL/XML category	Oracle specific	Closest DB2 equivalent	DB2 SQL/XML behavior
XMLFOREST	Generate	No	XMLFOREST	Returns a sequence of element nodes.

To convert Oracle SQL/XML functions to DB2, find the most equivalent function(s) on DB2 and then refer to *DB2 9 XML Guide*, SC10-4254 and *DB2 SQL Reference Part 1*, SC10-4249 to assist you with constructing the appropriate syntax for your particular conversion. Syntax differences may even exist when the function is shared by both Oracle and DB2 and part of the ISO/IEC or W3C standard.

We provide some examples to demonstrate these differences.

A sample conversion involving Oracle and DB2 SQL/XML functions is shown in Example 5-3 and Example 5-4.

In Example 5-3, both the `extract()` and `existsNode()` functions are Oracle SQL/XML functions; the reference to the name space value differs from how DB2 references the name space.

The example for DB2 (Example 5-4) uses the wildcard notation (*:), which is prefixed to the XML elements. The wildcard will match any name space specified. Although the wildcard notation is part of the W3C standard, it is not supported by Oracle.

Example 5-3 Using SQL/XML functions in Oracle

```
SELECT extract(info, '/customerinfo//addr', 'xmlns="http://posample.org"')
FROM customer_us
WHERE
    existsnode(info, '/customerinfo//addr[city="Aurora"]',
        'xmlns="http://posample.org")=1;
```

Example 5-4 DB2 conversion of using SQL/XML functions

```
SELECT XMLQUERY('$R/*:customerinfo/*:addr' PASSING info AS "R")
FROM customer
WHERE XMLEXISTS('$R/*:customerinfo/*:addr[*:city="Aurora"]'
    PASSING info as "R");;
```

The following example shows how to specify a name space in DB2's XMLQuery when not using the wildcard notation:

```
SELECT XMLQUERY ('declare default element namespace "http://posample.org";
$R/customerinfo//addr' PASSING INFO AS "R")
```

The next example demonstrates casting of XML values on Oracle and DB2:

► On Oracle:

```
cityxml := incust.extract('/customerinfo//city');  
city := cityxml.extract('//text()').getStringval();
```

► On DB2:

```
cityXml := XMLQUERY('$cust/customerinfo//city' passing inCust as "cust");  
city := XMLCAST(cityXml as VARCHAR(100));
```

When you specify an XPath or XQuery expression in an Oracle SQL/XML function, Oracle executes the expression based on the type of XMLType storage used. If XML is stored in XMLType unstructured storage as a CLOB, then Oracle builds a DOM tree of the XML document in memory to process the XPath expression. If XML is stored in XMLType structured storage, which, under the hood, is represented as object-relational data structures, Oracle rewrites the XPath expression into equivalent SQL statements.

On DB2, the XPath or XQuery expressions are not rewritten into SQL statements and the XML document does not have to be loaded into memory as a DOM tree in order for XML processing by these functions to occur. This is due to the DB2 storage model, where each node is already parsed and in a DOM-like, hierarchical format on disk and can easily be traversed by the XPath and XQuery languages.

5.3.2 XQuery

Although standardization was a goal of the World Wide Web Consortium (W3C) when designing the XQuery language, there are differences in how Oracle and IBM have implemented XQuery within their respective database products.

With DB2, XQuery is a case-sensitive, primary language that can be embedded directly within applications that access a DB2 database, or issued interactively from the DB2 Command Line Processor. An XQuery statement is prefixed with the keyword XQUERY and is not limited to being invoked only from an SQL/XML function. The keyword indicates that the primary language is XQuery.

In XQuery, two DB2-defined functions are used in a query to obtain input XML data from a DB2 database:

- db2-fn:xmlcolumn: For retrieving an XML sequence from the input of an XML column.
- db2-fn:sqlquery: For retrieving a sequence of XML values based on the input of an SQL fullselect statement.

In Oracle, the XQuery statement cannot be embedded directly within SQL applications. In applications, the XQuery language is executed from the functions XMLQuery() and XMLTable(). The XQuery command can only be executed natively from the SQL*PLUS environment. However, before executing XQuery from SQL*PLUS, the environment must be properly initialized; this is accomplished by running an Oracle-provided script. After running this script and setting some additional parameters, the XQuery command can be used.

Oracle provides several XQuery and XPath extension functions that have a prefix of ora. Some of these are ora:view, ora:contains, and ora:replace. The Oracle XQuery extension functions do not map to the DB2-defined XQuery functions mentioned previously. To convert the Oracle XQuery extension functions to DB2, you have to rewrite the XQuery expression. For example, ora:view is used to create XML views on relational data so that the data can be manipulated as an XML document. On DB2, this is accomplished by using the SQL/XML publishing functions.

Oracle supports the standard XQuery functions fn:doc and fn:collection. These functions are used to retrieve a single document or a collection of documents that are stored in files on the Oracle XML DB repository. On DB2, because the XML document is always stored in tables, the db2-fn:xmlcolumn function can be used instead.

DB2 supports the use of the XQuery command interactively as well. The XQuery command can be run from the Command Line Processor (CLP). When run from the CLP, no additional setup is required to run XQuery commands.

Example 5-5 compares the differences between an XMLQuery function on Oracle and DB2.

Example 5-5 XQuery differences

```
-- In Oracle -----
SELECT XMLQUERY('$i/customerinfo//city'
               PASSING incust AS "i" RETURNING CONTENT) INTO cityxml
FROM DUAL;

-- In DB2 -----
SELECT XMLQUERY('$cust/customerinfo//city' PASSING inCust as "cust") INTO
cityxml FROM DUAL;
```

Example 5-6 shows how looping through XML content may be done in an Oracle application.

Example 5-6 Looping through XML content in Oracle

```
CURSOR cur1(vcity IN VARCHAR2) IS
SELECT info from customer_us
WHERE
existsnode(info,'/customerinfo//addr[city="'||vcity||'"]','xmlns="http://posamp
le.org") = 1;
...
FOR c IN cur1(city) LOOP
customer := c.info.extract('//name','xmlns="http://posample.org"');
...
END LOOP;
```

In the DB2 application, the same iterating through XML content may be done, except that the FOR-LOOP is replaced, as DB2 does not currently process XML in the block. See Example 5-7.

Example 5-7 Looping through XML content in DB2

```
CURSOR cur1(vcity IN VARCHAR2) IS
SELECT info from customer
WHERE XMLEXISTS('$R//customerinfo//addr[city="$vcity"]' PASSING info as "R");
...
OPEN cur1(city);
LOOP

    FETCH cur1 INTO customer;
    EXIT
    WHEN cur1%NOTFOUND;
    customer1 := XMLQUERY('$cust/customerinfo//city' passing customer as
"cust");
END
LOOP;
```

Example 5-8 on page 189 and Example 5-9 on page 189 compare the use of XQuery when used in the XMLTABLE function on Oracle and DB2. Note that because the XMLTABLE function is a standard SQL/XML function, the same name space declaration is used for both.

Example 5-8 Using the XMLTABLE function in Oracle

```
select X.*
from customer_us,
     xmltable (XMLNAMESPACES (DEFAULT 'http://posample.org'),
              'for $m in $col/customerinfo
               return $m'
              passing customer_us.info as "col"
              columns
                "CUSTNAME" char(30) path 'name',
                "phonenum" xmltype path 'phone')
as X;
```

Example 5-9 Using the XMLTBLE function in DB2

```
select X.*
from xmltable (XMLNAMESPACES (DEFAULT 'http://posample.org'),
              'db2-fn:xmlcolumn("CUSTOMER.INFO")/customerinfo'
              columns
                "CUSTNAME" char(30) path 'name',
                "phone" xml path 'phone')
as X;
```

5.3.3 Updates and deletes

The initial release of the XQuery language only provided for querying of XML at the subdocument level and did not provide for updating or deleting portions of XML content, since the XQuery Update standards are nearing the final stage of specification. DB2 9.5 has taken the lead and made available the insert/delete/update enhancement at the subdocument level. Oracle 11g has also made available nonstandard functions such as updateXML, insertXML, and deleteXML that are not part of the ISO/IEC or W3C standard.

Consider the XML document shown in Example 5-10; the two examples that follow illustrate the differences in Oracle and DB2 when modifying the XML document at the subdocument level.

Example 5-10 An example XML document

```
<studentinfo xmlns="http://posample.org" Cid="1004">
  <student studentno="1">
    <name>John Smith</name>
    <addr country="Canada">
      <street>5 College Street</street>
      <city>Toronto</city>
      <prov>Ontario</prov>
    </addr>
    <phone>X1111</phone>
  </student>
  <student studentno="2">
    <name>William Jones</name>
    <addr country="Canada">
      <street>10 University Lane</street>
      <city>Toronto</city>
      <prov>Ontario</prov>
    </addr>
    <phone>X2222</phone>
  </student>
</studentinfo>
```

Assume you want to update the address of the student, William Jones, and you also want to remove the information about the phone. In Oracle, these changes can be carried out as shown in Example 5-11.

Example 5-11 An Oracle delete/update subdocument

```
UPDATE students
  SET classdatainfo =
        deleteXML(
            updateXML(classdatainfo,
                '/studentinfo/student[@studentno="2"]/addr/street/text()',
                '999 College Street', 'http://posample.org'),
            '/studentinfo/student[@studentno="2"]/phone', 'http://posample.org')
WHERE
...
```

In DB2 9.5 or later, the changes are carried out in a more standardized way, as shown in Example 5-12 on page 191.

Example 5-12 A DB2 delete/update subdocument

```
UPDATE students set classdatainfo =
XMLQUERY( 'declare default element namespace "http://posample.org";
transform
copy $mystudent := $s1
modify (
do replace $mystudent/studentinfo/student[@studentno="2"]/addr/street with
<street>999 College Drive</street>,
do delete ($mystudent/studentinfo/student/phone )
)
return $mystudent'
passing CLASSDATAINFO as "s1")
WHERE
...
```

DB2 update is more standardized and powerful, whereas Oracle requires nesting of functions.

For a more detailed reference about XML usages in features in DB2, refer to *Oracle to DB2 Conversion Guide for Linux, UNIX, and Windows*, SG24-7048.

5.4 Converting Oracle Pro*C applications to DB2

While many aspects of DB2 application development underwent changes in recent years (stored procedures from C/COBOL/Java to SQL procedure language, support for PL/SQL in user-defined functions, procedures, packages, triggers, in-line SQL, an enriched set of built-in functions, and so on), support for embedding SQL into other host languages (C/C++) has not changed in a practical sense.

Oracle's embedded SQL environment is Oracle Pro*C and supports C/C++ as well as COBOL. It has a precompiler to provide support of embedded SQL in the source program.

This section explains the steps that are needed during application conversion to programs with embedded DB2 SQL calls.

5.4.1 Connecting to the database

There is a difference in how C programs connect to the database. In Oracle, each instance (service name) can manage only one database. DB2 instances can be used to manage multiple databases, so the database name should be explicitly provided by a connection statement.

In order to connect to the Oracle database, you need to specify the Oracle user and the password for that user by running the following command:

```
EXEC SQL CONNECT :user_name IDENTIFIED BY :password;
```

In DB2, you need to specify the database name, user ID, and password for that user ID. So, the above statement will be converted to:

```
EXEC SQL CONNECT TO :dbname USERID :userid PASSWORD :password;
```

Note that dbname, userid, and password need to be declared as host variables.

5.4.2 Host variable declaration

Host variables are C or C++ language variables that are referenced within SQL statements. They allow an application to pass input data to and receive output data from the database manager. After the application is precompiled, host variables are used by the compiler as any other C/C++ variable.

Host variables should be compatible with DB2 data types (accepted by the DB2 precompiler), and must be acceptable to the programming language compiler.

As the C program manipulates the values from the tables using host variables, the first step is to convert Oracle table definitions to DB2 data types.

The next step is to match DB2 data types with C data types.

All host variables in a C program need to be declared in a special declaration section, so that the DB2 precompiler can identify the host variables and the data types:

```
EXEC SQL BEGIN DECLARE SECTION;
    char emp_name[31] = {'\0'};
    sqlint32 ret_code = 0;
EXEC SQL END DECLARE SECTION;
```

Within this declaration section, there are rules for host variable data types that are different from Oracle precompiler rules. Oracle precompiler permits host variables to be declared as VARCHAR. VARCHAR[n] is a pseudo-type recognized by the Pro*C precompiler. It is used to represent blank-padded, variable-length strings. Pro*C precompiler converts it into a structure with a 2-byte length field followed by an n-byte character array. DB2 requires usage of standard C constructs. So, the declaration for the variable emp_name VARCHAR[25] needs to be converted as follows:

```
struct
{
    short var_len;
```



```

    char var_data[25]
} emp_name ;

```

The use of a char emp_name[n] is also permitted for VARCHAR data.

Variables of user-defined types (using typedef) in PRO*C need to be converted to the source data type. For example, type theUser_t has been declared to host values from Oracle object type:

```

typedef struct user_s
{
    short int userNum;
    char userName[25];
    char userAddress[40];
} theUser_t;

```

In a Pro*C program, you can have host variables declared as theUser_t:

```

EXEC SQL BEGIN DECLARE SECTION;
    theUser_t *myUser;
EXEC SQL END DECLARE SECTION;

```

To use this host variable for DB2, you would need to take it out of EXEC SQL DECLARE SECTION and define the host variable MyUser as a structure.

DB2 allows for the host variable to be declared as a pointer with the following restriction: If a host variable is declared as a pointer, no other host variable may be declared with that same name within the same source file.

The host variable declaration char *ptr is accepted, but it does not mean a null-terminated character string of an undetermined length. Instead, it means a pointer to a fixed-length, single-character host variable. This may not be what was intended for the Oracle host variable declaration.

We recommend that sqlint32 and sqlint64 be used for INTEGER and BIGINT host variables, respectively. By default, the use of long host variables results in the precompiler error SQL0402 on platforms where long is a 64-bit quantity, such as 64-bit UNIX. Use the PREP option LONGERROR NO to force DB2 to accept long variables as acceptable host variable types and treat them as BIGINT variables.

One useful DB2 type is the CLOB type, if you need to deal with a very large character string. For example, you can declare:

```

EXEC SQL BEGIN DECLARE;
    SQL TYPE IS CLOB(200K) *statement
EXEC SQL END DECLARE SECTION;

```

You can later populate statement->data with, for example, a long SQL statement, and process it.

Starting with Version 9, DB2 supports the XML type for host variables. In the declarative section of the application, declare the XML host variables as LOB data types, as shown in the following:

```
EXEC SQL BEGIN DECLARE;
      SQL TYPE IS XML as CLOB(N) my_xml_var1;
      SQL TYPE IS XML as BLOB(N) my_xml_var2;
EXEC SQL END DECLARE SECTION;
```

You can learn more about handling XML types within C applications by referring to *DB2 Express-C: The Developer Handbook for XML, PHP, C/C++, Java, and .NET*, SG24-7301.

Example 5-13 shows a fragment of PRO*C that demonstrates this method. The last statement will place all 10 rows from the cursor into arrays.

Example 5-13 Array for host variable

```
EXEC SQL BEGIN DECLARE SECTION;

long int    dept_numb[10];
char        dept_name[10][14];
char        v_location[12];

EXEC SQL END DECLARE SECTION;
/* ..... */

EXEC SQL DECLARE CUR1 CURSOR FOR
      SELECT DEPTNUMB, DEPTNAME
      FROM org_table
      WHERE LOCATION = :v_location;
/*..... */

EXEC SQL FETCH CUR1 INTO :dept_num, :dept_name;
```

Example 5-14 shows the conversion of the arrays host variable declaration.

Example 5-14 Array host variable declaration conversion

```
EXEC SQL BEGIN DECLARE SECTION;

sqlint32    h_dept_numb = 0;
char        h_dept_name[14] = {'\0'};
char        v_location[12] = {'\0'};

EXEC SQL END DECLARE SECTION;
/* move array out of DECLARE section - just C variables */
long int    dept_numb[10];
char        dept_name[10][14];
```

```

short int    i = 0;

/* ..... */

EXEC SQL DECLARE CUR1 CURSOR FOR
    SELECT DEPTNUMB, DEPTNAME
    FROM org_table
    WHERE LOCATION = :v_location;

/*we need Fetch one row at the time and move to corresponding
   member of array */

for (i=0;i<10;i++){
    EXEC SQL FETCH CUR1 INTO :h_dept_num, :h_dept_name;
    if (SQLCODE == 100) {
        break;
    }
    dept_num[i] = h_dept_num;
    strcpy(dept_name[i], h_dept_name);
}

```

5.4.3 Exception handling

The mechanisms for trapping errors are quite similar between Oracle and DB2, using the same concept of separating error routines from the mainline logic. There are three different **WHENEVER** statements that could be used to define program behavior in case of an error in DB2:

```

EXEC SQL WHENEVER SQLERROR GOTO error_routine;
EXEC SQL WHENEVER SQLWARNING CONTINUE;
EXEC SQL WHENEVER NOT FOUND not_found_routine;

```

Although the **WHENEVER** statement is prefixed by **EXEC SQL** like other SQL statements, it is not an executable statement. Instead, a **WHENEVER** statement causes the precompiler to generate code in a program to check the **SQLCODE** attribute from the **SQLCA** after each SQL statement, and to perform the action specified in the **WHENEVER** statement. **SQLERROR** means that an SQL statement returns a negative **SQLCODE** indicating an error condition. **SQLWARNING** indicates a positive **SQLCODE** (except +100), while **NOT FOUND** specifies **SQLCODE** = +100, indicating that no data rows were found to satisfy a request.

A compilation unit can contain as many WHENEVER statements as necessary, and they can be placed anywhere in the program. The scope of one WHENEVER statement reaches from the placement of the statement in the file onward in the character stream of the file until the next suitable WHENEVER statement is found or end-of-file is reached. No functions or programming blocks are considered in that analysis. For example, you may have two different SELECT statements: one must return at least one row, and the other may not return any. You will need two different WHENEVER statements:

```
EXEC SQL WHENEVER NOT FOUND GOTO no_row_error;
EXEC SQL SELECT  address
                INTO  :address
                FROM   test_table
                WHERE  phone = :phone_num;

..... *
EXEC SQL WHENEVER NOT FOUND CONTINUE;
EXEC SQL SELECT  commis_rate
                INTO :rate :rateind
                WHERE prod_id = :prodId;
        if (rateind == -1) rate = 0.15;
.....
```

For the DO and STOP in a WHENEVER statement, convert to GOTO.

Another alternative is to check SQLCODE explicitly after each EXEC SQL statement because that allows more context-sensitive error handling.

5.4.4 Error messages and warnings

The SQL Communication Area (SQLCA) data structure in DB2 is similar to the same structure of Oracle. SQLCA provides information for diagnostic checking and event handling.

To get the full text of longer (or nested) error messages, you need the sqlglm() function:

```
sqlglm(message_buffer, &buffer_size, &message_length);
```

message_buffer is the character buffer in which you want Oracle to store the error message, buffer_size specifies the size of message_buffer in bytes, and Oracle stores the actual length of the error message in *message_length. The maximum length of an Oracle error message is 512 bytes.

DB2 provides its user with a special runtime API function to return an error message based on SQLCODE:

```
rc=sqlaintp(msg_buffer, 1024, 80, sqlca.sqlcode);
```

80 stands for the number of characters after which a line break will be inserted in the message. DB2 will search for word boundaries to place such a line break. 1024 specifies the length of the message buffer, for example, `char msg_buffer[1024]`. As a result of invoking this function, the allocated buffer will contain the descriptive error message, for example:

SQL0433N Value "TEST VALUES" is too long. SQLSTATE=22001.

If you need more information about a particular error, DB2 provides an API function that returns an extended message associated with the specific SQLSTATE:

```
rc=sqlgslt(msg_sqlstate_buffer, 1024, 80, sqlca.sqlcode);
```

As a result of invoking this function, `char msg_sqlstate_buffer[1024]` will contain, for example, the following message:

SQLSTATE 22001: Character data, right truncation occurred; for example, an update or insert value is a string that is too long for the column, or datetime value cannot be assigned to a host variable, because it is too small.

5.4.5 Passing data to a stored procedure from a C program

In Oracle, in order to invoke a remote database procedure (which may be part of an Oracle package), the following statements are used:

```
EXEC SQL EXECUTE
    BEGIN
        Package_name.SP_name(:arg_in1, :arg_in2, :status_out);
    END;
END-EXEC;
```

The value transfer between the calling environment and the stored procedure may be achieved through arguments. You can choose one of three modes for each argument: IN, OUT, or INOUT. For example, the above stored procedure may be declared as:

```
CREATE PACKAGE package_name IS
    PROCEDURE SP_name(
        arg_in1 IN NUMBER ,
        arg_in2 IN CHAR(30),
        status_out OUT NUMBER);
END;
```

When this stored procedure is invoked, values passed from the calling program will be accepted by the stored procedure correspondingly.

The DB2 client application invokes a stored procedure by using the CALL statement, which can pass parameters to the stored procedure and receive parameters returned from the stored procedure. Using the sample example above, it has the following syntax:

```
CALL package_name.SP_name (:arg_in1, :arg_in2, :status_out);
```

As with all SQL statements, you can also prepare CALL statement with parameter markers and then supply values for the markers using SQLDA:

```
CALL package_name.SP_name USING DESCRIPTOR :*psqlda;
```

The SQLDA has to be setup before use. It is very helpful if you have an unknown number of host variables or very many variables, such as 100 or more. Managing single variables in those cases can be very troublesome.

In order to invoke a stored procedure from a C client, the following items need to be in place:

- ▶ A stored procedure needs to be created and registered with the database.
- ▶ A host variable or parameter marker to each IN and INOUT parameter of the stored procedure should be declared and initialized.

Consider this example: The program must give a raise to each employee whose current salary is less than some value. The program will pass that value to a stored procedure, perform an update, and return back the status. The client code in C will look as shown in Example 5-15.

Example 5-15 Passing data to a stored procedure

```
#include <sqlenv.h>

main()
{
    EXEC SQL BEGIN DECLARE SECTION;
        Sqlint32 salary_val=0;
        Sqlint16 salind=1;
        Sqlint16 status=0;
        Sqlint16 statind=0;
    EXEC SQL END DECLARE SECTION;

    EXEC SQL INCLUDE SQLCA;
    EXEC SQL CONNECT TO sample;
    EXEC SQL WHENEVER SQLERROR GOTO err_routine;

    salary_val = getSalaryForRaise();
    statind = -1; /* set indicator variable to -1 */
                /* for status as output-only variable */
}
```

```

EXEC SQL CALL raiseSal(:salary_val :salind, :status :statind);
if (status == 0){
    printf (" The raises has been successfully given \n ");
    EXEC SQL COMMIT;
}
else
    if (status ==1)
        printf (" NO input  values has been provided.\n ");
    else
        if (status == 2)
            printf("Stored procedure failed.\n");

err_routine:
    printf (" SQL Error, SQLCODE =  \n ", SQLCODE);
    EXEC SQL ROLLBACK;
}

```

Note that all host variables that are used as parameters in the statement are declared and initialized in EXEC SQL DECLARE SECTION.

5.4.6 Building a C/C++ DB2 application

DB2 provides sample build scripts for precompiling, compiling, and linking C-embedded SQL programs. These are located in the `sqllib/samples/c` directory, along with sample programs that can be built with these files. This directory also contains the `embprep` script used within the build script to precompile an `*.sqc` file.

To help you compile/link the source files, “build” files are provided by DB2 for each language on supported platforms where the types of programs they build are available in the same directory as the sample programs for each language. These build files, unless otherwise indicated, are for supported languages on all supported platforms. The build files have the `.bat` (batch) extension on Windows, and have no extension on UNIX platforms. For example, `bldmapp.bat` is a script to build C/C++ applications on Windows.

DB2 also provides the `utilemb.sqc` and `utilemb.h` files, containing functions for error handling. In order to use utility functions, the utility file must first be compiled, and then its object file linked during the creation of the target program’s executable. Both the makefile and build files in the sample directories do this for the programs that require error-checking utilities.

For more information about building C applications, see *Application Development Guide: Building and Running Applications V8*, SC09-4825.

5.5 Converting Oracle Java applications to DB2

For Java programmers, DB2 offers two APIs: JDBC and SQLJ.

JDBC is a mandatory component of the Java programming language as defined in the Java 2, Standard Edition (J2SE) specification. To enable JDBC applications for DB2, an implementation of the various Java classes and interfaces, as defined in the standard, is required. This implementation is known as a JDBC driver. DB2 offers a complete set of JDBC drivers for this purpose. They are categorized as IBM Data Server Driver for JDBC and SQLJ (formerly IBM DB2 Driver for JDBC and SQLJ (Type 2 and Type 4).)

SQLJ is a standard development model for data access from Java applications. The SQLJ API is defined in the SQL 1999 specification. The IBM Data Server Driver for JDBC and SQLJ provides support for both JDBC and SQLJ APIs in a single implementation. JDBC and SQLJ can interoperate in the same application. SQLJ provides the unique ability to develop using static SQL statements and control access at the DB2 package level.

The Java code conversion is straight forward. The API itself is well defined and database independent. For example, the database connection logic is encapsulated in standard J2EE DataSource objects. The Oracle or DB2 specific things, such as user name, database name, and so on, are then configured declaratively within the application.

However, you must change your Java source code in regards to:

- ▶ The API driver (JDBC or SQLJ)
- ▶ The database connect string
- ▶ Any incompatible SQL statement

DB2 provides a different method for optimizer directives, but will tolerate and ignore Oracle style optimizer hints that appear in SQL. Although it is not necessary to remove these hints, you should consider doing so to reduce the complexity of your source code.

For complete information regarding the Java environment, drivers, programming, and other relevant information, refer to *Developing Java Applications*, SC10-4233.

5.5.1 Java access methods to DB2

DB2 has rich support for the Java programming environment. You can access DB2 data by putting the Java class into a module in one of the following ways:

- ▶ DB2 Server
 - Stored procedures (JDBC or SQLJ)
 - User-defined functions (JDBC or SQLJ)
- ▶ J2EE Application Servers (such as WebSphere® Application Server)
 - Java ServerPages (JSPs) (JDBC)
 - Servlets (SQLJ or JDBC)
 - Enterprise JavaBeans (EJBs) (SQLJ or JDBC)

5.5.2 JDBC driver for DB2

IBM Data Server Driver for JDBC and SQLJ is the JDBC drive supported for DB2 9.7. The DB2 JDBC Type 2 driver is deprecated. Although your Java applications that use the DB2 JDBC Type 2 driver will function successfully with DB2 Version 9.7, upgrading those applications to the IBM Data Server Driver for JDBC and SQLJ as soon as possible will help you avoid a lack of support in future releases.

The IBM Data Server Driver for JDBC and SQLJ supports:

- ▶ All of the methods that are described in the JDBC 3.0 specifications.
- ▶ SQLJ statements that perform equivalent functions to most JDBC methods.
- ▶ Connections that are enabled for connection pooling. WebSphere Application Server or another application server does the connection pooling.
- ▶ Java user-defined functions and stored procedures (IBM DB2 Driver for JDBC and SQLJ type 2 connectivity only).
- ▶ Global transactions that run under WebSphere Application Server Version 5.0 and above.
- ▶ Support for distributed transaction management. This support implements the Java 2 Platform, Enterprise Edition (J2EE) Java Transaction Service (JTS), and Java Transaction API (JTA) specifications, which conform to the X/Open standard for distributed transactions.

For IBM DB2 Driver for JDBC and SQLJ Type 4 connectivity, the `getConnection` method must specify a user ID and a password, through parameters or through property values. See Example 5-16.

```
getConnection(String url, user, password);
getConnection(String url, java.util.Properties info);
```

```
>>+--jdbc:db2:-----+//server--+-----+--/database----->
'-jdbc:db2j:net:--'          '-:port-'

>--+-----+----->
|      .-----|.
|      v       |
|'-:---property---value---;--+-'
```

Example 5-17 Setting the user ID and password in the user and password parameters

```
// Set URL for data source
String url = "jdbc:db2://puma.torolab.ibm.com:50000/sample";
// Create connection
String user = "db2inst1";
String password = "db2inst1";
Connection con = DriverManager.getConnection(url, user, password);
```

Oracle provides a JDBC OCI driver, among many other JDBC drivers, to enable a Java application to use OCI to access the Oracle database, by way of SQL*NET. In this case, the C layer talks to the Oracle database and then returns to the Java layer. In order to connect from a Java application to an Oracle database using the OCI driver, the following must be done:

- Example 5-18 on page 203 shows an Oracle JDBC connection through OCI.

Example 5-18 Oracle JDBC connection

```
import java.sql.*;
import java.io.*;
import oracle.jdbc.driver.*;

class rsetClient
{
    public static void main (String args []) throws SQLException
    {
        // Load the driver
        DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());

        // Connect to the database
        Connection conn =
            DriverManager.getConnection ("jdbc:oracle:oci8:@oracle","uid","pwd");

        // ...
    }
}
```

In the DB2 environment, there is no support of the JDBC OCI driver. It is really not necessary to execute a Java application using OCI, even though DB2 provides OCI support in DB2 9.7 Fix Pack 1. The earlier DB2 JDBC driver, in fact, talks to DB2 using DB2 Call Level Interface, but the newer DB2 JCC Driver has been completely rewritten to eliminate the DB2 Call Level Interface layer.

In the case of an Oracle application using the Oracle JDBC OCI driver, you can change it directly to use the JDBC driver or the JCC driver. It is *not* necessary to import a JDBC library when connecting to DB2. The registration and connection to DB2 is demonstrated in Example 5-19. Be aware that the parameters for the `getConnection` method will be determined by the connection type.

Example 5-19 DB2 JDBC connection

```
import java.sql.*;

class rsetClient
{
    public static void main (String args []) throws SQLException {

        // Load DB2 JDBC application driver
        try
        {
            // use ONE of the following - depending on the chosen driver:
            // IBM Data Server Driver for JDBC and SQLJ
            Class.forName("com.ibm.db2.jcc.DB2Driver");
            // DB2 JDBC Type 2 Driver for Linux, UNIX and Windows
            //Class.forName("COM.ibm.db2.jdbc.app.DB2Driver");
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
        // Connect to the database
        Connection conn =
            // Use appropriate parameters for getConnection depending on chosen
            driver. //
            DriverManager.getConnection("jdbc:db2://dbname","uid","pwd");
        // ...
    }
}
```

5.5.4 Stored procedure calls

The handling of input and output parameters in stored procedures calls differ between Oracle and DB2. The following examples explain the different kinds of procedure calls, and the usage of parameters and result sets.

Stored procedure with an input parameter

A stored procedure has been created in Oracle as follows:

```
CREATE OR REPLACE PROCEDURE sproc1(
in_parm1 IN INTEGER, out_parm2 OUT VARCHAR2)
```

The same procedure appears in DB2.

The procedure has one input parameter and one output parameter. There is no difference in the call between Oracle and DB2. In both cases, the parameter values need to be set *before* the stored procedure can be executed. Example 5-20 demonstrates this point.

Example 5-20 Java call of Oracle or DB2 procedure with input parameter

```
String procName = "sproc1"
String SP_CALL = "call " + procName + "(:in_parm1, :out_parm2)";

// Connect to the database
Connection conn =
    DriverManager.getConnection (url, userName, password);

CallableStatement stmt;
try {
    stmt = conn.prepareCall(SP_CALL);
    stmt.setInt(1,10);
    stmt.registerOutParameter(2, Types.VARCHAR);
    stmt.execute();
    // ...
}
```

Stored procedure with a result set

The next example shows a procedure without an input parameter, but defines a result set as an output parameter. The result set is an opened cursor defined in the procedure. The rows are fetched in the Java application with a loop.

The Oracle stored procedure is defined as:

```
CREATE OR REPLACE PROCEDURE sproc2(oCursor OUT SYS_REFCURSOR) AS
BEGIN
    open oCursor for select last_name from employees;
END;
```

The output parameter type is registered as `CURSOR` before the procedure is called. See Example 5-21.

Example 5-21 Java call of Oracle procedure with result set

```
String SP_CALL = "{call sproc2(?)}";

// Connect to the database
Connection conn =
    DriverManager.getConnection (url, userName, password);

try {
    CallableStatement stmt = conn.prepareCall(SP_CALL);
    stmt.registerOutParameter (1, OracleTypes.CURSOR);
    stmt.execute();
    ResultSet rs = (ResultSet) stmt.getObject(1);
    while(rs.next())
    {
        System.out.println(rs.getString(1));
        // ...
    }
}
```

With DB2 9.7, when you use the `CURSOR` type, you register with the similar `DB2Types`. See Example 5-22.

Example 5-22 Java call of DB2 procedure with result set

```
String SP_CALL = "{call sproc2(?)}";

// Connect to the database
Connection conn =
    DriverManager.getConnection (url, userName, password);

try {
    CallableStatement stmt = conn.prepareCall(SP_CALL);
    stmt.registerOutParameter (1, DB2Types.CURSOR);
    stmt.execute();
    ResultSet rs = (ResultSet) stmt.getObject(1);
    while(rs.next())
    {
        System.out.println(rs.getString(1));
        // ...
    }
}
```

If you are not using the CURSOR type, then you do not need to register the result set with the method `registerOutParameter()` in the Java application. To get the result set, call the method `getResultSet()` instead of `getObject()`, as demonstrated in Example 5-23.

Example 5-23 Java call of DB2 procedure with result set

```
String SP_CALL = "{call sproc2}";

// Connect to the database
Connection conn =
    DriverManager.getConnection (url, userName, password);

try {
    CallableStatement stmt = conn.prepareCall(SP_CALL);
    ResultSet rs = null;
    stmt.execute();
    rs = stmt.getResultSet();
    while(rs.next())
    {
        System.out.println(rs.getString(1));

        // ...
    }
}
```

Function returning cursor type

A function can return a cursor, just like a procedure, as discussed in “Stored procedure with a result set” on page 205. Consider a PL/SQL function defined as follows:

```
CREATE TYPE CursorType IS REF CURSOR;
CREATE OR REPLACE FUNCTION sfunc4(v_num IN INTEGER)
    RETURN CursorType
```

In Oracle, you can retrieve the cursor `ResultSet` in Java with a special syntax, for example, `SP_CALL` (shown in Example 5-24).

Example 5-24 Oracle function with input parameter and result set

```
String SP_CALL = "{? := call sfunc4(?)}" ;

// Connect to the database
Connection conn =
    DriverManager.getConnection (url, userName, password);

try {
    CallableStatement stmt = conn.prepareCall(SP_CALL);
    stmt.registerOutParameter (1, OracleTypes.CURSOR);
    stmt.setInt(2, 6);
    stmt.execute();
    ResultSet rs = (ResultSet) stmt.getObject(1);
    while(rs.next())
    {
        // ...
    }
}
```

To call a function returning a cursor directly with DB2 Java driver, we recommend that such an Oracle function be converted to a stored procedure in DB2 or wrapped into a procedure that would return the cursor of the function as the first argument of the procedure to minimize changes to the logic in the function. Example 5-25 illustrates a function wrapper.

Example 5-25 Function wrapper

```
CREATE OR REPLACE PROCEDURE myfunction_wrapper(C OUT sys_refcursor ,arg1,
...argn ) IS
BEGIN
    c:= myfunctn(arg1, ...argn );
END;
/

CREATE OR REPLACE FUNCTION myfunction(arg1, ...argn) IS
BEGIN
    ...UDF logic here...
End;
/
```

5.6 Converting Oracle Call Interface applications

The DB2 Call Interface (DB2CI) provides compatibility for the Oracle Call Interface (OCI). OCI is one of the many programming interfaces used by C/C++ developers to program against an Oracle database. OCI is application interface that does not require a precompiler step, which is the case in a Pro*C development environment. DB2 9.7 provides compatible support with Oracle OC, which substantially reduces the complexity of moving an Oracle OCI application to DB2. The support for Oracle OCI applications is provided by the IBM Data Server Driver for DB2CI, which is part of the IBM Data Server Driver Package. In addition, DB2CI provides a tracing facility for application development.

5.6.1 DB2 OCI functions

The DB2CI driver provides extensive support for the commonly used OCI functions. The OCI functions include connection, initializations, handle and descriptor function, binding, define, statement, execution, result set, transaction control, data type (NUMBER, STRING, DATE, and so on) functions, date and time, large object processing, arrays, stored procedure executions, File I/O, and so on. These supported functions have a compatible syntax with the Oracle OCI functions.

The DB2CI driver is evolving and provides the OCI functions listed in Table 5-2 in the first release of the DB2 OCI driver (DB2 9.7 Fix Pack 1). The list may be expanded, but is accurate at the time of the writing of this book.

Table 5-2 DB2 OCI functions

OCIDescribeAny	OCINumberDec	OCIServerDetach
OCIDescriptorAlloc	OCINumberDiv	OCIServerVersion
OCIDescriptorFree	OCINumberExp	OCISessionBegin
OCIEnvCreate	OCINumberFloor	OCISessionEnd
OCIEnvInit	OCINumberFromInt	OCISessionGet
OCIErrorGet	OCINumberFromReal	OCISessionRelease
OCIFileClose	OCINumberFromText	OCIStmtExecute
OCIFileExists	OCINumberHypCos	OCIStmtFetch
OCIFileFlush	OCINumberHypSin	OCIStmtFetch2
OCIFileGetLength	OCINumberHypTan	OCIStmtGetBindInfo

OCIFileInit	OCINumberInc	OCIStmtGetPieceInfo
OCIFileOpen	OCINumberIntPower	OCIStmtPrepare
OCIFileRead	OCINumberIsInt	OCIStmtPrepare2
OCIFileSeek	OCINumberIsZero	OCIStmtRelease
OCIFileTerm	OCINumberLn	OCIStmtSetPieceInfo
OCIFileWrite	OCINumberLog	OCIStringAllocSize
OCIHandleAlloc	OCINumberMod	OCIStringAssign
OCIHandleFree	OCINumberMul	OCIStringAssignText
OCIInitialize	OCINumberNeg	OCIStringPtr
OCILobAppend	OCINumberPower	OCIStringResize
OCILobAssign	OCINumberPrec	OCIStringSize
OCILobClose	OCINumberRound	OCITerminate
OCILobCopy	OCINumberSetPi	OCITransCommit
OCILobDisableBuffering	OCINumberSetZero	OCITransRollback
OCILobEnableBuffering	OCINumberShift	
OCILobErase	OCINumberSign	

If your application uses OCI calls that are not currently available in DB2, such as OCILobCreateTemporary, OCILobFreeTemporary, or OCILobWrite, you can replace these using combinations of supported calls.

Since the DB2 OCI calls are very compatible with Oracle OCI calls, very often you just have to compile and link the application using the DB2 specific include file (db2ci.h), and DB2CI library (libdb2ci.a) (for AIX). We provide a sample DB2 OCI program in Appendix D, “DB2 OCI sample program” on page 273.

5.7 Converting ODBC applications

The Open Database Connectivity (ODBC) is similar to the CLI standard. Applications based on ODBC can connect to the most popular databases. Thus, the application conversion is pretty easy. All the ODBC Core Level, Level 1, and Level 2 functions are supported in DB2 CLI, except for SQLDriver. The SQL Datatypes are same in ODBC, as in DB2 Call Level Interface (DB2 CLI).

You may have to perform the conversion of database-specific items in your application, such as:

- ▶ SQL query syntax changes
- ▶ Possible changes in calling stored procedures and functions
- ▶ Possible logic changes

You also have to do the testing and rollout as well. Your current development environment will be the same. For a more detailed description of the necessary steps, refer to 5.2, “Application enablement planning” on page 180.

5.7.1 Introduction to DB2 CLI

DB2 Call Level Interface (DB2 CLI) is a callable SQL interface to the DB2 family of database servers. It is a C and C++ API for relational database access that uses function calls to pass dynamic SQL statements as function arguments. It is an alternative to embedded dynamic SQL, but unlike embedded SQL, DB2 CLI does not require host variables or a precompiler.

DB2 CLI is based on the Microsoft Open Database Connectivity (ODBC) specification, and the International Standard for SQL/CLI. These specifications were chosen as the basis for the DB2 Call Level Interface in an effort to follow industry standards, and to provide a shorter learning curve for application programmers already familiar with either of these database interfaces. In addition, some DB2-specific extensions have been added to help the application programmer specifically exploit DB2 features.

The DB2 CLI driver also acts as an ODBC driver when loaded by an ODBC driver manager. It conforms to ODBC V3.51.

5.7.2 Setting up the DB2 CLI environment

Runtime support for DB2 CLI applications is contained in all DB2 clients. Support for building and running DB2 CLI applications is contained in the DB2 Application Development (DB2 AD) Client.

The DB2 CLI/ODBC driver will automatically bind on the first connection to the database, provided the user has the appropriate privilege or authorization. The administrator may want to perform the first connection or explicitly bind the required files.

Procedure

In order for a DB2 CLI application to successfully access a DB2 database, perform these steps:

1. Catalog the DB2 database and node if the database is being accessed from a remote client. On the Windows platform, you can use the DB2 CLI/ODBC settings GUI to catalog the DB2 database.
2. Optional: Explicitly bind the DB2 CLI/ODBC bind files to the database with the following command:

```
db2 bind ~/sqllib/bnd/@db2cli.lst blocking all messages cli.msg\  
grant public
```

On the Windows platform, you can use the DB2 CLI/ODBC settings GUI to bind the DB2 CLI/ODBC bind files to the database.

3. Optional: Change the DB2 CLI/ODBC configuration keywords by editing the `db2cli.ini` file.

On the Windows platform, you can use the DB2 CLI/ODBC settings GUI to set the DB2 CLI/ODBC configuration keywords.

5.8 Converting Perl applications

In this section, we discuss using Perl for connecting to Oracle and DB2 databases, and demonstrate the conversion from Oracle to DB2 by some simple Perl programs.

We create a stored procedure and a Perl program to demonstrate the following syntactical differences between Oracle and DB2:

- ▶ Connecting to a database using Perl
- ▶ Calling a stored procedure with an input and an output parameter
- ▶ Returning an output parameter

Example 5-26 is an Oracle stored procedure *Greeting*. It contains an input parameter *name*, and an output parameter *message*.

Example 5-26 Oracle stored procedure Greeting

```
CREATE OR REPLACE PROCEDURE Greeting (name IN VARCHAR2, message OUT VARCHAR2)  
AS  
BEGIN  
message := 'Hello ' || UPPER(name) || ', the date is: ' || SYSDATE;  
END;
```

Example 5-27 shows the Perl program oraCallGreeting.pl. This program connects to the Oracle database, binds the input and output parameters, executes the call to the Greeting stored procedure, and returns the output parameter.

Example 5-27 Oracle Perl program oraCallGreeting.pl

```
#!/usr/bin/perl
use DBI;

$database='dbi:Oracle:xp10g';
$user='sample';
$password='sample';

$dbh = DBI->connect($database,$user,$password);
print " Connected to database.\n";

$name = 'Ariel';
$message;

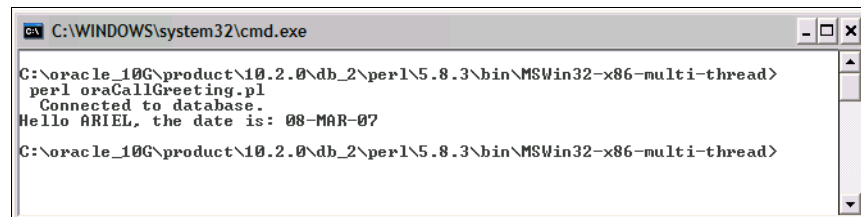
$sth = $dbh->prepare(q{
    BEGIN
        Greeting(:name, :message);
    END;
});

$sth->bind_param(":name", $name);
$sth->bind_param_inout(":message", \$message, 100);
$sth->execute;
print "$message", "\n";

# check for problems ...
warn $DBI::errstr if $DBI::err;

$dbh->disconnect;
```

The results of the execution of oraCallGreeting.pl are shown in Figure 5-3.



```
C:\WINDOWS\system32\cmd.exe
C:\oracle_10G\product\10.2.0\db_2\perl\5.8.3\bin\MSWin32-x86-multi-thread>
perl oraCallGreeting.pl
Connected to database.
Hello ARIEL, the date is: 08-MAR-07
C:\oracle_10G\product\10.2.0\db_2\perl\5.8.3\bin\MSWin32-x86-multi-thread>
```

Figure 5-3 The result of executing the Perl program oraCallGreeting.pl

Converting a Perl application to DB2

In this section, we demonstrate how to connect to DB2 using Perl.

Example 5-28 is a DB2 stored procedure that has the same function and same name as the Oracle stored procedure shown in Example 5-26 on page 212. Since DB2 also supports the Oracle PL/SQL syntax, the procedure looks exactly the same.

Example 5-28 DB2 stored procedure Greeting

```
CREATE OR REPLACE PROCEDURE Greeting (name IN VARCHAR2, message OUT VARCHAR2)
AS
BEGIN
message := 'Hello ' || UPPER(name) || ', the date is: ' || SYSDATE;
END;
```

Minor changes may be necessary to convert the Oracle Perl application (Example 5-27 on page 213) to use DB2, but in this example, it is the same. The steps (besides entering the correct values for user and password) are:

- ▶ Observing the syntax difference in the parameters for the *connect* method, and making the necessary changes.
- ▶ Observing the syntax differences for calling stored procedures, and making the necessary changes.

DB2 Connect method syntax

The syntax for a database connection to DB2 is shown in Example 5-29.

Example 5-29 Generic syntax for a DB2 connection string in a Perl application

```
$dbhhandle = DBI->connect('dbi:DB2:dbalias', $userID, $password)
```

The parameters of this connection are as follows:

- ▶ `dbhhandle`: This represents the database handle returned by the connect statement.
- ▶ `dbalias`: This represents a DB2 alias cataloged in the DB2 database directory.

Oracle requires the `sid` for the database in the place where DB2 would require `dbalias`; the Oracle syntax can be summarized as `dbi:oracle:sid`. In our example, this is coded as `dbi:oracle:xp10g`.

- ▶ `userID`: This represents the user ID used to connect to the database.
- ▶ `password`: This represents the password for the user ID that is used to connect to the database.

Syntax for calling a DB2 stored procedures

In Oracle, a stored procedure is called from an *anonymous block*, that is, BEGIN...END; within a PREPARE statement. The input and output parameters of the Oracle stored procedure are defined as host variables, for example, ::name, :message. Example 5-30 demonstrates these points.

Example 5-30 Calling a stored procedure in an Oracle Perl program

```
$sth = $dbh->prepare(q{  
    BEGIN  
        Greeting(:name, :message);  
    END;  
});
```

In contrast, a DB2 stored procedure is executed by issuing a CALL statement from within a PREPARE statement. Also, the stored procedure input and output parameters are designated as parameter markers (?, ?). This is shown in Example 5-31.

Example 5-31 Calling a stored procedure in a DB2 Perl program

```
$sth = $dbh->prepare(q{  
    CALL Greeting(?,?);  
});
```

The complete Perl program, converted to DB2, is shown in Example 5-32.

Example 5-32 DB2 Perl program db2CallGreeting.pl

```
#!/usr/bin/perl
use DBI;

$database='dbi:DB2:sample';
$user='db2inst1';
$password='db2inst1';

$dbh = DBI->connect($database, $user, $password) or die "Can't connect to
$database: $DBI::errstr";

print " Connected to database.\n";

$name = 'Ariel';
$message;

$sth = $dbh->prepare(q{
    CALL Greeting(?,?);
});

$sth->bind_param(1,$name);

$sth->bind_param_inout(2, \$message, 100);

$sth->execute;

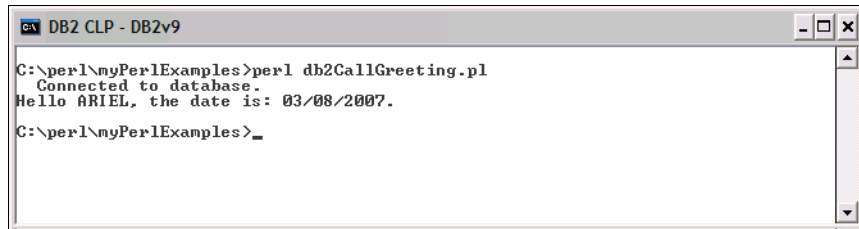
print "$message", "\n";

# check for problems...
warn $DBI::errstr if $DBI::err;

$sth-> finish;

$dbh->disconnect;
```

The results of executing db2CallGreeting.pl are shown in Figure 5-4 on page 217.

A screenshot of a DB2 CLP terminal window. The title bar reads "DB2 CLP - DB2v9". The command prompt shows the user at "C:\perl\myPerlExamples" running "perl db2CallGreeting.pl". The output is "Connected to database." followed by "Hello ARIEL, the date is: 03/08/2007." The prompt then shows a blank line after the command execution.

```
DB2 CLP - DB2v9
C:\perl\myPerlExamples>perl db2CallGreeting.pl
Connected to database.
Hello ARIEL, the date is: 03/08/2007.
C:\perl\myPerlExamples>
```

Figure 5-4 The results of executing the *db2CallGreeting.pl* program

5.9 Converting PHP applications

Oracle supports access to Oracle databases in a PHP application through two extensions:

- ▶ **PDO_OCI**

The PDO_OCI driver implements the PHP Data Objects (PDO) interface to enable access from PHP to Oracle databases through the OCI library.

- ▶ **OCI8**

The functions in this extension allow access to Oracle 9, Oracle 10, and earlier using the Oracle Call Interface (OCI). They support binding of PHP variables to Oracle placeholders, have full LOB, FILE, and ROWID support, and allow you to use user-supplied define variables. This is the *preferred* extension for PHP connections to an Oracle database.

Connecting to Oracle using PDO

Example 5-33 shows the Oracle PHP program *oraGreeting.php*. This program connects to the Oracle default database, because the optional *dbname* is not specified after OCI.

Example 5-33 Connecting to Oracle using PDO

```
<?php
try {
    $dbh = new PDO('OCI:', 'userid', 'password');
    echo "Connected\n";
} catch (Exception $e) {
    echo "Failed: " . $e->getMessage();
}
?>
```

Connecting to DB2 using PDO

Since there is no PDO OCI interface access for DB2, we use the PDO_ODBC driver. The PDO_ODBC implements the PHP Data Objects (PDO) interface to enable access from PHP to databases through ODBC drivers or through the IBM DB2 Call Level Interface (DB2 CLI) library. DB2 connections are established by creating an instance of PDO class. Only the data source name is mandatory. However, you can pass its user ID, password, and additional parameters for passing in tuning information. The program shown in Example 5-34 connects to the DB2 sample database.

Example 5-34 Connecting to DB2 using PDO

```
<?php
try {
    $dbh = new PDO('odbc:SAMPLE', 'userid', 'password');
    echo "Connected\n";
} catch (Exception $e) {
    echo "Failed: " . $e->getMessage();
}
?>
```

Example 5-35 shows the call to Greeting procedure after the connection is established.

Example 5-35 Call a PHP procedure

```
$stmt = $dbh->prepare("CALL Greeting(?, ?)");
$name = 'Ariel';
$stmt->bindParam(1, $name, PDO_PARAM_STR, 100);
$stmt->bindParam(2, $return_msg, PDO_PARAM_OUTPUT, 100);
$stmt->execute();
```

Information regarding the PHP extensions that are available for DB2 are discussed in 5.1.1, “Driver support” on page 174.

In order to demonstrate some differences between PHP programming in Oracle and DB2, we show a sample program that demonstrates the following:

- ▶ Connecting to a database through PHP
- ▶ Calling a stored procedure with an input and an output parameter
- ▶ Returning an output parameter

The same stored procedure, Greeting, that was shown in 5.8, “Converting Perl applications” on page 212, is used in the examples in this section.

Connecting to Oracle using PHP (OCI8)

Example 5-36 shows the Oracle PHP program oraGreeting.php. This program connects to the Oracle database using the OCI8 Extension Module, binds the input and output parameters, executes the call to the Greeting stored procedure, and returns the output parameter.

Example 5-36 Oracle PHP program oraGreeting.php

```
<?php
$conn = oci_connect("userid","password") or die;

$sql = "BEGIN Greeting(:name, :message); END;";

$stmt = oci_parse($conn,$sql);

// Bind the input parameter
oci_bind_by_name($stmt,":name",$name,32);

// Bind the output parameter
oci_bind_by_name($stmt,":message",$message,100);

// Assign a value to the input
$name = "Ariel";

oci_execute($stmt);

// $message is now populated with the output value
print "$message\n";
?>
```

The result of executing oraGreeting.php is shown in Figure 5-5.

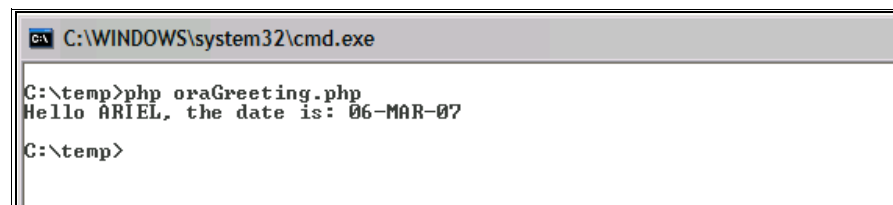


Figure 5-5 The result of executing oraGreeting.php

Connecting PHP applications to DB2

Two extensions, `ibm_db2` and `PDO_ODBC`, can be used to access DB2 databases from a PHP application. See “PHP extensions” on page 176 for details. For the DB2 conversion of the Oracle PHP program shown in Example 5-36 on page 219, the `ibm_db2` extension is used. Example 5-37 shows the source code for this converted program.

Example 5-37 DB2 PHP program db2Greeting.php

```
<?php
$database = 'sample';
$user = 'db2inst1';
$password = 'db2inst1';
// Next parameters used when making an uncataloged connection
// $hostname = 'localhost';
// $port = 50000;

$conn = db2_connect($database, $user, $password) or die;

// use this connection string for uncataloged connections:
// $conn_string = "DRIVER={IBM DB2 ODBC DRIVER};DATABASE=$database;
// HOSTNAME=$hostname;PORT=$port;PROTOCOL=TCPIP;UID=$user;PWD=$password";

// $conn = db2_connect($conn_string, '', '');

if ($conn) {
    //echo "Connection succeeded.<br />\n";

    $sql = 'CALL Greeting(?, ?)';
    $stmt = db2_prepare($conn, $sql);

    $name = 'Ariel';
    $message = '';

    db2_bind_param($stmt, 1, "name", DB2_PARAM_IN);
    db2_bind_param($stmt, 2, "message", DB2_PARAM_OUT);

    db2_execute($stmt);

    // $message is now populated with the output value
    print "$message\n";
}

?>
```

As can be seen in this example, there are few changes that need to be completed before the application may use DB2. These can be summarized by observing the differences between the following functions:

- ▶ `oci_connect` and `db2_connect`
- ▶ `oci_bind_by_name` and `db2_bind_param`
- ▶ `oci_parse` and `db2_prepare`

`oci_connect` and `db2_connect`

`oci_connect` takes the following required and optional (shown in []) parameters:

```
(string $username, string $password [, string $db [, string $charset [, int  
$session_mode]]] )
```

Note: Database (\$db) is an *optional* parameter. If the database is *not* specified, PHP uses the environments `ORACLE_SID` and `TWO_TASK` to determine the name of the local Oracle instance and the location of `tnsnames.ora`.

`oci_connect` will be converted to `ibm_db2` function `db2_connect`, which takes the following required and optional (shown in []) parameters:

```
(string database, string username, string password, [array options])
```

When connecting to DB2 through PHP, the connection may be made through either a cataloged or an uncataloged database.

Note: Refer to the DB2 manual *Administration Guide: Implementation*, SC10-4221, for detailed information regarding cataloging a DB2 database.

For an uncataloged connection to a database, the *database* parameter represents a complete connection string in the format shown in Example 5-38.

Example 5-38 Connection string for an uncataloged DB2 database

```
DRIVER={IBM DB2 ODBC DRIVER};DATABASE=database;HOSTNAME=hostname;  
PORT=port;PROTOCOL=TCPIP;UID=username;PWD=password
```

This is demonstrated by the code shown in Example 5-39.

Example 5-39 Connection string for uncataloged database used in the example

```
$database = 'sample';  
$user = 'db2inst1';  
$password = 'db2inst1';  
$hostname = 'localhost';  
$port = 50000;  
  
$conn_string = "DRIVER={IBM DB2 ODBC DRIVER};DATABASE=$database;  
HOSTNAME=$hostname;PORT=$port;PROTOCOL=TCPIP;UID=$user;PWD=$password;";  
  
$conn = db2_connect($conn_string, '', '');
```

oci_bind_by_name and db2_bind_param

oci_bind_by_name takes the following required and optional (shown in []) parameters:

```
( resource $statement, string $ph_name, mixed &$amp;variable [, int $maxlength [,  
int $type]] )
```

oci_bind_by_name will be converted to ibm_db2 function db2_bind_param, which accepts the following required and optional [] parameters:

```
(resource stmt, int parameter-number, string variable-name, [int  
parameter-type, [int data-type, [int precision, [int scale]]])
```

The parameter information is as follows:

- ▶ stmt: A prepared statement returned from db2_prepare().
- ▶ parameter-number: Specifies the 1-indexed position of the parameter in the prepared statement.
- ▶ variable-name: A string specifying the name of the PHP variable to bind to the parameter specified by parameter-number.

- ▶ **parameter-type:** A constant specifying whether the PHP variable should be bound to the SQL parameter as an input parameter (DB2_PARAM_IN), an output parameter (DB2_PARAM_OUT), or as a parameter that accepts input and returns output (DB2_PARAM_INOUT). This parameter is optional.
- ▶ **data-type:** A constant specifying the SQL data type that the PHP variable should be bound as, whether one of DB2_BINARY, DB2_CHAR, DB2_DOUBLE, or DB2_LONG. This parameter is optional.
- ▶ **precision:** Specifies the precision with which the variable should be bound to the database. This parameter is optional.
- ▶ **scale:** Specifies the scale with which the variable should be bound to the database. This parameter is optional.

With this information, binding the stored procedure input and output parameters is converted, as shown in Example 5-40.

Example 5-40 Converting oci_bind_by_name to db2_bind_param

ORACLE:

```
// Bind the input parameter
oci_bind_by_name($stmt, ':name', $name, 32);

// Bind the output parameter
oci_bind_by_name($stmt, ':message', $message, 100);
```

DB2 conversion:

```
// Bind the input parameter
db2_bind_param($stmt, 1, "name", DB2_PARAM_IN);

// Bind the output parameter
db2_bind_param($stmt, 2, "message", DB2_PARAM_OUT);
```

oci_parse and db2_prepare

The `oci_parse` function prepares a query. The function accepts the following parameters:

```
( resource $connection, string $query )
```

`oci_parse` will be converted to `db2_prepare`, which accepts the following required and optional (shown in []) parameters:

```
(resource connection, string statement, [array options])
```

The parameters are defined as follows:

- ▶ **connection:** A valid database connection resource variable as returned from `db2_connect()` or `db2_pconnect()`.
- ▶ **statement:** An SQL statement, optionally containing one or more parameter markers.
- ▶ **options:** [optional] An associative array containing statement options. You can use this parameter to request a scrollable cursor on database management systems that support this functionality.

Using this information, the calling of a DB2 stored procedure is converted, as shown in Example 5-41.

Example 5-41 Converting oci_parse to db2_prepare

Oracle:

```
$sql = 'BEGIN Greeting(:name, :message); END;';  
$stmt = oci_parse($conn,$sql);
```

DB2 conversion:

```
$sql = 'CALL Greeting(?, ?)';  
$stmt = db2_prepare($conn, $sql);
```

After the changes described above have been implemented, the application is fully converted to DB2. The result of executing this program is shown in Figure 5-6.

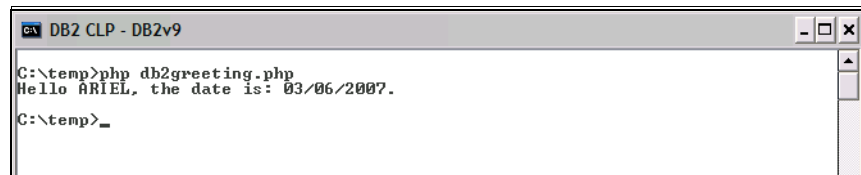


Figure 5-6 The result of executing db2Greeting.php

Note: For complete information regarding PHP, refer to *Developing Perl and PHP Applications*, SC10-4234.

5.10 Converting .NET applications

The supported operating systems for developing and deploying .NET Framework 1.1 applications are:

- ▶ Windows 2000
- ▶ Windows XP (32-bit edition)
- ▶ Windows Server 2003 (32-bit edition)

The supported operating systems for developing and deploying .NET Framework 2.0 applications are:

- ▶ Windows 2000, Service Pack 3
- ▶ Windows XP with Service Pack 2 (32-bit and 64-bit editions)
- ▶ Windows Server 2003 (32-bit and 64-bit editions)

Supported development software for .NET Framework applications

In addition to a DB2 client, you need one of the following options to develop .NET Framework applications:

- ▶ Visual Studio 2003 (for .NET Framework 1.1 applications)
- ▶ Visual Studio 2005 (for .NET Framework 2.0 applications)
- ▶ .NET Framework 1.1 Software Development Kit and .NET Framework Version 1.1 Redistributable Package (for .NET Framework 1.1 applications)
- ▶ .NET Framework 2.0 Software Development Kit and .NET Framework Version 2.0 Redistributable Package (for .NET Framework 2.0 applications))

.NET Data Providers

DB2 for Linux, AIX, and Windows includes three .NET Data Providers:

- ▶ DB2 .NET Data Provider

A high performance, managed ADO.NET Data Provider. This is the *recommended* .NET Data Provider for use with DB2 family databases. ADO.NET database access using the DB2 .NET Data Provider has fewer restrictions, and provides significantly better performance than the OLE DB and ODBC .NET bridge providers. This provides support for .NET 2.0, .NET3.0, and .NET 3.5 framework.

- ▶ OLE DB .NET Data Provider

A bridge provider that feeds ADO.NET requests to the IBM OLE DB provider (by way of the COM interop module). This .NET Data Provider is *not recommended* for access to DB2 family databases. The DB2 .NET Data Provider is faster and more feature-rich.

► ODBC .NET Data Provider

A bridge provider that feeds ADO.NET requests to the IBM ODBC driver. This .NET Data Provider is *not recommended* for access to DB2 family databases. The DB2 .NET Data Provider is faster and more feature-rich

In addition to the DB2 .NET Data Provider, IBM also provides a collection of add-ins to the Microsoft Visual Studio .NET IDE, providing tight integration between Optim Data Studio and DB2 for LUW, and the host database using DB2 Connect. The add-ins simplify the creation of DB2 applications that use the ADO.NET interface. The add-ins can also be used to develop server-side objects, such as SQL stored procedures and user-defined functions. The DB2 Visual Studio add-ins can be obtained at the following Web site:

<http://www-306.ibm.com/software/data/db2/windows/dotnet.html>

The IBM.Data.DB2 name space contains the DB2 .NET Data Provider. To use the DB2 .NET Data Provider, you must add the Imports or using statement for the IBM.Data.DB2 name space to your application .DLL, as shown in Example 5-42.

Example 5-42 Examples of the required Imports or using statement

```
[Visual Basic]
Imports IBM.Data.DB2
```

```
[C#]
using IBM.Data.DB2;
```

Also, references to IBM.Data.db2.dll and IBM.Data.DB2.Server.dll must be added to the project.

VB .NET conversion example

In general, converting a .NET application from Oracle to DB2 is quite simple. In most cases, it will entail *replacing* the classes that are available in the Oracle .NET Data Provider with functionally equivalent classes that are available in the DB2 .NET Data Provider, for example, OracleConnection with DB2Connection or OracleCommand with DB2Command, and so on.

In this section, we demonstrate this point using a simple VB .NET application that connects to a database, executes a SELECT, and returns a result set. This example is demonstrated in Oracle and then converted to DB2. In the DB2 example, the changes that are necessary for converting from Oracle to DB2 are outlined.

Components of the GUI for the conversion example

The GUI, used for both examples, consists of several CONTROLS:

- ▶ A RUN QUERY button: When this button is clicked, the code in the Click event will:
 - Connect to the database.
 - Execute the query.
- ▶ A Query Results Text box (for aesthetic purposes only).
- ▶ A List Box: Once the query has been executed, the results are displayed in this list box.
- ▶ A Quit button: Clicking this button ends the application.

Figure 5-7 shows the GUI used to demonstrate the VB .NET application conversion example.

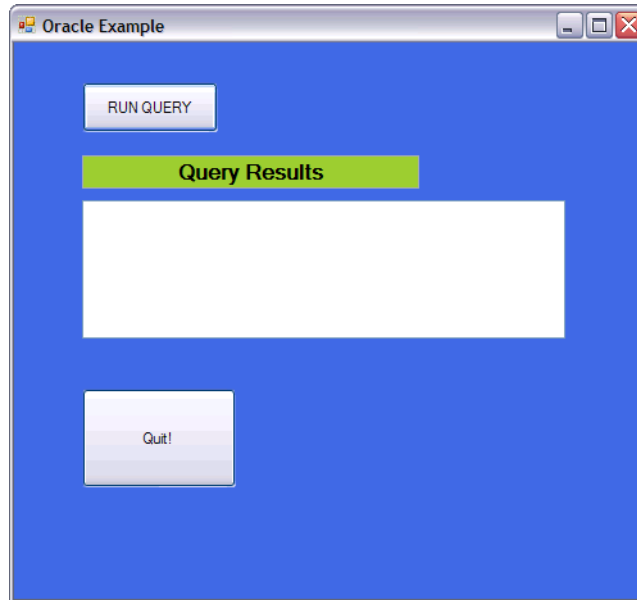


Figure 5-7 GUI for the VB .NET application conversion example

The essential components of this application are contained within the Click Event for the RUN QUERY control button. Example 5-43 shows the code in the Button1_Click event as it might appear in an Oracle application. Some explanations of the changes are documented in the notes that appear after the code example.

Example 5-43 The Button1_Click event

```
Imports Oracle.DataAccess.Client ' ODP.NET Oracle managed provider [1]
```

```
Public Class Form1
```

```
    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As  
System.EventArgs) Handles Button1.Click
```

```
    Dim oradb As String = "Data Source=(DESCRIPTION=(ADDRESS_LIST=" _ +  
" (ADDRESS=(PROTOCOL=TCP) (HOST=9.10.11.12) (PORT=1521)))" _ +  
" (CONNECT_DATA=(SERVER=DEDICATED) (SERVICE_NAME=ora10g)))"; _ + "User  
Id=ora_usr;Password=ora_usr;" [2]
```

```
    Dim conn As New OracleConnection(oradb) [3]  
    conn.Open()
```

```
    Dim cmd As New OracleCommand [4]  
    cmd.Connection = conn  
    cmd.CommandText = "select first_name, last_name from employees  
                        where dept_code = 'IT'"
```

```
    cmd.CommandType = CommandType.Text
```

```
    Dim dr As OracleDataReader = cmd.ExecuteReader() [5]
```

```
        While dr.Read()  
            ListBox1.Items.Add("The name of this employee is: " +  
dr.Item("first_name") + dr.Item("last_name")) [6]  
        End While
```

```
        conn.Dispose()
```

```
    End Sub
```

Notes:

1. IMPORT Oracle.DataAccess.Client is added to the application .DLL.
2. A String (OraDb) is declared as the connection string for the Oracle database.
3. A connection (conn) is defined as an OracleConnection.
4. A command (cmd) is defined as an OracleCommand and populated with the text of the query.
5. A DataReader (dr) is defined as an OracleDataReader and the query is executed.
6. The List Box is populated with the results of the query.

When the application executes, clicking **RUN QUERY** yields the results that are shown in Figure 5-8.

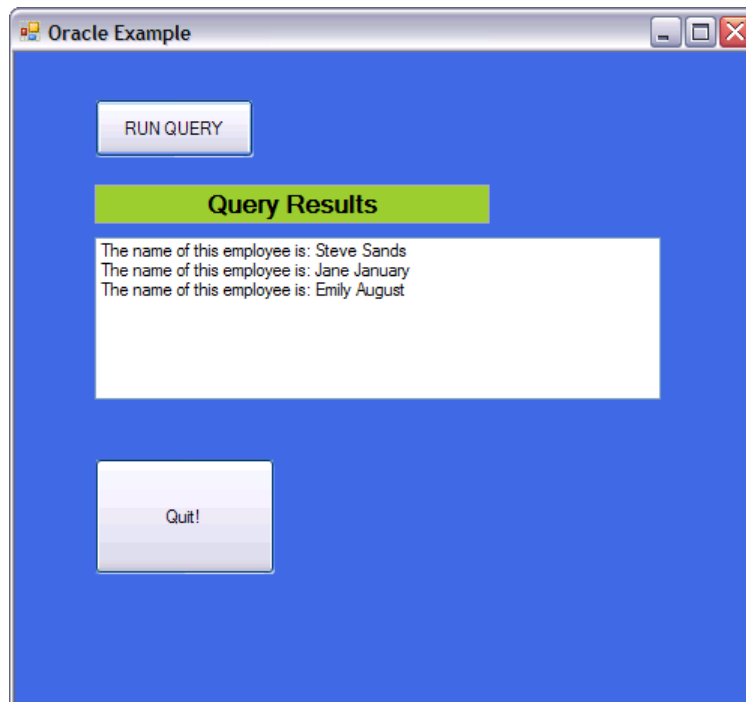


Figure 5-8 The results of the Oracle Example are displayed in the List Box

DB2 Example (conversion)

The GUI for the DB2 conversion is shown in Figure 5-7 on page 227.

Because the essential components of this application are contained within the Click Event for the RUN QUERY control button, the focus of the conversion centers on this control. Example 5-44 shows the code in the Button1_Click event as it will appear after conversion to DB2. Some explanations of the changes are documented in the notes that appear after the code example.

Example 5-44 The code in the Button1_Click event after conversion

```
Imports IBM.Data.DB2 [1]

Public Class Form1

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button1.Click
Dim db2db As String = [2]
"Server=localhost:50000;Database=testdb;UID=db2inst1;PWD=db2inst1"

Dim conn As New DB2Connection(db2db) [3]

conn.Open()

Dim cmd As New DB2Command [4]
cmd.Connection = conn

cmd.CommandText = "select first_name, last_name from employees where dept_code
= 'IT'"

cmd.CommandType = CommandType.Text

Dim dr As DB2DataReader = cmd.ExecuteReader() [5]
    While dr.Read()

        ListBox1.Items.Add("The name of this employee is: " +
dr.Item("first_name") + dr.Item("last_name"))

    End While [6]

conn.Dispose()
End Sub
```

Notes:

1. To use the DB2 .NET Data Provider, you must add the Imports (VB) or using (C#) statement for the IBM.Data.DB2 name space to your application .DLL.
2. A string (db2db) is declared and populated as the connection string for the DB2 database (converted from OraDb).
3. A connection (conn) is defined as DB2Connection (converted from OracleConnection).
4. A command (cmd) is defined as DB2Command (converted from OracleCommand).
5. A DataReader (dr) is declared as a DB2DataReader (converted from OracleDataReader).
6. The List Box is populated with the results of the query.

Once the changes are effected, clicking **RUN QUERY** yields the results shown in Figure 5-9.

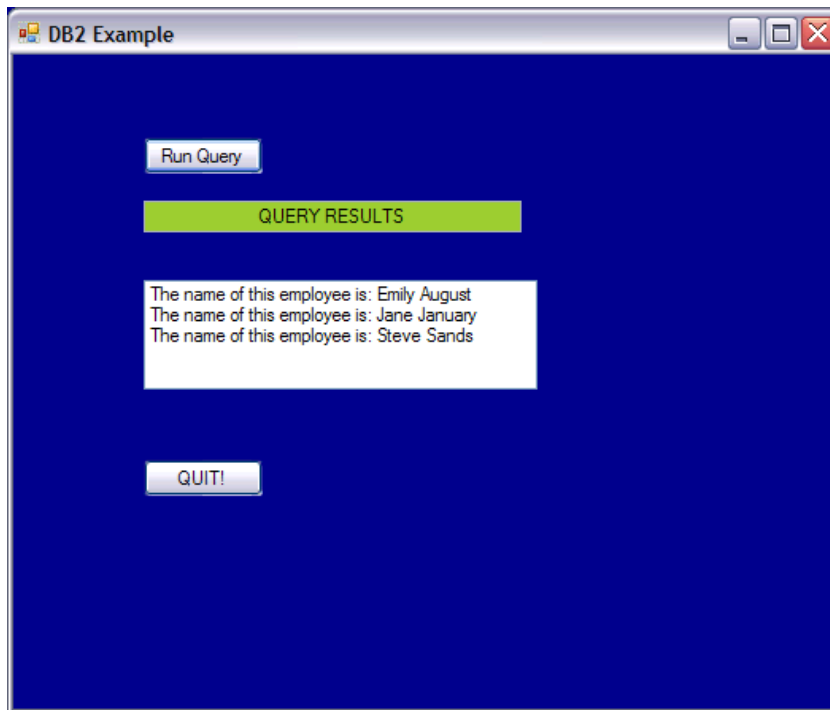


Figure 5-9 The results of executing the DB2 Example application

Note: For complete information about the DB2 .NET provider, consult the information at the following URL:

<http://publib.boulder.ibm.com/infocenter/db2luw/v8/index.jsp?topic=/com.ibm.db2.udb.dndp.doc/htm/fr1rfIBMDDataDB2.htm>

For in-depth information about .NET programming, refer to *Developing ADO.NET and OLE DB Applications*, SC10-4230.



Terminology mapping

This appendix contains the terminology mapping of Oracle to DB2, as shown in Table A-1.

Table A-1 Oracle terminology to DB2 mapping

Oracle	DB2	Comments
Oracle EE	DB2 Enterprise 9	Enterprise product.
Oracle Parallel	DB2 Enterprise DPF	Support server partitioning.
Oracle Gateway	DB2 Connect	DRDA® access to hosts.
PL/SQL	SQL Procedural Language	Programming language extension to SQL. DB2 stored procedures can be programmed in SQL Control Statements (subset of PSM standard), Java, C, C++, COBOL, FORTRAN, OLE, and REXX. DB2 functions can be programmed in Java, C, C++, OLE, or SQL control statements.
SQL*PLUS	CLPPlus	Command-line interface to the server.

Oracle	DB2	Comments
Instance	Instance	Processes and shared memory. In DB2, it also includes a permanent directory structure: an instance is usually created at install time (or can be later) and must exist before a database can be created. A DB2 instance is also known as the <i>database manager</i> (DBM). A DB2 instance can have multiple databases. But an Oracle instance can only have one database.
Database	Database	Physical structure containing data. In Oracle, multiple instances can use the same database, and an instance can connect to one and only one database. In DB2, multiple databases can be created and used concurrently in the same instance.
Control files and .ora files	DBM and database configuration files and so on	In Oracle, these are files that name the locations of files making up the database and provide configuration values. In DB2, each instance (DBM) and database has its own set of configuration parameters stored in a binary file. There are also other internal files and directories; none are manually edited.
Database Link	Federated System	In Oracle, this is an object that describes a path from one database to another. In DB2, a federated system is used. One database is chosen as the federated database and within it wrappers, servers, nicknames, and other optional objects are created to define how to access the other databases (including Oracle databases) and objects in them. Once an application is connected to the federated database, it can access all authorized objects in the federated system.
Table spaces	Table spaces	Contains actual database data.
Data files	Containers	Entities inside the table spaces.
Segments	Objects	Entities inside the containers/data files.
Extents	Extents	Entities inside the objects/segments.
Data blocks	Pages	Smallest storage entity in the storage model.

Oracle	DB2	Comments
Clusters	N/A	A data structure that allows related data to be stored together on disk. This data can be table or hash clusters. The closest facility to this in DB2 is a <i>clustering index</i> , which causes rows inserted into a table to be placed physically close to the rows for which the key values of this index are in the same range.
Data Dictionary	System Catalog	Meta data of the database.
N/A	SMS	System-managed table space.
Data files	DMS containers	The file and raw devices under database-managed table space.
Data cache	Buffer pools	Buffers data in the table spaces to reduce disk I/O.
Statement cache	Package cache	Caches prepared dynamic SQL statements.
Redo logs	Log files	Recovery logs.
Rollback segments	N/A	Store the old version of data for a mutating table. In DB2, the old version of an updated row is stored in the log file along with the new version.
SGA	Database manager and database shared memory	Shared memory area(s) for the database server. In Oracle, there is one, while in DB2 there is one at the database manager (instance) level and one for each active database.
UGA	Agent / application shared memory	Shared memory area to store user-specific data passed between application process and the database server.
N/A	Package	A precompiled access plan for an embedded static SQL application stored in the server.
Package	Module	A logical grouping of PL/SQL blocks that can be invoked by other PL/SQL applications.



Data types

This appendix explains data types in different environments:

- ▶ Supported SQL data types in C/C++
- ▶ Supported SQL data types in Java
- ▶ Data types available in PL/SQL
- ▶ Mapping Oracle data types to DB2 data types

B.1 Supported SQL data types in C/C++

Table B-1 provides a complete list of SQL data types, C and C/C++ data type mapping, and a brief description of each.

For more information about mapping between SQL data types and C and C++ data types, refer to the following resources:

- *Developing Embedded SQL Applications*, SC27-2445
- DB2 Information Center
 - Supported SQL data types in C and C++ embedded SQL applications, found at:
<http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/topic/com.ibm.db2.1uw.apdv.embed.doc/doc/r0006090.html>
 - Data types for procedures, functions, and methods in C and C++ embedded SQL applications, found at:
<http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/topic/com.ibm.db2.1uw.apdv.embed.doc/doc/r0006094.html>

Table B-1 Oracle to DB2 data type mapping

Item	SQL data type sqltype	C/C++ type	sqlen	Description
integer	SMALLINT (500 or 501)	short short int sqlint 16	2	<ul style="list-style-type: none"> ► 16-bit signed integer. ► Range between (-32,768 and 32,767). ► Precision of 5 digits.
	INTEGER INT (496 or 497)	long long int sqlint32	4	<ul style="list-style-type: none"> ► 32-bit signed integer. ► Range between (-2,147,483,648 and 2,147,483,647). ► Precision of 10 digits.
	BIGINT (492 or 493)	long long long __int64 sqlint64	8	64-bit signed integer.
floating point	REAL FLOAT (480 or 481)	float		<ul style="list-style-type: none"> ► Single precision floating point. ► 32-bit approximation of a real number. ► FLOAT(n) can be synonym for REAL if $0 < n < 25$.
	DOUBLE (480 or 481) DOUBLE PRECISION	double	8	<ul style="list-style-type: none"> ► Double precision floating point. ► 64-bit approximation of a real number. ► Range between (0, -1.79769E+308 to -2.225E-307, 2.225E-307 to 1.79769E+308). ► FLOAT(n) can be synonym for DOUBLE if $24 < n < 54$.

Item	SQL data type sqltype	C/C++ type	sqlen	Description
decimal	DECIMAL(p,s) DEC(p,s) (484 or 485) NUMERIC(p,s) NUM(p,s)	double / decimal	p/2+1	<ul style="list-style-type: none"> ▶ Packed decimal. ▶ If precision /scale not specified, the default is (5,0). ▶ The max precision is 31 digits, and the max range is between (-10E31+1 ... 10E31 -1). ▶ Consider using char / decimal functions to manipulate packed decimal fields as char data.
date and time	DATE (384 or 385)	struct { short len; char data[10]; } dt; char dt[11];	10	<ul style="list-style-type: none"> ▶ Null-terminated character form (11 characters) or varchar struct form (10 characters). ▶ struct can be divided as desired to obtain the individual fields. ▶ Example: 11/02/2000. ▶ Stored internally as a packed string of 4 bytes.
	TIME (388 or 389)	char	8	<ul style="list-style-type: none"> ▶ Null-terminated character form (9 characters) or varchar struct form (8 characters). ▶ struct can be divided as desired to obtain the individual fields. ▶ Example: 19:21:39. ▶ Stored internally as a packed string of 3 bytes.
	TIMESTAMP (392 or 393)	char	26	<ul style="list-style-type: none"> ▶ Null-terminated character form or varchar struct form. ▶ Allows 19-32 characters. ▶ struct can be divided as desired to obtain the individual fields. ▶ Example: 2003-08-04-01.02.03.000000. ▶ Stored internally as a packed string of 10 bytes.

Item	SQL data type sqltype	C/C++ type	sqlen	Description
character	CHAR(1) (452 or 453)	char	1	Single character.
	CHAR(n) (452 or 453)	char	n	<ul style="list-style-type: none"> ► Fixed-length character string consisting of n bytes. ► Use char[n+1] where 1 <= n <= 254. ► If length not specified, defaults to 1.
	VARCHAR (460 or 461)	char	n	<ul style="list-style-type: none"> ► Null-terminated variable length character string. ► Use char[n+1] where 1 <= n <= 32672.
	VARCHAR (448 or 449) or VARCHAR2 (448 or 449)	struct tag { short int; char[n] }	len	<ul style="list-style-type: none"> ► Non null-terminated varying character string with 2-byte string length indicator. ► Use char[n] in struct form where 1 <= n <= 32672. ► Default SQL type.
	LONG VARCHAR (456 or 457)	struct tag { short int; char[n] }	len	<ul style="list-style-type: none"> ► Non null-terminated varying character string with 2-byte string length indicator. ► Use char[n] in struct form where 32673 <= n <= 32700.
	CLOB(n) (408 or 409)	clob	n	<ul style="list-style-type: none"> ► Non null-terminated varying character string with 4-byte string length indicator. ► Use char[n] in struct form where 1 <= n <= 2147483647.
	CLOB (964 or 965)	clob_locator		► Identifies CLOB entities residing on the server.
	CLOB (920 or 921)	clob_file		► Descriptor for file containing CLOB data.
binary	BLOB(n) (404 or 405)	blob	n	<ul style="list-style-type: none"> ► Non null-terminated varying binary string with 4-byte string length indicator. ► Use char[n] in struct form where 1 <= n <= 2147483647.
	BLOB (960 or 961)	blob_locator		Identifies BLOB entities on the server
	BLOB (916 or 917)	blob_file		Descriptor for the file containing BLOB data.

Item	SQL data type sqltype	C/C++ type	sqlen	Description
double-byte	GRAPHIC(1) GRAPHIC(n) (468 or 469)	sqldbchar	24	<ul style="list-style-type: none"> ▶ sqldbchar is a single double-byte character string. ▶ For a fixed-length graphic string of length integer which may range from 1 to 127. If the length specification is omitted, a length of 1 is assumed. ▶ Precompiled with WCHARTYPE NOCONVERT option.
	VARGRAPHIC(n) (464 or 465)	struct { short int; sqldbchar[n] } tag; alternately: sqldbchar[n+1]	n*2+4	<ul style="list-style-type: none"> ▶ For a varying-length graphic string of maximum length integer, which may range from 1 to 16336. ▶ Precompiled with WCHARTYPE NOCONVERT option. ▶ Null terminated variable-length.
	LONG VARGRAPHIC(n) (472 or 473)	struct { short int; sqldbchar[n] } tag;		<ul style="list-style-type: none"> ▶ For a varying-length graphic string with a maximum length of 16350 and a 2-byte string length indicator 16337<=n <=16350. ▶ Precompiled with WCHARTYPE NOCONVERT option.
	DBCLOB(n) (412 or 413)	dbclob		<ul style="list-style-type: none"> ▶ For non-null-terminated varying double-byte character large object maximum length in double-byte characters. ▶ 4 bytes string length indicator. ▶ Use dbclob(n) where 1<=n <= 1073741823 double-byte characters. ▶ Precompiled with WCHARTYPE NOCONVERT option.
	DBCLOB	dbclob_locator		<ul style="list-style-type: none"> ▶ Identifies DBCLOB entities residing on the server. ▶ Precompiled with WCHARTYPE NOCONVERT option.
	DBCLOB	dbclob_file		<ul style="list-style-type: none"> ▶ Descriptor for file containing DBCLOB data. ▶ Precompiled with WCHARTYPE NOCONVERT option.
external data	Datalink(n)		n+54	The length of a DATALINK column is 200 bytes.
	XML (988 or 989)	struct { sqluint32 length; char data[n]; }		<ul style="list-style-type: none"> ▶ XML value. ▶ 1<=n<=2 147 483 647.

B.2 Supported SQL data types in Java

Table B-2 shows the Java equivalent of each SQL data type, based on the JDBC specification for data type mappings. The JDBC driver converts the data exchanged between the application and the database using the following mapping schema. Use these mappings in your Java applications and your PARAMETER STYLE JAVA procedures and UDFs.

For more information about mapping between SQL data types and Java data types, refer to *Developing Embedded SQL Applications*, SC27-2445 or the DB2 Information Center, found at:

<http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/topic/com.ibm.db2.luw.apdv.java.doc/doc/rjvjdata.html>

Table B-2 SQL data types mapped to Java declarations

Item	SQL data type sqltype	Java type	sqllen	Description
integer	SMALLINT (500 or 501)	short	2	16-bit, signed integer
	INTEGER (496 or 497)	int	4	32-bit, signed integer
	BIGINT ¹ (492 or 493)	long	8	64-bit, signed integer
floating point	REAL (480 or 481)	float		Single precision floating point
	DOUBLE (480 or 481)	double	4	Single precision floating point
	DOUBLE (480 or 481)	double	8	Double precision floating point
decimal	DECIMAL(p,s) (484 or 485)	java.math. BigDecimal	n/2	Packed decimal
	DECFLOAT(n)	java.math. BigDecimal		n=16 or n=34
date and time	DATE (384 or 385)	java.sql.Date	10	10 byte character string
	TIME (388 or 389)	java.sql.Time	8	8 byte character string
	TIMESTAMP (392 or 393)	java.sql.Timestamp	26	26 byte character string
	TIMESTAMP(n)	java.sql.Timestamp	26	0<=p<=12, default 6

Item	SQL data type sqltype	Java type	sqllen	Description
character	CHAR(n) (452 or 453)	java.lang.String	n	Fixed-length character string of length n where n is from 1 to 254
	CHAR (n) FOR BIT DATA	byte[]		Fixed-length character string of length n where n is from 1 to 254
	VARCHAR (n) (448 or 449)	java.lang.String	n	Variable-length character string, n <= 32672
	VARCHAR (n) FOR BIT DATA	byte[]		Variable-length character string n <= 32672
	LONG VARCHAR (456 or 457)	java.lang.String	n	Long variable-length character string, n <= 32672
	CLOB(n) (408 or 409)	java.lang.Clob, java.lang.String, java.io.ByteArrayInputStream, java.io.StringReader	n	Large object variable-length character string
binary	BLOB(n) (404 or 405)	java.lang.Blob, byte[]	n	Large object variable-length binary string
	BINARY(n)	byte[]		n<=254.
	VARBINARY(n)	byte[]		n<=32672
double- byte	GRAPHIC(n) (468 or 469)	java.lang.String	n	Fixed-length double-byte character string, n<=127.
	VARGRAPHIC(n) (464 or 465)	java.lang.String	n*2+4	Non-null-terminated varying double-byte character string with 2 byte string length indicator, n<=16336
	LONG VARGRAPHIC(n) (472 or 473)	java.lang.String	n	Non-null-terminated varying double-byte character string with 2 byte string length indicator
	DBCLOB(n) (412 or 413)	java.lang.Clob	n	Large object variable-length double-byte character string
	CLOB(n) (408 or 409)	java.sql.Clob, java.lang.String	n	Non null-terminated varying character string with 4 byte string length indicator

Item	SQL data type sqltype	Java type	sqllen	Description
binary	BLOB(n) (404 or 405)	java.sql.Blob, byte[],	n	Non null-terminated varying binary string with 4 byte string length indicator
	ROWID	java.sql.RowId, byte[], com.ibm.db2.jcc.DB2RowID ^a		RowId identifier
	XML	java.lang.String, com.ibm.db2.jcc.DB2Xml ^a , java.sql.SQLXML, java.io.InputStream, java.sql.Clob, java.sql.Blob, byte[]		XML value

a. Deprecated

B.3 Data types available in PL/SQL

The DB2 data server supports a wide range of data types that can be used to declare constants and variables in a PL/SQL block. Table B-3 presents the supported scalar data types that are available in PL/SQL.

Table B-3 Supported scalar data types that are available in PL/SQL

PL/SQL data type	DB2 SQL data type	Description
BINARY_INTEGER	INTEGER	Integer numeric data
BLOB	BLOB(4096)	Binary data
BLOB (n)	BLOB (n) n = 1 to 2 147 483 647	Binary large object data
BOOLEAN	BOOLEAN	Logical Boolean (true or false)
CHAR	CHAR (1)	Fixed-length character string data of length 1
CHAR (n)	CHAR (n) n = 1 to 254	Fixed-length character string data of length n
CHAR VARYING (n)	VARCHAR (n)	Variable-length character string data of maximum length n
CHARACTER	CHARACTER (1)	Fixed-length character string data of length 1
CHARACTER (n)	CHARACTER (n) n = 1 to 254	Fixed-length character string data of length n
CHARACTER VARYING (n)	VARCHAR (n) n = 1 to 32 672	Variable-length character string data of maximum length n

PL/SQL data type	DB2 SQL data type	Description
CLOB	CLOB (1 MB)	Character large object data
CLOB (<i>n</i>)	CLOB (<i>n</i>) <i>n</i> = 1 to 2 147 483 647	Fixed-length long character string data of length <i>n</i>
DATE	DATE ^a	Date and time data (expressed to the second)
DEC	DEC (9, 2)	Decimal numeric data
DEC (<i>p</i>)	DEC (<i>p</i>) <i>p</i> = 1 to 31	Decimal numeric data of precision <i>p</i>
DEC (<i>p</i> , <i>s</i>)	DEC (<i>p</i> , <i>s</i>) <i>p</i> = 1 to 31; <i>s</i> = 1 to 31	Decimal numeric data of precision <i>p</i> and scale <i>s</i>
DECIMAL	DECIMAL (9, 2)	Decimal numeric data
DECIMAL (<i>p</i>)	DECIMAL (<i>p</i>) <i>p</i> = 1 to 31	Decimal numeric data of precision <i>p</i>
DECIMAL (<i>p</i> , <i>s</i>)	DECIMAL (<i>p</i> , <i>s</i>) <i>p</i> = 1 to 31; <i>s</i> = 1 to 31	Decimal numeric data of precision <i>p</i> and scale <i>s</i>
DOUBLE	DOUBLE	Double precision floating-point number
DOUBLE PRECISION	DOUBLE PRECISION	Double precision floating-point number
FLOAT	FLOAT	Float numeric data
FLOAT (<i>n</i>) <i>n</i> = 1 to 24	REAL	Real numeric data
FLOAT (<i>n</i>) <i>n</i> = 25 to 53	DOUBLE	Double numeric data
INT	INT	Signed 4 byte integer numeric data
INTEGER	INTEGER	Signed 4 byte integer numeric data
LONG	CLOB (32760)	Character large object data
LONG RAW	BLOB (32760)	Binary large object data
LONG VARCHAR	CLOB (32760)	Character large object data
NATURAL	INTEGER	Signed 4 byte integer numeric data
NCHAR	GRAPHIC (127)	Fixed-length graphic string data
NCHAR (<i>n</i>) <i>n</i> = 1 to 2000	GRAPHIC (<i>n</i>) <i>n</i> = 1 to 127	Fixed-length graphic string data of length <i>n</i>
NCLOB ^b 2	DBCLOB (1 MB)	Double-byte character large object data
NCLOB (<i>n</i>)	DBCLOB (2000)	Double-byte long character string data of maximum length <i>n</i>
NVARCHAR2	VARGRAPHIC (2048)	Variable-length graphic string data
NVARCHAR2 (<i>n</i>)	VARGRAPHIC (<i>n</i>)	Variable-length graphic string data of maximum length <i>n</i>

PL/SQL data type	DB2 SQL data type	Description
NUMBER	NUMBER ^c	Exact numeric data
NUMBER (<i>p</i>)	NUMBER (<i>p</i>) ^c	Exact numeric data of maximum precision <i>p</i>
NUMBER (<i>p</i> , <i>s</i>)	NUMBER (<i>p</i> , <i>s</i>) ^c <i>p</i> = 1 to 31	Exact numeric data of maximum precision <i>p</i> and scale <i>s</i>
NUMERIC	NUMERIC (9.2)	Exact numeric data
NUMERIC (<i>p</i>)	NUMERIC (<i>p</i>) <i>p</i> = 1 to 31	Exact numeric data of maximum precision <i>p</i>
NUMERIC (<i>p</i> , <i>s</i>)	NUMERIC (<i>p</i> , <i>s</i>) <i>p</i> = 1 to 31; <i>s</i> = 0 to 31	Exact numeric data of maximum precision <i>p</i> and scale <i>s</i>
PLS_INTEGER	INTEGER	Integer numeric data
RAW	BLOB (32767)	Binary large object data
RAW (<i>n</i>)	BLOB (<i>n</i>) <i>n</i> = 1 to 32 767	Binary large object data
SMALLINT	SMALLINT	Signed two-byte integer data
TIMESTAMP (0)	TIMESTAMP (0)	Date data with timestamp information
TIMESTAMP (<i>p</i>)	TIMESTAMP (<i>p</i>)	Date and time data with optional fractional seconds and precision <i>p</i>
VARCHAR	VARCHAR (4096)	Variable-length character string data with a maximum length of 4096 characters
VARCHAR (<i>n</i>)	VARCHAR (<i>n</i>)	Variable-length character string data with a maximum length of <i>n</i> characters
VARCHAR2 (<i>n</i>)	VARCHAR2 (<i>n</i>) ^d	Variable-length character string data with a maximum length of <i>n</i> characters

a. When the DB2_COMPATIBILITY_VECTOR registry variable is set for the DATE data type, DATE is equivalent to TIMESTAMP (0).

b. For restrictions for the NCLOB data type in certain database environments, see “Restrictions on PL/SQL support” in the DB2 Information Center.

c. This data type is supported when the number_compat database configuration parameter is set to ON.

d. This data type is supported when the varchar2_compat database configuration parameter is set to ON.

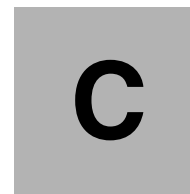
B.4 Mapping Oracle data types to DB2 data types

Table B-4 on page 247 summarizes the mapping from Oracle data types to the corresponding DB2 data types. In some cases, the mapping is one to many and depends on the actual usage of the data.

Table B-4 Mapping Oracle data types to DB2 data types

Oracle data type	DB2 data type	Notes
CHAR(n)	CHAR(n)	1 <= n <= 254.
VARCHAR2(n)	VARCHAR2(n)	n <= 32762.
NCHAR(n)	CHAR(n) ^a	1 <= n <= 254.
NVARCHAR2(n)	VARCHAR2(n) ^a	n <= 32762.
LONG	LONG VARCHAR(n)	if n <= 32700 bytes.
LONG	CLOB(2 GB)	if n <= 2 GB.
NUMBER(p)	NUMBER(p)	
NUMBER(p,s)	NUMBER(p,s)	if s > 0.
NUMBER	NUMBER	
RAW(n)	CHAR(n) FOR BIT DATA, VARCHAR(n) FOR BIT DATA , BLOB(n)	CHAR, if n <= 254 VARCHAR, if 254 < n <= 32672 BLOB, if 32672 < n <= 2 GB.
LONG RAW	LONG VARCHAR(n) FOR BIT DATA BLOB(n)	LONG, if n <= 32700 BLOB, if 32700 < n <= 2 GB.
BLOB	BLOB(n)	if n <= 2 GB.
CLOB	CLOB(n)	if n <= 2 GB.
NCLOB	DBCLOB(n)	if n <= 2 GB, use DBCLOB(n/2).
DATE	DATE	Oracle default format is DD-MON-YY.
DATE (only the date)	DATE (MM/DD/YYYY)	Use Oracle TO_CHAR() function to extract data for a subsequent DB2 load.
DATE (only the time)	TIME (HH24:MI:SS)	Use Oracle TO_CHAR() function to extract for a subsequent DB2 load.
TIMESTAMP (0)	TIMESTAMP (0)	Date data with timestamp information.
TIMESTAMP (p)	TIMESTAMP (p)	Date and time data with optional fractional seconds and precision p.
XMLType	XML	XML storage.

a. You can map Oracle NCHAR and NVARCHAR2 columns to DB2 CHAR and VARCHAR2 columns created in an Unicode database or column created with CCSID clause to an Unicode compatible code set.



Built-in packages

This appendix provides details for the built-in packages supported in DB2 9.7, which are:

- DBMS_OUTPUT
- DBMS_ALERT
- DBMS_PIPE
- DBMS_JOB
- DBMS_LOB
- DBMS_SQL
- DBMS_UTILITY
- UTL_FILE
- UTL_DIR
- UTL_MAIL
- UTL_SMTP

C.1 DBMS_OUTPUT

The DBMS_OUTPUT package provides a set of procedures that allow you to work with the message buffer by putting lines of text (messages) in a message buffer and getting messages from the message buffer. These procedures are useful during application debugging when you need to write messages to standard output. The command SET SERVEROUTPUT ON could be used to redirect the output to standard output.

Table C-1 lists the system-defined routines included in the DBMS_OUTPUT module.

Table C-1 System-defined routines available in the DBMS_OUTPUT module

Routine name	Description
DISABLE procedure	Disables the message buffer.
ENABLE procedure	Enables the message buffer.
GET_LINE procedure	Gets a line of text from the message buffer.
GET_LINES procedure	Gets one or more lines of text from the message buffer and places the text into a collection.
NEW_LINE procedure	Puts an end-of-line character sequence in the message buffer.
PUT procedure	Puts a string that includes no end-of-line character sequence in the message buffer.
PUT_LINE procedure	Puts a single line that includes an end-of-line character sequence in the message buffer.

Example C-1 shows an example of DBMS_OUTPUT and the output of PUT and PUT_LINE procedures.

Example C-1 Anonymous block with DBMS_OUTPUT procedures

```
SET SERVEROUTPUT ON
/

DECLARE
    v_message VARCHAR2(50);
BEGIN
    DBMS_OUTPUT.PUT(CHR(10));
    DBMS_OUTPUT.PUT_LINE('This is the beginning');
    DBMS_OUTPUT.PUT(CHR(10));

    v_message := 'You're seeing now the second line.';
    DBMS_OUTPUT.PUT_LINE(v_message);
```

```
END;  
/  
DB20000I The SQL command completed successfully.
```

This is the beginning

You're seeing now the second line.

C.2 DBMS_ALERT

The DBMS_ALERT package provides a set of procedures for registering, sending, and receiving alerts for a specific event. Alerts are stored in SYSTOOLS.DBMS_ALERT_INFO, which is created in the SYSTOOLSPACE when you first reference this package for each database. The DBMS_ALERT package requires that the database configuration parameter CUR_COMMIT is set to ON.

Table C-2 lists the system-defined routines included in the DBMS_ALERT module.

Table C-2 System-defined routines available in the DBMS_ALERT module

Routine name	Description
REGISTER procedure	Registers the current session to receive a specified alert.
REMOVE procedure	Removes registration for a specified alert.
REMOVEALL procedure	Removes registration for all alerts.
SIGNAL procedure	Signals the occurrence of a specified alert.
SET_DEFAULTS procedure	Sets the polling interval for the WAITONE and WAITANY procedures.
WAITANY procedure	Waits for any registered alert to occur.
WAITONE procedure	Waits for a specified alert to occur.

Example C-2 shows the use of some ALERT related routines. Assume the “AlertFromTrigger” alert is signalled as a result of an insert, which fires a trigger defined by TRIGGER trig1.

Example C-2 Signalling from Insert Trigger

```
CREATE OR REPLACE TRIGGER TRIG1
AFTER INSERT ON T1alert
FOR EACH ROW
BEGIN
    DBMS_ALERT.SIGNAL( 'alertfromtrigger', :NEW.C1 );
END;
/
```

You can catch the alert, by way of the WAITONE routine, after having registered the alert name, with a 60 second timeout, as shown in Example C-3.

Example C-3 Intercepting the alert

```
DECLARE
v_stat1 INTEGER;
v_msg1 VARCHAR2(50);
BEGIN
    DBMS_ALERT.REGISTER('alertfromtrigger');
    DBMS_ALERT.WAITONE('alertfromtrigger', v_msg1, v_stat1, 60);
END;
/
```

For more information and examples about DBMS_ALERT, visit the DB2 Information Center, found at:

<http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/topic/com.ibm.db2.luw.sql.rtn.doc/doc/r0053671.html>

C.3 DBMS_PIPE

The DBMS_PIPE package provides a set of routines for sending messages through a pipe within or between sessions that are connected to the same database.

Pipes are created either implicitly or explicitly during procedure calls. An implicit pipe is created when a procedure call contains a reference to a pipe name that does not exist. For example, if a pipe named “mailbox” is passed to the SEND_MESSAGE function and that pipe does not already exist, a new pipe named “mailbox” is created. An explicit pipe is created by calling the CREATE_PIPE function and specifying the name of the pipe.

Pipes can be private or public. A private pipe can only be accessed by the user who created the pipe. Even an administrator cannot access a private pipe that was created by another user. A public pipe can be accessed by any user who has access to the DBMS_PIPE package. To specify the access level for a pipe, use the CREATE_PIPE function and specify a value for the private parameter:

- ▶ “false” specifies that the pipe is public.
- ▶ “true” specifies that the pipe is private.

If no value is specified, the default is to create a private pipe. All implicit pipes are private.

Table C-3 lists the system-defined routines included in the DBMS_PIPE module.

Table C-3 System-defined routines available in the DBMS_PIPE module

Routine name	Description
CREATE_PIPE function	Explicitly creates a private or public pipe.
NEXT_ITEM_TYPE function	Determines the data type of the next item in a received message.
PACK_MESSAGE function	Puts an item in the session's local message buffer.
PACK_MESSAGE_RAW procedure	Puts an item of type RAW in the session's local message buffer.
PURGE procedure	Removes unreceived messages in the specified pipe.
RECEIVE_MESSAGE function	Gets a message from the specified pipe.
REMOVE_PIPE function	Deletes an explicitly created pipe.
RESET_BUFFER procedure	Resets the local message buffer.
SEND_MESSAGE procedure	Sends a message on the specified pipe.
UNIQUE_SESSION_NAME function	Returns a unique session name.
UNPACK_MESSAGE procedures	Retrieves the next data item from a message and assigns it to a variable.

To send a message through a pipe, call the `PACK_MESSAGE` function to put individual data items (lines) in a local message buffer that is unique to the current session. Then, run the `SEND_MESSAGE` function to send the message through the pipe.

To receive a message, call the `RECEIVE_MESSAGE` function to get a message from the specified pipe. The message is written to the receiving session's local message buffer. Then, call the `UNPACK_MESSAGE` procedure to retrieve the next data item from the local message buffer and assign it to a specified program variable. If a pipe contains multiple messages, the `RECEIVE_MESSAGE` function gets the messages in FIFO (first-in-first-out) order.

Each session maintains separate message buffers for messages that are created by the `PACK_MESSAGE` function and messages that are retrieved by the `RECEIVE_MESSAGE` function. The separate message buffers allow you to build and receive messages in the same session. However, when consecutive calls are made to the `RECEIVE_MESSAGE` function, only the message from the last `RECEIVE_MESSAGE` call is preserved in the local message buffer.

Example C-4 shows a simple example that creates a pipe and sends a message through the pipe.

Example C-4 Sending a message by way of pipe

```
DECLARE
    status INT;
BEGIN
    status := DBMS_PIPE.CREATE_PIPE( 'pipe1' );
    status := DBMS_PIPE.PACK_MESSAGE('message1');
    status := DBMS_PIPE.SEND_MESSAGE( 'pipe1' );
END;
/
```

Example C-5 shows reading the message from a pipe.

Example C-5 Receiving a message from PIPE

```
DECLARE
    status INTEGER;
    itemType INTEGER;
    string1 VARCHAR(50);
BEGIN
    status := DBMS_PIPE.RECEIVE_MESSAGE( 'pipe1' );
    IF ( status = 0 ) THEN
        itemType := DBMS_PIPE.NEXT_ITEM_TYPE();
        IF ( itemType = 9 ) THEN
            DBMS_PIPE.UNPACK_MESSAGE_CHAR( string1 );
```

```
        DBMS_OUTPUT.PUT_LINE( 'string1 is: ' || string1 );
    ELSE
        DBMS_OUTPUT.PUT_LINE( 'unexpected data!');
    END IF;
END IF;
END;
/
```

More information and examples about DBMS_PIPE are available in the DB2 Information Center, found at:

<http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/topic/com.ibm.db2.luw.sql.rtn.doc/doc/r0053678.html>

C.4 DBMS_JOB

DBMS_JOB provides a set of procedures for creating, scheduling, and managing jobs. DBMS_JOB is an alternate interface for the Administrative Task Scheduler (ATS). To use the DBMS_JOB package, you must activate the Administrative Task Scheduler (ATS). This facility is turned off by default, although you are still able to define and modify jobs (tasks). You could enable the ATS by setting its registry variable as follows:

```
db2set DB2_ATS_ENABLE=YES
```

For more details about this topic, refer to the “Setting up the administrative task scheduler” topic in the Information Center, found at:

<http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/topic/com.ibm.db2.luw.admin.gui.doc/doc/t0054396.html>

Example C-6 shows the running of a job called job_proc at SYSDATE, with an automatic scheduled run at SYSDATE+1.

Example C-6 DBMS_JOB examples

```
CREATE OR REPLACE PROCEDURE job_proc
IS
BEGIN
    INSERT INTO jobrun VALUES ('job_proc run at ' || TO_CHAR(SYSDATE,
        'yyyy-mm-dd hh24:mi:ss'));
END;
/

DECLARE
    jobid          INTEGER;
BEGIN

    DBMS_JOB.SUBMIT(jobid,'job_proc;',SYSDATE, 'SYSDATE + 1');

END;
/
```

C.5 DBMS_LOB

DBMS_LOB provides a set of routines for operating on large objects (LOBs). In Example C-7, the APPEND and ERASE routines are invoked from DB2 Command Line to quickly demonstrate the actions performed on the LOB.

Example C-7 DBMS_LOB examples

```
call DBMS_LOB.APPEND_CLOB('ABCD','1234')
```

```
Value of output parameters
```

```
-----
Parameter Name  : DEST_LOB
Parameter Value : ABCD1234
```

```
Return Status = 0
```

```
call DBMS_LOB.ERASE_CLOB('DBMS', 1,3)
```

```
Value of output parameters
```

```
-----
Parameter Name  : LOB_LOC
Parameter Value : DB S
```


Parameter Name : AMOUNT
Parameter Value : 1

ReturnStatus=0

You can also use a simple anonymous block to illustrate the usage, as shown Example C-8.

Example C-8 DBMS_LOB in anonymous block

```
DECLARE
    v_dest_lob CLOB := 'ABCD';
BEGIN
    DBMS_OUTPUT.PUT_LINE('Original lob: ' || v_dest_lob);
    DBMS_LOB.APPEND_CLOB(v_dest_lob, '1234');
    DBMS_OUTPUT.PUT_LINE('New lob : ' || v_dest_lob);
END;
/

DECLARE
    v_dest_lob CLOB := 'DBMS';
    v_amount INTEGER := 1;
BEGIN
    DBMS_OUTPUT.PUT_LINE('Original lob: ' || v_dest_lob);
    DBMS_LOB.ERASE_CLOB(v_dest_lob, v_amount, 3);
    DBMS_OUTPUT.PUT_LINE('New lob : ' || v_dest_lob);
    DBMS_OUTPUT.PUT_LINE('Amount : ' || v_amount);
END;
/
```

C.6 DBMS_SQL

The DBMS_SQL package provides a set of procedures for executing dynamic SQL, and therefore supports various DML or DDL statements. The routines in the DBMS_SQL package are useful when you want to construct and execute dynamic SQL statements or call a function that uses dynamic SQL from within an SQL statement. Some DDL commands, such as “ALTER TABLE” or “DROP TABLE”, can also be prepared and executed “on-the-fly”.

Table C-4 lists the system-defined routines included in the DBMS_SQL module.

Table C-4 System-defined routines available in the DBMS_SQL module

Procedure name	Description
BIND_VARIABLE_BLOB procedure	Provides the input BLOB value for the IN or INOUT parameter and defines the data type of the output value to be BLOB for the INOUT or OUT parameter.
BIND_VARIABLE_CHAR procedure	Provides the input CHAR value for the IN or INOUT parameter and defines the data type of the output value to be CHAR for the INOUT or OUT parameter.
BIND_VARIABLE_CLOB procedure	Provides the input CLOB value for the IN or INOUT parameter and defines the data type of the output value to be CLOB for the INOUT or OUT parameter.
BIND_VARIABLE_DATE procedure	Provides the input DATE value for the IN or INOUT parameter and defines the data type of the output value to be DATE for the INOUT or OUT parameter.
BIND_VARIABLE_DOUBLE procedure	Provides the input DOUBLE value for the IN or INOUT parameter and defines the data type of the output value to be DOUBLE for the INOUT or OUT parameter.
BIND_VARIABLE_INT procedure	Provides the input INTEGER value for the IN or INOUT parameter and defines the data type of the output value to be INTEGER for the INOUT or OUT parameter.
BIND_VARIABLE_NUMBER procedure	Provides the input DECFLOAT value for the IN or INOUT parameter and defines the data type of the output value to be DECFLOAT for the INOUT or OUT parameter.
BIND_VARIABLE_RAW procedure	Provides the input BLOB(32767) value for the IN or INOUT parameter and defines the data type of the output value to be BLOB(32767) for the INOUT or OUT parameter.
BIND_VARIABLE_TIMESTAMP procedure	Provides the input TIMESTAMP value for the IN or INOUT parameter and defines the data type of the output value to be TIMESTAMP for the INOUT or OUT parameter.

Procedure name	Description
BIND_VARIABLE_VARCHAR procedure	Provides the input VARCHAR value for the IN or INOUT parameter and defines the data type of the output value to be VARCHAR for the INOUT or OUT parameter.
CLOSE_CURSOR procedure	Closes a cursor.
COLUMN_VALUE_BLOB procedure	Retrieves the value of a column of type BLOB.
COLUMN_VALUE_CHAR procedure	Retrieves the value of a column of type CHAR.
COLUMN_VALUE_CLOB procedure	Retrieves the value of a column of type CLOB.
COLUMN_VALUE_DATE procedure	Retrieves the value of a column of type DATE.
COLUMN_VALUE_DOUBLE procedure	Retrieves the value of a column of type DOUBLE.
COLUMN_VALUE_INT procedure	Retrieves the value of a column of type INTEGER.
COLUMN_VALUE_LONG procedure	Retrieves the value of a column of type CLOB(32767).
COLUMN_VALUE_NUMBER procedure	Retrieves the value of a column of type DECFLOAT.
COLUMN_VALUE_RAW procedure	Retrieves the value of a column of type BLOB(32767).
COLUMN_VALUE_TIMESTAMP procedure	Retrieves the value of a column of type TIMESTAMP.
COLUMN_VALUE_VARCHAR procedure	Retrieves the value of a column of type VARCHAR.
DEFINE_COLUMN_BLOB procedure	Defines the data type of the column to be BLOB.
DEFINE_COLUMN_CHAR procedure	Defines the data type of the column to be CHAR.
DEFINE_COLUMN_CLOB procedure	Defines the data type of the column to be CLOB.
DEFINE_COLUMN_DATE procedure	Defines the data type of the column to be DATE.
DEFINE_COLUMN_DOUBLE procedure	Defines the data type of the column to be DOUBLE.

Procedure name	Description
DEFINE_COLUMN_INT procedure	Defines the data type of the column to be INTEGER.
DEFINE_COLUMN_LONG procedure	Defines the data type of the column to be CLOB(32767).
DEFINE_COLUMN_NUMBER procedure	Defines the data type of the column to be DECFLOAT.
DEFINE_COLUMN_RAW procedure	Defines the data type of the column to be BLOB(32767).
DEFINE_COLUMN_TIMESTAMP procedure	Defines the data type of the column to be TIMESTAMP.
DEFINE_COLUMN_VARCHAR procedure	Defines the data type of the column to be VARCHAR.
DESCRIBE_COLUMNS procedure	Return a description of the columns retrieved by a cursor.
DESCRIBE_COLUMNS2 procedure	Identical to DESCRIBE_COLUMNS, but allows for column names greater than 32 characters.
EXECUTE procedure	Executes a cursor.
EXECUTE_AND_FETCH procedure	Executes a cursor and fetches one row.
FETCH_ROWS procedure	Fetches rows from a cursor.
IS_OPEN procedure	Checks if a cursor is open.
LAST_ROW_COUNT procedure	Returns the total number of rows fetched.
OPEN_CURSOR procedure	Opens a cursor.
PARSE procedure	Parses a DDL statement.
VARIABLE_VALUE_BLOB procedure	Retrieves the value of INOUT or OUT parameters as BLOB.
VARIABLE_VALUE_CHAR procedure	Retrieves the value of INOUT or OUT parameters as CHAR.
VARIABLE_VALUE_CLOB procedure	Retrieves the value of INOUT or OUT parameters as CLOB.
VARIABLE_VALUE_DATE procedure	Retrieves the value of INOUT or OUT parameters as DATE.
VARIABLE_VALUE_DOUBLE procedure	Retrieves the value of INOUT or OUT parameters as DOUBLE.
VARIABLE_VALUE_INT procedure	Retrieves the value of INOUT or OUT parameters as INTEGER.

Procedure name	Description
VARIABLE_VALUE_NUMBER procedure	Retrieves the value of INOUT or OUT parameters as DECFLOAT.
VARIABLE_VALUE_RAW procedure	Retrieves the value of INOUT or OUT parameters as BLOB(32767).
VARIABLE_VALUE_TIMESTAMP procedure	Retrieves the value of INOUT or OUT parameters as TIMESTAMP.
VARIABLE_VALUE_VARCHAR procedure	Retrieves the value of INOUT or OUT parameters as VARCHAR.

As you have probably noticed, some of the names of the procedures in this module are more detailed than the corresponding names in the Oracle package. For example, depending on the data type they will handle, the names of the DBMS_SQL.COLUMN_VALUE procedures could be qualified as COLUMN_VALUE_NUMBER, COLUMN_VALUE_CHAR, or COLUMN_VALUE_DATE, which is the data type to be bound.

Table C-5 lists the system-defined types and constants available in the DBMS_SQL package.

Table C-5 DBMS_SQL system-defined types and constants

Name	Type or constant	Description
DESC_REC	Type	A record of column information.
DESC_REC2	Type	A record of column information.
DESC_TAB	Type	An array of records of type DESC_REC.
DESC_TAB2	Type	An array of records of type DESC_REC2.
NATIVE	Constant	The only value supported for the language_flag parameter of the PARSE procedure.

The procedure EMPLOYEE_DYNAMIC_QUERY in Appendix E, “Test cases” on page 279 shows an implementation example.

C.7 DBMS_UTILITY

The DBMS_UTILITY package provides various utility programs for analyzing a database or a schema, compiling or validating objects, executing a DDL statement, getting information about database objects, database version, and CPU, and other functions.

Table C-6 lists the system-defined routines available in the DBMS_UTILITY module.

Table C-6 System-defined routines available in the DBMS_UTILITY module

Routine name	Description
ANALYZE_DATABASE procedure	Provides the capability to gather statistics on tables, clusters, and indexes in the database.
ANALYZE_PART_OBJECT procedure	Analyzes a partitioned table or partitioned index.
ANALYZE_SCHEMA procedure	Provides the capability to gather statistics on tables, clusters, and indexes in the specified schema.
CANONICALIZE procedure	Canonicalizes a string (for example, strips off white space).
COMMA_TO_TABLE procedure	Converts a comma-delimited list of names into an array of names where each entry in the list becomes an element in the array.
COMPILE_SCHEMA procedure	Recompiles all objects (functions, procedures, triggers, packages, and so on) in a schema.
DB_VERSION procedure	Returns the version number of the database.
EXEC_DDL_STATEMENT procedure	Executes a DDL statement.
GET_CPU_TIME function	Returns the CPU time in hundredths of a second from some arbitrary point in time.
GET_DEPENDENCY procedure	Lists all objects that are dependent upon the given object.
GET_HASH_VALUE function	Computes a hash value for a given string.
GET_TIME function	Returns the current time in hundredths of a second.
NAME_RESOLVE procedure	Obtains schema and other membership information of a database object. (Synonyms are resolved to their base objects.)
NAME_TOKENIZE procedure	Parses the given name into its component parts. (Names without double quotes are put into uppercase, and double quotes are stripped from names with double quotes.)
TABLE_TO_COMMA procedure	Converts an array of names into a comma-delimited list of names. Each array element becomes a list entry.
VALIDATE procedure	Provides the capability to change the state of an invalid routine to valid.

Table C-7 lists the system-defined variables and types available in the DBMS_UTILITY package.

Table C-7 DBMS_UTILITY public variables

Public variables	Data type	Description
lname_array	TABLE	For lists of long names
uncl_array	TABLE	For lists of users and names

Example C-9 shows the use of COMMA_TO_TABLE to change a comma delimited list into a table.

Example C-9 COMMA_TO_TABLE examples

```
CREATE OR REPLACE PROCEDURE comma_to_table (  
    p_list      VARCHAR2  
)  
IS  
    r_lname     DBMS_UTILITY.LNAME_ARRAY;  
    v_length    BINARY_INTEGER;  
BEGIN  
    DBMS_UTILITY.COMMA_TO_TABLE_LNAME(p_list,v_length,r_lname);  
    FOR i IN 1..v_length LOOP  
        DBMS_OUTPUT.PUT_LINE(r_lname(i));  
    END LOOP;  
END;  
  
call comma_to_table('schema.dept, schema.emp, schema.jobhist')  
schema.dept  
schema.emp  
schema.jobhist
```

For details and examples about the usage of these procedures and variables, refer to the DB2 Information Center, found at:

<http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/topic/com.ibm.db2.luw.sql.rtn.doc/doc/r0055155.html>

C.8 UTL_FILE

The UTL_FILE package provides a set of routines for reading from and writing to files located on the file system of the database server. Table C-8 lists the system-defined routines available in the UTL_FILE module.

Table C-8 System-defined routines in the UTL_FILE module

Routine name	Description
FCLOSE procedure	Closes a specified file.
FCLOSE_ALL procedure	Closes all open files.
FCOPY procedure	Copies text from one file to another.
FFLUSH procedure	Flushes unwritten data to a file.
FOPEN function	Opens a file.
FREMOVE procedure	Removes a file.
FRENAME procedure	Renames a file.
GET_LINE procedure	Gets a line from a file.
IS_OPEN function	Determines whether a specified file is open.
NEW_LINE procedure	Writes an end-of-line character sequence to a file.
PUT procedure	Writes a string to a file.
PUT_LINE procedure	Writes a single line to a file.
PUTF procedure	Writes a formatted string to a file.
UTL_FILE.FILE_TYPE procedure	Stores a file handle.

Table C-9 lists the named conditions (called “exceptions” in Oracle) that an application can receive.

Table C-9 Named conditions for an application

Condition name	Description
access_denied	Access to the file is denied by the operating system.
charsetmismatch	A file was opened using FOPEN_NCHAR, but later I/O operations used non-CHAR functions, such as PUTF or GET_LINE.
delete_failed	Unable to delete file.
file_open	File is already open.
internal_error	Unhandled internal error in the UTL_FILE package.

Condition name	Description
invalid_filehandle	File handle does not exist.
invalid_filename	A file with the specified name does not exist in the path.
invalid_maxlinesize	The MAX_LINESIZE value for FOPEN is invalid. It must be between 1 and 32672.
invalid_mode	The open_mode argument in FOPEN is invalid.
invalid_offset	The ABSOLUTE_OFFSET argument for FSEEK is invalid. It must be greater than 0 and less than the total number of bytes in the file.
invalid_operation	File could not be opened or operated on as requested.
invalid_path	The specified path does not exist or is not visible to the database.
read_error	Unable to read the file.
rename_failed	Unable to rename the file.
write_error	Unable to write to the file.

In Oracle, prior to using the UTL_FILE package in Oracle, you have to create a file system directory where the files will be located using the CREATE OR REPLACE DIRECTORY command. As this is a DDL statement, to incorporate this command in PL/SQL code, you have to use the EXECUTE IMMEDIATE statement. DB2 simplifies the implementation. To reference directories on a file system, simply use a directory alias, created by directly calling the UTL_DIR.CREATE_DIRECTORY or UTL_DIR.CREATE_OR_REPLACE_DIRECTORY system procedures.

Example C-10 presents a comparison of both statements.

Example C-10 Create directory statement in Oracle and DB2

```
-- Oracle syntax:
CREATE OR REPLACE DIRECTORY mydir AS 'home/user/temp/mydir';

-- DB2 syntax:
BEGIN
    UTL_DIR.CREATE_DIRECTORY('mydir', '/home/user/temp/mydir');
END;
/
```

A more detailed description of the UTL_DIR package is provided in “UTL_DIR” on page 266.

Because the UTL_FILE package executes file operations by using the authentication of the DB2 instance user ID, make sure that the DB2 instance user ID has the appropriate operating system permissions.

The SAVE_ORG_STRUCT_TO_FILE function in Appendix E, “Test cases” on page 279 demonstrates a usage of the UTL_FILE and UTL_DIR packages.

C.9 UTL_DIR

The UTL_DIR package provides a set of routines for maintaining directory aliases that are used with the UTL_FILE package. You can create (or create or replace), drop, and gather information about the directory alias using this package.

Example C-11 demonstrates how to retrieve the corresponding path for a directory alias.

Example C-11 Get the path for a directory alias

```
SET SERVEROUTPUT ON
/

DECLARE
    v_dir VARCHAR2(200);
BEGIN
    UTL_DIR.GET_DIRECTORY_PATH('mydir', v_dir );
    DBMS_OUTPUT.PUT_LINE('Directory path: ' || v_dir);
END;
/
```

--This example results in the following output:

```
Value of output parameters
-----
Parameter Name   : PATH
Parameter Value  : home/myuser/temp/mydir
Return Status    = 0
```

Example C-10 on page 265 shows how to create a directory alias. Alternatively, you can use the CREATE_OR_REPLACE_DIRECTORY procedure. Directory information is stored in SYSTOOLS.DIRECTORIES, which is created in the SYSTOOLSPACE when you first reference this package for each database.

Example C-12 on page 267 uses the DROP_DIRECTORY procedure to drop the specified directory alias.

Example C-12 Drop a directory alias

```
BEGIN
    UTL_DIR.DROP_DIRECTORY('mydir');
END;
/
```

C.10 UTL_MAIL

The UTL_MAIL package provides the capability to send e-mail with or without an attachment. In order to successfully send an e-mail message using the UTL_MAIL package, the database configuration parameter SMTP_SERVER must contain one or more valid Simple Mail Transfer Protocol (SMTP) server addresses.

Table C-10 lists the system-defined routines included in the UTL_MAIL module.

Table C-10 System-defined routines available in the UTL_MAIL module

Routine name	Description
SEND procedure	Packages and sends an e-mail to an SMTP server.
SEND_ATTACH_RAW procedure	Same as the SEND procedure, but with BLOB (binary) attachments.
SEND_ATTACH_VARCHAR2 procedure	Same as the SEND procedure, but with VARCHAR (text) attachments.

Example C-13 shows how to set up your SMTP server entries. To set up a list of SMTP servers, separate them by commas and provide different port numbers. An e-mail will be sent to each of the SMTP servers, in the order listed, until a successful reply is received from one of the SMTP servers.

Example C-13 SMTP server entry setup

```
-- Set up a single SMTP server that uses port 2000:
-- (if a port is not specified, the default port 25 will be used)
db2 update db cfg using smtp_server 'smtp2.ibm.com:2000'
--
-- Set up a list of SMTP server
db2 update db cfg using smtp_server
    'smtp1.example.com, smtp2.example.com:23, smtp3.example.com:2000'
```

Example C-14 demonstrates an anonymous block that sends an e-mail message.

Example C-14 Sending an e-mail with the UTL_MAIL package

```
DECLARE
    v_sender      VARCHAR2(50);
    v_recipients  VARCHAR2(50);
    v_subj        VARCHAR2(250);
    v_msg         VARCHAR2(200);
BEGIN
    v_sender := 'ibm_user@ibm.com';
    v_recipients := 'recipient1@mycompany.com, recipient2@ mycompany.com';
    v_subj := 'Test UTL_MAIL package on DB2';
    v_msg := ' UTL_MAIL package works great! ' ||
            'Please, setup properly your server!';
    UTL_MAIL.SEND(v_sender,v_recipients,NULL,NULL,v_subj,v_msg);
END;
/
```

C.11 UTL_SMTP

The UTL_SMTP package provides a set of routines for sending e-mail over SMTP.

Table C-11 lists the system-defined routines included in the UTL_SMTP module.

Table C-11 System-defined routines available in the UTL_SMTP module

Routine name	Description
CLOSE_DATA procedure	Ends an e-mail message.
COMMAND procedure	Executes an SMTP command.
COMMAND_REPLIES procedure	Executes an SMTP command where multiple reply lines are expected.
DATA procedure	Specifies the body of an e-mail message.
EHLO procedure	Performs initial handshaking with an SMTP server and returns extended information.
HELO procedure	Performs initial handshaking with an SMTP server.
HELP procedure	Sends the HELP command.
MAIL procedure	Starts a mail transaction.
NOOP procedure	Sends the null command.

Routine name	Description
OPEN_CONNECTION function	Opens a connection.
OPEN_CONNECTION procedure	Opens a connection.
OPEN_DATA procedure	Sends the DATA command.
QUIT procedure	Terminates the SMTP session and disconnects.
RCPT procedure	Specifies the recipient of an e-mail message.
RSET procedure	Terminates the current mail transaction.
VERFY procedure	Validates an e-mail address.
WRITE_DATA procedure	Writes a portion of the e-mail message.
WRITE_RAW_DATA procedure	Writes a portion of the e-mail message consisting of RAW data.

Table C-12 lists the public variables available in the package.

Table C-12 . System-defined types available in the UTL_SMTP package

Public variable	Data type	Description
connection	RECORD	Provides a description of an SMTP connection.
reply	RECORD	Provides a description of an SMTP reply line. (REPLIES is an array of SMTP reply lines.)

The procedure `send_mail` in Example C-15 constructs and sends a text e-mail message using the `UTL_SMTP` package and the `UTL_SMTP.DATA` method. This example also demonstrates how to call this procedure.

Example C-15 Sending message with UTL_SMTP package

```
CREATE OR REPLACE PROCEDURE send_mail (
    p_sender      VARCHAR2,
    p_recipient    VARCHAR2,
    p_subj         VARCHAR2,
    p_msg          VARCHAR2,
    p_mailhost     VARCHAR2
)
IS
    v_conn         UTL_SMTP.CONNECTION;
    v_crlf         CONSTANT VARCHAR2(2) := CHR(13) || CHR(10);
    v_port         CONSTANT PLS_INTEGER := 25;
BEGIN
    UTL_SMTP.OPEN_CONNECTION(p_mailhost,v_port, v_conn, 20, v_reply);
    UTL_SMTP.HELO(v_conn,p_mailhost);
    UTL_SMTP.MAIL(v_conn,p_sender);
    UTL_SMTP.RCPT(v_conn,p_recipient);
    UTL_SMTP.DATA(v_conn, SUBSTR(
        'Date: ' || TO_CHAR(SYSDATE,
        'Dy, DD Mon YYYY HH24:MI:SS') || v_crlf
        || 'From: ' || p_sender || v_crlf
        || 'To: ' || p_recipient || v_crlf
        || 'Subject: ' || p_subj || v_crlf
        || p_msg
        , 1, 32767));
    UTL_SMTP.QUIT(v_conn);
END;
/
```

```
call send_mail('name1@mycompany.com', 'name2@ mycompany.com', 'Test UTL_SMTP
package on DB2', 'Please, setup properly your server!', 'smtp.mycompany.com');
```

Example C-16 uses the OPEN_DATA, WRITE_DATA, and CLOSE_DATA procedures instead of the DATA procedure. The call to the send_mail_2 procedure is identical to the call of send_mail in Example C-15 on page 270.

Example C-16 Sending a message with the UTL_SMTP package

```
CREATE OR REPLACE PROCEDURE send_mail_2 (  
    p_sender      VARCHAR2,  
    p_recipient   VARCHAR2,  
    p_subj        VARCHAR2,  
    p_msg         VARCHAR2,  
    p_mailhost    VARCHAR2  
)  
IS  
    v_conn        UTL_SMTP.CONNECTION;  
    v_crlf        CONSTANT VARCHAR2(2) := CHR(13) || CHR(10);  
    v_port        CONSTANT PLS_INTEGER := 25;  
BEGIN  
    UTL_SMTP.OPEN_CONNECTION(p_mailhost,v_port,v_conn, 20, v_reply);  
    UTL_SMTP.HELO(v_conn,p_mailhost);  
    UTL_SMTP.MAIL(v_conn,p_sender);  
    UTL_SMTP.RCPT(v_conn,p_recipient);  
    UTL_SMTP.OPEN_DATA(v_conn);  
    UTL_SMTP.WRITE_DATA(v_conn,'From: ' || p_sender || v_crlf);  
    UTL_SMTP.WRITE_DATA(v_conn,'To: ' || p_recipient || v_crlf);  
    UTL_SMTP.WRITE_DATA(v_conn,'Subject: ' || p_subj || v_crlf);  
    UTL_SMTP.WRITE_DATA(v_conn,v_crlf || p_msg);  
    UTL_SMTP.CLOSE_DATA(v_conn);  
    UTL_SMTP.QUIT(v_conn);  
END;  
/
```

DB2 OCI sample program

This appendix provides a sample DB2 Oracle Call Interface (OCI) program.

Example D-1 shows a simplified DB2 OCI program that reads data from the ORG table in the DB2 SAMPLE database. It illustrates some of the common OCI calls that have identical usage with Oracle. This example uses utility functions included in `utilci.c`, and `utilci.h` that are part of the shipped OCI samples in DB2 9.7 Fix Pack 1.

Example D-1 DB2 OCI table read example

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <db2ci.h>
#include "utilci.h" /* Header file for DB2CI sample code */

#define ROWSET_SIZE 5

int TbSelectWithParam( OCIEEnv * envhp, OCISvcCtx * svchp, OCIError * errhp );

int main(int argc, char *argv[])
{
    sb4 ciRC = OCI_SUCCESS;
    int rc = 0;
    OCIEEnv * envhp; /* environment handle */
    OCISvcCtx * svchp; /* connection handle */
    OCIError * errhp; /* error handle */
```

```

char dbAlias[SQL_MAX_DSN_LENGTH + 1];
char user[MAX_UID_LENGTH + 1];
char pswd[MAX_PWD_LENGTH + 1];

/* check the command line arguments */
rc = CmdLineArgsCheck1(argc, argv, dbAlias, user, pswd);
if (rc != 0)
{
    return rc;
}

printf("\nTHIS SAMPLE SHOWS HOW TO READ TABLES.\n");

/* initialize the DB2CI application by calling a helper
   utility function defined in utilci.c */
rc = CIAppInit(dbAlias,
               user,
               pswd,
               &envhp,
               &svchp,
               &errhp );
if (rc != 0)
{
    return rc;
}

/* SELECT with parameter markers */
rc = TbSelectWithParam( envhp, svchp, errhp );

/* terminate the DB2CI application by calling a helper
   utility function defined in utilci.c */
rc = CIAppTerm(&envhp, &svchp, errhp, dbAlias);

return rc;
}/* main */

perform a SELECT that contains parameter markers */
int TbSelectWithParam( OCIEnv * envhp, OCISvcCtx * svchp, OCLError * errhp )
{
    sb4 ciRC = OCI_SUCCESS;
    int rc = 0;
    OCISmt * hstmt; /* statement handle */
    OCIDefine * defnhp1 = NULL; /* define handle */
    OCIDefine * defnhp2 = NULL; /* define handle */
    OCIBind * hBind = NULL; /* bind handle */

    char *stmt = (char *)
"SELECT deptnumb, location FROM org WHERE division = :1";

    char divisionParam[15];

    struct
    {

```

```

        sb2 ind;
        sb2 val;
        ub2 length;
        ub2 rcode;
    }
    deptnumb; /* variable to be bound to the DEPTNUMB column */

    struct
    {
        sb2 ind;
        char val[15];
        ub2 length;
        ub2 rcode;
    }
    location; /* variable to be bound to the LOCATION column */

    printf("\n-----");
    printf("\nUSE THE DB2CI FUNCTIONS\n");
    printf("  OCIHandleAlloc\n");
    printf("  OCIStmtPrepare\n");
    printf("  OCIStmtExecute\n");
    printf("  OCIBindByPos\n");
    printf("  OCIDefineByPos\n");
    printf("  OCIStmtFetch\n");
    printf("  OCIHandleFree\n");
    printf("TO PERFORM A SELECT WITH PARAMETERS:\n");

    /* allocate a statement handle */
    ciRC = OCIHandleAlloc( (dvoid *)envhp, (dvoid **)&hstmt, OCI_HTYPE_STMT, 0, NU
LL );
    ERR_HANDLE_CHECK(errhp, ciRC);

    printf("\n  Prepare the statement\n");
    printf("    %s\n", stmt);
    /* prepare the statement */
    ciRC = OCIStmtPrepare(
        hstmt,
        errhp,
        (OraText *)stmt,
        strlen( stmt ),
        OCI_NTV_SYNTAX,
        OCI_DEFAULT );
    ERR_HANDLE_CHECK(errhp, ciRC);

    printf("\n  Bind divisionParam to the statement\n");
    printf("    %s\n", stmt);

    /* bind divisionParam to the statement */
    ciRC = OCIBindByPos(
        hstmt,
        &hBind,
        errhp,
        1,
        divisionParam,

```

```

        sizeof( divisionParam ),
        SQLT_STR,
        NULL,
        NULL,
        NULL,
        0,
        NULL,
        OCI_DEFAULT );
ERR_HANDLE_CHECK(errhp, ciRC);

/* execute the statement for divisionParam = Eastern */
printf("\n Execute the prepared statement for\n");
printf("    divisionParam = 'Eastern'\n");
strcpy(divisionParam, "Eastern");
/* execute the statement */
ciRC = OCISStmtExecute(
    svchp,
    hstmt,
    errhp,
    0,
    0,
    NULL,
    NULL,
    OCI_DEFAULT );
ERR_HANDLE_CHECK(errhp, ciRC);

/* bind column 1 to variable */
ciRC = OCIDefineByPos(
    hstmt,
    &defnbp1,
    errhp,
    1,
    &deptnumb.val,
    sizeof( sb2 ),
    SQLT_INT,
    &deptnumb.ind,
    &deptnumb.length,
    &deptnumb.rcode,
    OCI_DEFAULT );
ERR_HANDLE_CHECK(errhp, ciRC);

/* bind column 2 to variable */
ciRC = OCIDefineByPos(
    hstmt,
    &defnbp2,
    errhp,
    2,
    location.val,
    sizeof( location.val ),
    SQLT_STR,
    &location.ind,
    &location.length,
    &location.rcode,
    OCI_DEFAULT );

```

```

ERR_HANDLE_CHECK(errhp, ciRC);

printf("\n  Fetch each row and display.\n");
printf("    DEPTNUMB LOCATION      \n");
printf("    ----- \n");

/* fetch each row and display */
ciRC = OCISmtFetch(
    hstmt,
    errhp,
    1,
    OCI_FETCH_NEXT,
    OCI_DEFAULT );
ERR_HANDLE_CHECK(errhp, ciRC);

if (ciRC == OCI_NO_DATA )
{
    printf("\n  Data not found.\n");
}
while (ciRC != OCI_NO_DATA )
{
    printf("    %-8d %-14.14s \n", deptnumb.val, location.val);

    /* fetch next row */
    ciRC = OCISmtFetch(
        hstmt,
        errhp,
        1,
        OCI_FETCH_NEXT,
        OCI_DEFAULT );
    ERR_HANDLE_CHECK(errhp, ciRC);
}

/* free the statement handle */
ciRC = OCIHandleFree( hstmt, OCI_HTYPE_STMT );
ERR_HANDLE_CHECK(errhp, ciRC);

return rc;
} /* TbSelectWithParam */

```

Example D-2 illustrate a typical compile and link steps for AIX platform.

Example D-2 Compiling and linking an OCI program

```

xlc -c test01.C -q64 -DUNIX -I$HOME/sql/lib/include -I$HOME/sql/lib/include
xlc -o test01 test01.o $HOME/sql/lib/lib/libdb2ci.a -lpthread -lm -ls -v
-bloadmap:map -bmaxdata:0x80000000 -q64
-b64

```



Test cases

This appendix provides test cases for enablement exercises for:

- ▶ Oracle DDL

The DDL used to create all database objects in the source Oracle database.

- ▶ DB2 DDL

The DB2 DDL contains the corresponding database objects enabled in the DB2 database.

These DDLs can be download from the IBM Redbooks Web site. Refer to Appendix F, “Additional material” on page 317 for the download instructions.

E.1 Oracle DDL

The sample DDLs listed here are used in this book for demonstration purposes.

E.1.1 Tables and views

```

CREATE SEQUENCE EMPLOYEE_SEQUENCE
MINVALUE 1
MAXVALUE 9999999999999999999999999999999
INCREMENT BY 1
START WITH 2
CACHE 20 NOCYCLE NOORDER
/

CREATE SEQUENCE CUSTOMER_SEQUENCE
MINVALUE 1
MAXVALUE 9999999999999999999999999999999
INCREMENT BY 1
START WITH 2
CACHE 20 NOCYCLE NOORDER
/

CREATE table DEPARTMENTS (
"DEPT_CODE"          CHAR(3) NOT NULL,
"DEPT_NAME"          VARCHAR2(30),
"TOTAL_PROJECTS"     NUMBER,
"TOTAL_EMPLOYEES"    NUMBER)
/

CREATE table ACCOUNTS (
"ACCT_ID"            NUMBER(38) NOT NULL,
"DEPT_CODE"          CHAR(3) NOT NULL,
"ACCT_DESC"           VARCHAR2(2000),
"MAX_EMPLOYEES"       NUMBER(3),
"CURRENT_EMPLOYEES"   NUMBER(3),
"NUM_PROJECTS"        NUMBER(1),
"CREATE_DATE"         DATE DEFAULT SYSDATE,
"CLOSED_DATE"         DATE DEFAULT SYSDATE)
/

CREATE table CUSTOMERS (
"CUST_ID"             NUMBER(5),
"CUST_DETAILS_XML"    XMLType ,
"LAST_UPDATE_DATE"    DATE)
/

CREATE table EMPLOYEES (
"EMP_ID"              NUMBER(5) NOT NULL,
"FIRST_NAME"          VARCHAR2(20),
"LAST_NAME"           VARCHAR2(20),
"CURRENT PROJECTS"    NUMBER(3),

```



```

"EMP_MGR_ID"      NUMBER(5),
"DEPT_CODE"       CHAR(3) NOT NULL,
"ACCT_ID"         NUMBER(3) NOT NULL,
"OFFICE_ID"       NUMBER(5),
"BAND"           CHAR(1),
"CREATE_DATE"     TIMESTAMP(3) DEFAULT SYSDATE)
/

CREATE table EMP_DETAILS (
    "EMP_ID"       NUMBER(5) NOT NULL,
    "EDUCATION"    CLOB NOT NULL,
    "WORK_EXPERIENCE" CLOB NOT NULL,
    "PHOTO_FORMAT" VARCHAR2(10) NOT NULL,
    "PICTURE"      BLOB)
/

CREATE table OFFICES (
    "OFFICE_ID"    NUMBER(5) NOT NULL,
    "BUILDING"     VARCHAR2(25),
    "NUMBER_SEATS" NUMBER(4),
    "DESCRIPTION"  VARCHAR2(50))
/

COMMENT ON TABLE DEPARTMENTS IS 'Contains information about departments and their names'
/

COMMENT ON TABLE ACCOUNTS IS 'Contains information about accounts and number of
projects'
/

COMMENT ON TABLE CUSTOMERS IS 'Contains information about customers in XML format'
/

COMMENT ON TABLE EMPLOYEES IS 'Contains information about employees'
/

COMMENT ON TABLE EMP_DETAILS IS 'Contains additional information on employees like
biography and picture'
/

COMMENT ON TABLE OFFICES IS 'Contains information about buildigs and offices'
/

ALTER TABLE DEPARTMENTS ADD CONSTRAINT PK_DEPT_CODE PRIMARY KEY ( "DEPT_CODE" )
/

ALTER TABLE ACCOUNTS ADD CONSTRAINT PK_ACCOUNTS PRIMARY KEY ( "DEPT_CODE", "ACCT_ID" )
/

ALTER TABLE EMPLOYEES ADD CONSTRAINT PK_EMPLOYEES PRIMARY KEY ( "EMP_ID" )
/

ALTER TABLE EMP_DETAILS ADD CONSTRAINT PK_EMP_DETAILS PRIMARY KEY ( "EMP_ID" )
/

```

```

ALTER TABLE OFFICES ADD CONSTRAINT PK_OFFICES PRIMARY KEY ( "OFFICE_ID" )
/

ALTER TABLE ACCOUNTS ADD CONSTRAINT FK_ACC_DEPT_CODE FOREIGN KEY ( "DEPT_CODE" )
REFERENCES DEPARTMENTS ( "DEPT_CODE" )
/

ALTER TABLE EMPLOYEES ADD CONSTRAINT FK_EMP_MGR_ID FOREIGN KEY ( "EMP_MGR_ID" )
REFERENCES EMPLOYEES ( "EMP_ID" )
/

ALTER TABLE EMPLOYEES ADD CONSTRAINT FK_EMP_OFFICE_ID FOREIGN KEY ( "OFFICE_ID" )
REFERENCES OFFICES ( "OFFICE_ID" )
/

ALTER TABLE EMPLOYEES ADD CONSTRAINT M_DEPT_CODE_ACCT_ID FOREIGN KEY ( "DEPT_CODE",
"ACCT_ID" ) REFERENCES ACCOUNTS ( "DEPT_CODE", "ACCT_ID" )
/

ALTER TABLE EMP_DETAILS ADD CONSTRAINT FK_EMP_DETAILS_ID FOREIGN KEY ( "EMP_ID" )
REFERENCES EMPLOYEES ( "EMP_ID" ) ON DELETE CASCADE
/

ALTER TABLE EMPLOYEES ADD CONSTRAINT BAND_VALIDATION CHECK (BAND IN ('1', '2', '3', '4',
'5'))
/

CREATE INDEX ACCT_DEPT_IND ON ACCOUNTS (DEPT_CODE)
/

CREATE INDEX EMP_SEARCH_IND ON EMPLOYEES(LAST_NAME, FIRST_NAME, DEPT_CODE)
/

CREATE INDEX CUSTOMER_CITY_IND ON CUSTOMERS a
(XMLType.getStringVal(
XMLType.extract(a.cust_details_xml,'//customer-details/addr/city/text()')))
/

CREATE GLOBAL TEMPORARY TABLE TEMP_TABLE (
"NUM_COL" NUMBER,
"CHAR_COL" VARCHAR2(60))
/

CREATE OR REPLACE VIEW ORGANIZATION_STRUCTURE ("ORG_LEVEL", "FULL_NAME", "DEPARTMENT")
AS
/*
||-----
|| DESCRIPTION: Dispay hierarchy of people in the organization structure
||
||
|| DEMO PURPOSE: Support of recursive SQL "START WITH ... CONNECT BY" along
|| with LEVEL keyword.
||
|| New built-in functions INITCAP and NVL.

```

```

||          New syntax for outer join - (+)
||-----
*/
SELECT
    LEVEL as ORG_LEVEL,
    SUBSTR((LPAD(' ', 4 * LEVEL - 1) || INITCAP(e.last_name) || ', ' ||
INITCAP(e.first_name)), 1, 40),
    NVL(d.dept_name, 'Uknown')
FROM
    EMPLOYEES e,
    DEPARTMENTS d
WHERE
    e.dept_code=d.dept_code(+)
START WITH emp_id = 1
CONNECT BY NOCYCLE PRIOR emp_id = emp_mgr_id
/

-- public synonyms on sequence
CREATE PUBLIC SYNONYM EMPLOYEE_SEQUENCE FOR SALES.EMPLOYEE_SEQUENCE//

-- public synonyms on tables
CREATE PUBLIC SYNONYM DEPARTMENTS FOR SALES.DEPARTMENTS //
CREATE PUBLIC SYNONYM EMPLOYEES FOR SALES.EMPLOYEES //
CREATE PUBLIC SYNONYM EMP_DETAILS FOR SALES.EMP_DETAILS/
CREATE PUBLIC SYNONYM OFFICES FOR SALES.OFFICES /

--public synonyms on views
CREATE PUBLIC SYNONYM ORGANIZATION_STRUCTURE FOR SALES.ORGANIZATION_STRUCTURE/

--public synonyms on packages
CREATE PUBLIC SYNONYM HELPER FOR SALES.HELPER/
CREATE PUBLIC SYNONYM ACCOUNT_PACKAGE FOR SALES.ACCOUNT_PACKAGE/

--public synonym on procedures for visualising information
CREATE PUBLIC SYNONYM EMPLOYEE_DYNAMIC_QUERY FOR SALES.EMPLOYEE_DYNAMIC_QUERY/
CREATE PUBLIC SYNONYM SAVE_ORG_TO_FILE FOR SALES.SAVE_ORG_TO_FILE/

CREATE OR REPLACE TYPE EMP_INFO_TYPE AS OBJECT (
    EMP_ID      NUMBER(5),
    FIRST_NAME  VARCHAR2(20),
    LAST_NAME   VARCHAR2(20),
    BAND        CHAR(1))
/

```

E.1.2 Packages, procedures, and functions

```
=====
-- PACKAGES CREATION
=====

CREATE OR REPLACE PACKAGE HELPER AS
/*
||-----
|| DESCRIPTION: the purpose of this package is to create types that can be used by
|| stand-alone procedures and functions
||
|| DEMO PURPOSE: Definition of package for creating new types, Reference Cursor,
||               Record data type, Variable arrays
||-----
|| */

-- type declaration
TYPE rct1 IS REF CURSOR;
TYPE emp_array_type IS VARRAY(10) OF EMP_INFO_TYPE;

END HELPER;
/

CREATE OR REPLACE PACKAGE ACCOUNT_PACKAGE AS
/*
||-----
|| DESCRIPTION: Account package contains the procedures to manage the accounts.
||               using the procedures in this package, users can add an account, remove
||               an account, provide account information in form of associative array
||               and display the account information using server output.
||
|| DEMO PURPOSE: Package header declaration, anchor datatypes like %TYPE and %ROWTYPE,
||               definition of associative array TYPE ... IS TABLE OF ... INDEX BY ...
||
||-----
|| */

-- type declaration
TYPE customer_name_cache IS TABLE OF EMPLOYEES%ROWTYPE INDEX BY PLS_INTEGER;

-- PROCEDURE declaration
PROCEDURE Add_Account(p_AccountId IN ACCOUNTS.acct_id%TYPE,
                    p_DeptCode   IN ACCOUNTS.dept_code%TYPE,
                    p_AccountDesc IN ACCOUNTS.acct_desc%TYPE,
                    p_MaxEmployees IN ACCOUNTS.max_employees%TYPE);

PROCEDURE Remove_Account(p_AccountId IN ACCOUNTS.acct_id%TYPE,
                       p_DeptCode   IN ACCOUNTS.dept_code%TYPE);
```

```

PROCEDURE Account_List(p_dept_code          IN ACCOUNTS.dept_code%TYPE,
                      p_acct_id            IN ACCOUNTS.acct_id%TYPE,
                      p_Employees_Name_Cache OUT Customer_Name_Cache);

PROCEDURE Display_Account_List(p_dept_code IN ACCOUNTS.dept_code%TYPE,
                              p_acct_id   IN ACCOUNTS.acct_id%TYPE);

END ACCOUNT_PACKAGE;
/

CREATE OR REPLACE PACKAGE BODY ACCOUNT_PACKAGE AS

PROCEDURE ADD_ACCOUNT(p_AccountId  IN ACCOUNTS.acct_id%TYPE,
                     p_DeptCode    IN ACCOUNTS.dept_code%TYPE,
                     p_AccountDesc  IN ACCOUNTS.acct_desc%TYPE,
                     p_MaxEmployees IN ACCOUNTS.max_employees%TYPE) IS

/*
||-----
|| DESCRIPTION: Add a new Employee into the specified Account
||
|| DEMO PURPOSE: predefined exceptions like DUP_VAL_ON_INDEX and OTHERS
||
|| EXAMPLE: EXEC ACCOUNT_PACKAGE.ADD_ACCOUNT(1, 1, 'Description', 10)
||-----
*/

BEGIN

INSERT INTO ACCOUNTS (acct_id, dept_code, acct_desc, max_employees,
current_employees, num_projects)
VALUES (p_AccountId, p_DeptCode, p_AccountDesc, p_MaxEmployees, 0, 0);

DBMS_OUTPUT.PUT_LINE('Account ' || p_AccountId || ' successfully created .');

EXCEPTION
WHEN dup_val_on_index THEN
DBMS_OUTPUT.PUT_LINE('Duplicate Account was rejected');
WHEN others THEN
DBMS_OUTPUT.PUT_LINE('SQLCODE: ' || SQLCODE);
DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
RAISE;

END ADD_ACCOUNT;

PROCEDURE REMOVE_ACCOUNT(p_AccountId IN ACCOUNTS.acct_id%TYPE,
                         p_DeptCode   IN ACCOUNTS.dept_code%TYPE) IS

/*
||-----
|| DESCRIPTION: Removes the Account from database based on the account id and

```

```

|| department code
||
|| DEMO PURPOSE: Exception declaration
||
|| EXAMPLE: EXEC ACCOUNT_PACKAGE.REMOVE_ACCOUNT(1, 1)
||-----
*/

-- exception declaration
e_AccountNotRegistered EXCEPTION;
PRAGMA EXCEPTION_INIT (e_AccountNotRegistered, -20050);

BEGIN
  DELETE FROM ACCOUNTS WHERE acct_id = p_AccountId AND dept_code = p_DeptCode;

  IF SQL%NOTFOUND THEN
    RAISE e_AccountNotRegistered;
  END IF;

  DBMS_OUTPUT.PUT_LINE('Account ' || p_AccountId || ' was successfully removed.');
```

```

EXCEPTION
  WHEN e_AccountNotRegistered THEN
    DBMS_OUTPUT.PUT_LINE('Account ' || p_AccountId || ' does not exist.');
```

```

  WHEN others THEN
    RAISE;

END REMOVE_ACCOUNT;
```

```

PROCEDURE ACCOUNT_LIST(p_dept_code          IN ACCOUNTS.dept_code%TYPE,
                       p_acct_id           IN ACCOUNTS.acct_id%TYPE,
                       p_Employees_Name_Cache OUT CUSTOMER_NAME_CACHE) IS
/*
||-----
|| DESCRIPTION: Stores all employees from particular department in the associative
|| array
||
|| DEMO PURPOSE: Cursor definition, Cursor iteration through LOOP, population of
|| associative array
||-----
*/

-- variable declaration
  v_NumEmployees NUMBER := 1;

-- cursor declaration
  CURSOR c_RegisteredEmployees IS
    -- Local cursor to fetch the registered Employees.
    SELECT *
```

```

        FROM EMPLOYEES
        WHERE dept_code = p_dept_code
        AND acct_id = p_acct_id;

BEGIN
/*   p_NumEmployees will be the table index. It will start at 0,
    and be incremented each time through the fetch loop.
    At the end of the loop, it will have the number of rows
    fetched, and therefore the number of rows returned in p_IDs.   */

    OPEN c_RegisteredEmployees;

    LOOP
        FETCH c_RegisteredEmployees INTO p_Employees_Name_Cache(v_NumEmployees);
        EXIT WHEN c_RegisteredEmployees%NOTFOUND;
        v_NumEmployees := v_NumEmployees + 1;
    END LOOP;

    CLOSE c_RegisteredEmployees;

END ACCOUNT_LIST;

PROCEDURE DISPLAY_ACCOUNT_LIST(p_dept_code IN ACCOUNTS.dept_code%TYPE,
                              p_acct_id   IN ACCOUNTS.acct_id%TYPE) IS
/*
|-----
| DESCRIPTION: Displays the information about Employees and number of their project
| assigned to specific account.
| DEMO PURPOSE: Usage of associative arrays, DBMS_OUTPUT built-in package,
|               call to the procedure in the same package
| EXAMPLE:  EXEC ACCOUNT_PACKAGE.DISPLAY_ACCOUNT_LIST('A00', 1);
|-----
*/
    -- variable declaration
    v_customer_name_cache CUSTOMER_NAME_CACHE;
    k                      NUMBER := 1;

    -- definition of the nested function
    FUNCTION AVERAGE_BAND ( p_Department IN EMPLOYEES.dept_code%TYPE,
                            p_ACCT_ID    IN EMPLOYEES.acct_id%TYPE)
    RETURN CHAR AS
/*
|-----
| DESCRIPTION: Nested procedure to derive the average band of employees in the
| department
| DEMO PURPOSE: Cursor definition, cursor attributes %NOTFOUND and %ROWCOUNT,
|               DECODE built-in function, user defined exception
|-----
*/

```

```

-- variable declaration
v_AverageBAND      CHAR(1);
v_NumericBand       NUMBER;
v_TotalBand         NUMBER:=0;
v_NumberEmployees   NUMBER;

-- CURSOR declaration
CURSOR c_Employees IS
SELECT band
  FROM EMPLOYEES
 WHERE dept_code = p_Deptament
       AND acct_id = p_ACCT_ID;

BEGIN
  OPEN c_Employees;
  LOOP
    FETCH c_Employees INTO v_NumericBand;
    EXIT WHEN c_Employees%NOTFOUND;
    v_TotalBand := v_TotalBand + v_NumericBand;
  END LOOP;

  v_NumberEmployees:=c_Employees%ROWCOUNT;
  IF(v_NumberEmployees = 0) THEN
    RAISE_APPLICATION_ERROR(-20001, 'No employees exist for ' || p_Deptament || ' '
|| p_ACCT_ID);
  END IF;

  SELECT DECODE(ROUND(v_TotalBand/v_NumberEmployees), 5, 'A', 4, 'B', 3, 'C', 2,
'D', 1, 'E')
    INTO v_AverageBand
  FROM DUAL;

  RETURN v_AverageBand;

END AVERAGE_BAND;

BEGIN
  DBMS_OUTPUT.PUT_LINE('List of employees');
  DBMS_OUTPUT.PUT_LINE('-----');
  account_list(p_dept_code, p_acct_id, v_customer_name_cache);

  FOR k IN 1..v_customer_name_cache.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE('Record id           : ' || k );
    DBMS_OUTPUT.PUT_LINE('Employee           : ' ||
v_customer_name_cache(k).last_name);
    DBMS_OUTPUT.PUT_LINE('Number of projects : ' ||
v_customer_name_cache(k).Current_Projects);
    DBMS_OUTPUT.PUT_LINE('Average Band in department : ' ||
average_band(v_customer_name_cache(k).dept_code, v_customer_name_cache(k).acct_id));
  END LOOP;

END DISPLAY_ACCOUNT_LIST;

```



```

END ACCOUNT_PACKAGE;
/
--public synonyms on packages
CREATE PUBLIC SYNONYM HELPER FOR MODULE SALES.HELPER;
CREATE PUBLIC SYNONYM ACCOUNT_PACKAGE FOR MODULE SALES.ACCOUNT_PACKAGE;

CREATE OR REPLACE FUNCTION COUNT_PROJECTS (p_empID IN employees.emp_ID%TYPE,
                                           o_acct_id OUT employees.acct_id%TYPE)
RETURN NUMBER AS
/*
|-----|
| DESCRIPTION: Function that counts the project based on the employee id and also
|               returns information on total projects of the account to which employee
|               ID belongs
| DEMO PURPOSE: Function with OUT parameter, FOR LOOP over cursor
|
| EXAMPLE: SELECT COUNT_PROJECTS(1, acct_id) FROM DUAL;
|-----|
*/
-- variable declaration
v_TotalProjects NUMBER:=0;
v_AccountProjects NUMBER;

-- CURSOR declaration
CURSOR c_DeptAccts IS
SELECT dept_code, acct_id
FROM EMPLOYEES
WHERE emp_id = p_empID;

BEGIN
FOR v_AccountRec IN c_DeptAccts LOOP
o_acct_id:=v_AccountRec.acct_id;
-- Determine the projects for this account.
SELECT num_projects
INTO v_AccountProjects
FROM ACCOUNTS
WHERE dept_code = v_AccountRec.dept_code
AND acct_id = v_AccountRec.acct_id;

-- Add it to the total so far.
v_Totalprojects := v_Totalprojects + v_AccountProjects;
END LOOP;
-- different line for DB2 and Oracle
RETURN v_Totalprojects;

END COUNT_PROJECTS;
/

CREATE OR REPLACE PROCEDURE ADD_NEW_EMPLOYEE (
p_FirstName EMPLOYEES.first_name%TYPE,
p_LastName EMPLOYEES.last_name%TYPE,

```

```

p_EmpMgrId      EMPLOYEES.emp_mgr_id%TYPE,
p_DeptCode      EMPLOYEES.dept_code%TYPE,
p_Account       EMPLOYEES.acct_id%TYPE,
o_Employee OUT  EMP_INFO_TYPE,
p_CreateDate    EMPLOYEES.create_date%TYPE DEFAULT SYSDATE,
p_OfficeId      EMPLOYEES.office_id%TYPE DEFAULT 2
) AS
/*
|-----
| DESCRIPTION: Procedure to add a new employee
|-----
| DEMO PURPOSE: Default values in the procedure definition, Regular loops,
|                 sequence keywords like NEXTVAL and CURVAL
|                 EXECUTE IMMEDIATE
|-----
| EXAMPLE: EXEC ADD_NEW_EMPLOYEE('Max', 'Trenton', 2, 1, 1, emp_info)
|-----
*/

-- variable declaration
v_EmployeeId      EMPLOYEES.emp_id%TYPE :=1;
v_EmployeeIdTemp  EMPLOYEES.emp_id%TYPE;

-- cursor declaration
CURSOR c_CheckEmployeeId IS
SELECT 1
FROM EMPLOYEES
WHERE emp_id=v_EmployeeId;

CURSOR c_get_employee IS
SELECT emp_id, first_name, last_name, band
FROM EMPLOYEES
WHERE emp_id=v_EmployeeId;

BEGIN
-- Find Next available employee id from the employee sequence
LOOP
SELECT employee_sequence.NEXTVAL INTO v_EmployeeId FROM DUAL;
OPEN c_CheckEmployeeId;
FETCH c_CheckEmployeeId INTO v_EmployeeIdTemp;
EXIT WHEN c_CheckEmployeeId%NOTFOUND;
CLOSE c_CheckEmployeeId;
END LOOP;

select employee_sequence.CURRVAL INTO v_EmployeeId FROM DUAL;

EXECUTE IMMEDIATE 'INSERT INTO EMPLOYEES(emp_id, first_name, last_name,
current_projects, emp_mgr_id, dept_code, acct_id, office_id, band, create_date)
VALUES ( ' || v_EmployeeId || ', UPPER('' || p_FirstName || ''),
UPPER('' || p_LastName || ''),

```

```

0, '|| p_EmpMgrId || ', '|| p_DeptCode || ', ' || p_Account ||
', ' || p_OfficeId || ', 1, '|| p_CreateDate || ' ' ' ');

```

```

FOR x IN c_get_employee
LOOP
o_Employee:=EMP_INFO_TYPE(x.emp_id, x.first_name, x.last_name, x.band);
END LOOP;
DBMS_OUTPUT.PUT_LINE('Employee record id ' || v_EmployeeId || ' was created
successfully. ');
EXCEPTION
WHEN others THEN
DBMS_OUTPUT.PUT_LINE('Employee record was not created. ');
RAISE;

```

```

END ADD_NEW_EMPLOYEE;
/

```

```

CREATE OR REPLACE PROCEDURE GET_EMPLOYEE_RESUME (p_empID IN employees.emp_ID%TYPE,
o_resume OUT CLOB) AS

```

```

/*
||-----
|| DESCRIPTION: Builds employee's resume in the CLOB format based on the employee id
||
|| DEMO PURPOSE: DBMS_LOB built-in package
||
|| EXAMPLE: EXEC GET_EMPLOYEE_RESUME(1, clob_resume)
||-----
*/

```

```

-- variable declaration
v_education CLOB;
v_work_experience CLOB;
v_picture BLOB;
v_position NUMBER:=1;
BEGIN
DBMS_LOB.CREATETEMPORARY(o_resume, TRUE, DBMS_LOB.session);
SELECT education, work_experience
INTO v_education, v_work_experience
FROM EMP_DETAILS
WHERE emp_id=p_empID;
DBMS_LOB.WRITE(o_resume, 7, v_position, 'Resume' || chr(10));
v_position:=v_position+7;
DBMS_LOB.WRITE(o_resume, 11, v_position, 'Education: ');
v_position:=v_position+11;
DBMS_LOB.APPEND(o_resume, v_education);
v_position:=v_position+DBMS_LOB.GETLENGTH(v_education);
DBMS_LOB.WRITE(o_resume, 12, v_position, 'Experience: ');
v_position:=v_position+12;
DBMS_LOB.APPEND(o_resume, v_work_experience);
v_position:=v_position+DBMS_LOB.GETLENGTH(v_work_experience);

EXCEPTION

```

```

        WHEN others THEN
            DBMS_OUTPUT.PUT_LINE('Problems while building the employee resume');
            RAISE;
    END GET_EMPLOYEE_RESUME;
    /

CREATE OR REPLACE PROCEDURE  ASSIGN_EMPLOYEE_TO_NEW_ACCOUNT (
    p_EmployeeId IN EMPLOYEES.emp_id%TYPE,
    p_DeptCode   IN ACCOUNTS.dept_code%TYPE,
    p_AcctId     IN ACCOUNTS.acct_id%TYPE) AS

    /*
    |-----
    | DESCRIPTION: Re-assigns employee to a new account
    |
    | DEMO PURPOSE: RAISE_APPLICATION_ERROR,
    |
    | EXAMPLE: EXEC ASSIGN_EMPLOYEE_TO_NEW_ACCOUNT(47, 'A01', 1)
    |-----
    */
    -- variable declaration
    v_CurrentEmployees NUMBER; -- Current number of employees assigned to account
    v_MaxEmployees     NUMBER; -- Maximum number of employees assigned to account

BEGIN

    SELECT current_employees, max_employees
       INTO v_CurrentEmployees, v_MaxEmployees
      FROM ACCOUNTS
     WHERE acct_id = p_AcctId
       AND dept_code = p_DeptCode;

    --Make sure there is enough room for this additional employee
    IF v_CurrentEmployees = v_MaxEmployees THEN
        RAISE_APPLICATION_ERROR(-20000, 'Can''t assign more employees to ' || p_DeptCode ||
        ' ' || p_AcctId);
    END IF;

    -- Add the employee to account
    UPDATE ACCOUNTS
       SET current_employees = current_employees-1
     WHERE acct_id=(SELECT acct_id
                    FROM EMPLOYEES
                   WHERE emp_id=p_EmployeeId);

    UPDATE EMPLOYEES
       SET acct_id = p_AcctId, dept_code = p_DeptCode
     WHERE emp_id=p_EmployeeId;

    UPDATE ACCOUNTS
       SET current_employees = current_employees+1

```

```

        WHERE acct_id=p_AcctId;

EXCEPTION
    WHEN NO_DATA_FOUND THEN
        --Account information passed to this procedure doesn't exist. Raise an error
        RAISE_APPLICATION_ERROR(-20001, p_DeptCode || ' ' || p_AcctId || ' doesn't
exist!');

END ASSIGN_EMPLOYEE_TO_NEW_ACCOUNT;
/

CREATE OR REPLACE PROCEDURE EMPLOYEE_DYNAMIC_QUERY (
    o_RefCur    OUT HELPER.RCT1,
    p_department1 IN EMPLOYEES.dept_code%TYPE DEFAULT NULL,
    p_department2 IN EMPLOYEES.dept_code%TYPE DEFAULT NULL) AS
/*
|-----
| DESCRIPTION: Search routine that returns the list of employees in the form of
|               reference cursor based on the input of department code.
|
| DEMO PURPOSE: Reference cursors, DBMS_SQL build-in package
|
| EXAMPLE: EXEC EMPLOYEE_DYNAMIC_QUERY(ref_cursor, 1, 2)
|-----
*/
-- variable declaration
v_CursorID    INTEGER;
v_SelectStmt  VARCHAR2(500);
v_FirstName   EMPLOYEES.first_name%TYPE;
v_LastName    EMPLOYEES.last_name%TYPE;
v_DeptCode    EMPLOYEES.dept_code%TYPE;
v_Dummy       INTEGER;

BEGIN

    -- Open the cursor for processing.
    v_CursorID := DBMS_SQL.OPEN_CURSOR;

    -- Create the query string.
    v_SelectStmt := 'SELECT first_name, last_name, dept_code
                    FROM EMPLOYEES
                    WHERE dept_code IN (:d1, :d2)
                    ORDER BY last_name';

    -- Parse the query.
    DBMS_SQL.PARSE(v_CursorID, v_SelectStmt, DBMS_SQL.NATIVE);

    -- Bind the input variables.
    DBMS_SQL.BIND_VARIABLE(v_CursorID, ':d1', p_department1);
    DBMS_SQL.BIND_VARIABLE(v_CursorID, ':d2', p_department2);

    -- Define the select list items.
    DBMS_SQL.DEFINE_COLUMN(v_CursorID, 1, v_FirstName, 20);

```

```

DBMS_SQL.DEFINE_COLUMN(v_CursorID, 2, v_LastName, 20);
DBMS_SQL.DEFINE_COLUMN(v_CursorID, 3, v_DeptCode, 30);

-- Execute the statement. We don't care about the return
-- value, but we do need to declare a variable for it.
v_Dummy := DBMS_SQL.EXECUTE(v_CursorID);

-- This is the fetch loop.
LOOP
    -- Fetch the rows into the buffer, and also check for the exit
    -- condition from the loop.
    v_Dummy:= DBMS_SQL.FETCH_ROWS(v_CursorID);
    IF v_Dummy = 0 THEN
        EXIT;
    END IF;

    -- Retrieve the rows from the buffer into PL/SQL variables.
    DBMS_SQL.COLUMN_VALUE(v_CursorID, 1, v_FirstName);
    DBMS_SQL.COLUMN_VALUE(v_CursorID, 2, v_LastName);
    DBMS_SQL.COLUMN_VALUE(v_CursorID, 3, v_DeptCode);

    -- Insert the fetched data into temp_table
    INSERT INTO TEMP_TABLE (char_col)
        VALUES (v_FirstName || ' ' || v_LastName || ' is a ' || v_DeptCode || '
department. ');

END LOOP;

-- Close the cursor.
DBMS_SQL.CLOSE_CURSOR(v_CursorID);
OPEN o_RefCur FOR SELECT char_col FROM TEMP_TABLE;

EXCEPTION
    WHEN OTHERS THEN
        -- Close the cursor, then raise the error again.
        DBMS_SQL.CLOSE_CURSOR(v_CursorID);
        RAISE;
END EMPLOYEE_DYNAMIC_QUERY ;
/

CREATE OR REPLACE DIRECTORY mydir AS 'C:\temp'
/

CREATE OR REPLACE PROCEDURE SAVE_ORG_STRUCT_TO_FILE IS
/*
||-----
|| DESCRIPTION: Stores the hierarchy of organization in the OS file
||
|| DEMO PURPOSE: UTL_FILE built-in package
||
|| EXAMPLE: EXEC SAVE_ORG_STRUCT_TO_FILE
||

```

```

||-----
*/
-- variable declaration
v_filehandle      UTL_FILE.FILE_TYPE;
v_filename        VARCHAR2(100) DEFAULT 'catalog.out';
v_temp_line       VARCHAR2(100);

BEGIN

    v_filehandle := UTL_FILE.FOPEN('MYDIR',v_filename,'w');
    IF (UTL_FILE.IS_OPEN( v_filehandle ) = FALSE ) THEN
        DBMS_OUTPUT.PUT_LINE('Cannot open file');
    END IF;
    FOR i IN (SELECT org_level, full_name, department
              FROM ORGANIZATION_STRUCTURE)
    LOOP
        UTL_FILE.PUT_LINE(v_filehandle, 'Level: ' || i.org_level || ' ' || i.full_name
|| ' Department: ' || i.department);
    END LOOP;
    UTL_FILE.FCLOSE(v_filehandle);

    EXCEPTION
        WHEN others THEN
            DBMS_OUTPUT.PUT_LINE('Some problem with saving organization structure to
file');

END SAVE_ORG_STRUCT_TO_FILE;
/

CREATE OR REPLACE PROCEDURE INSERT_CUSTOMER_IN_XML (cust_in IN VARCHAR2)
IS
/*
||-----
|| DESCRIPTION: This procedure selects the customer information stored in XML data
|| type and process it in a cursor loop.
||
|| DEMO PURPOSE: Procedure that utilizes the power of XML and Xquery to process XML
|| data. It demonstrates a comparison between DB2 and Oracle syntax.
||
|| EXAMPLE: EXEC Insert_Customer_in_XML
|| ('<customerinfo xmlns="http://posample.org" Cid="1000">
||          <name>Kathy Smith</name><addr country="Canada">
||          <street>5 Rosewood</street>
||          <city>Toronto</city><prov-state>Ontario</prov-state>
||          <pcode-zip>M6W 1E6</pcode-zip></addr>
||          <phone type="work">416-555-1358</phone></customerinfo>')
||-----
*/

v_cust_id PLS_INTEGER;
v_city VARCHAR2(50);

BEGIN

```

```

SELECT customer_sequence.nextval INTO v_cust_id FROM DUAL;
INSERT INTO CUSTOMERS VALUES (v_cust_id, XMLType(cust_in), SYSDATE);

SELECT extract(cust_details_xml,'//customer-details/addr/city/text()').getStringVal()
      INTO v_city
      FROM CUSTOMERS
      WHERE cust_id=v_cust_id;
DBMS_OUTPUT.PUT_LINE('Customers located in the same city: ');
FOR i IN (SELECT
extract(cust_details_xml,'//customer-details/name/text()').getStringVal() as cust_name
      FROM CUSTOMERS
      WHERE existsNode(cust_details_xml,'//customer-details') = 1
      AND
extract(cust_details_xml,'//customer-details/addr/city/text()').getStringVal()=v_city
      AND cust_id<>v_cust_id)
  LOOP
    DBMS_OUTPUT.PUT_LINE(i.cust_name);
  END LOOP;
EXCEPTION
  WHEN others THEN
    DBMS_OUTPUT.PUT_LINE('Problem with inserting XML data in customers table');
END Insert_Customer_in_XML;
/

CREATE OR REPLACE TRIGGER UPDATE_ACC_ON_NEW_EMPL
/*
||-----
|| DESCRIPTION: Trigger to update accounts and employees tables
||              upon addition of new employee
||
|| DEMO PURPOSE: Showcase PL/SQL support in triggers
||-----
||
*/
AFTER INSERT ON EMPLOYEES FOR EACH ROW
BEGIN
  -- Add one to the number of employees in the project.
  UPDATE ACCOUNTS
    SET current_employees = current_employees + 1
    WHERE dept_code = :new.dept_code
      AND acct_id = :new.acct_id;

END UPDATE_ACC_ON_NEW_EMPL;
/

```


E.1.3 Triggers and anonymous blocks

```
CREATE OR REPLACE TRIGGER UPDATE_DEPARTMENTS
/*
||-----
|| DESCRIPTION: Trigger to keep the entries in the managers, employees, and accounts
|| tables in sync. When a record is inserted
||
|| DEMO PURPOSE: Showcase PL/SQL support in triggers
||-----
*/
AFTER INSERT OR DELETE OR UPDATE ON employees FOR EACH ROW
BEGIN
    IF DELETING THEN
        UPDATE DEPARTMENTS
        SET total_projects=total_projects-:old.current_projects,
total_employees=total_employees-1
        WHERE dept_code=:old.dept_code;
    ELSIF INSERTING THEN
        UPDATE DEPARTMENTS
        SET total_projects=total_projects+:new.current_projects,
total_employees=total_employees+1
        WHERE dept_code=:new.dept_code;
    ELSIF UPDATING THEN
        UPDATE DEPARTMENTS
        SET total_projects=total_projects+:new.current_projects-:old.current_projects
        WHERE dept_code IN (:old.dept_code, :new.dept_code);
    END IF;

END UPDATE_DEPARTMENTS;
/

/*
||-----
|| DESCRIPTION: Anonymous blocks that simulates the applications run
||
|| DEMO PURPOSE: Showcase Anonymous blocks and output to console
||-----
*/

DECLARE
    v_emp_info EMP_INFO_TYPE;
    v_resume   CLOB;
    temp_string VARCHAR2(4000);
    o_RefCur  HELPER.RCT1;
BEGIN

    DBMS_OUTPUT.PUT_LINE('-----');
    DBMS_OUTPUT.PUT_LINE('----Account manipulation test--');
    ACCOUNT_PACKAGE.REMOVE_ACCOUNT(11, 'A00');
```

```

ACCOUNT_PACKAGE.ADD_ACCOUNT(11, 'A00', 'QUALITY PROGRAMS', 10);
ADD_NEW_EMPLOYEE('Max', 'Trenton', 2, 'A00', 1, v_emp_info);
ASSIGN_EMPLOYEE_TO_NEW_ACCOUNT(v_emp_info.emp_id, 'A00', 11);
ACCOUNT_PACKAGE.DISPLAY_ACCOUNT_LIST('A00', 11);

DBMS_OUTPUT.PUT_LINE('-----');
DBMS_OUTPUT.PUT_LINE('-----DBMS_LOB test-----');
GET_EMPLOYEE_RESUME(1, v_resume);
DBMS_OUTPUT.PUT_LINE(v_resume);

DBMS_OUTPUT.PUT_LINE('-----');
DBMS_OUTPUT.PUT_LINE('-----UTL_FILE test-----');
SAVE_ORG_STRUCT_TO_FILE();

DBMS_OUTPUT.PUT_LINE('-----');
DBMS_OUTPUT.PUT_LINE('--DBMS_SQL and Ref cursor test-');
EMPLOYEE_DYNAMIC_QUERY(o_RefCur, 'A00', 'B01');
FETCH o_RefCur INTO temp_string;
DBMS_OUTPUT.PUT_LINE(temp_string);

DBMS_OUTPUT.PUT_LINE('-----');
DBMS_OUTPUT.PUT_LINE('-----XML test-----');
Insert_Customer_in_XML ('<?xml version="1.0"><customer-details>
<name>Kathy Smith</name>
<addr country="Canada">
  <street>5 Rosewood</street>
  <city>Toronto</city>
  <prov-state>Ontario</prov-state>
  <pcode-zip>M6W 1E6</pcode-zip>
</addr>
<phone type="work">416-555-1358</phone>
</customer-details>');
Insert_Customer_in_XML ('<?xml version="1.0"><customer-details>
<name>John Edward</name>
<addr country="Canada">
  <street>15 Mintwood</street>
  <city>Toronto</city>
  <prov-state>Ontario</prov-state>
  <pcode-zip>L6A 4F2</pcode-zip>
</addr>
<phone type="work">416-112-2324</phone>
</customer-details>');
END;
/

```



```

"CURRENT_PROJECTS"  NUMBER(3),
"EMP_MGR_ID"        NUMBER(5),
"DEPT_CODE"         CHAR(3) NOT NULL,
"ACCT_ID"           NUMBER(3) NOT NULL,
"OFFICE_ID"         NUMBER(5),
"BAND"              CHAR(1),
"CREATE_DATE"       TIMESTAMP(3) DEFAULT SYSDATE)
;

CREATE table EMP_DETAILS (
  "EMP_ID"          NUMBER(5) NOT NULL,
  "EDUCATION"       CLOB NOT NULL,
  "WORK_EXPERIENCE" CLOB NOT NULL,
  "PHOTO_FORMAT"    VARCHAR2(10) NOT NULL,
  "PICTURE"         BLOB)
;

CREATE table OFFICES (
  "OFFICE_ID"       NUMBER(5) NOT NULL,
  "BUILDING"        VARCHAR2(25),
  "NUMBER_SEATS"    NUMBER(4),
  "DESCRIPTION"     VARCHAR2(50))
;

COMMENT ON TABLE DEPARTMENTS IS 'Contains information about departments and their
names';

COMMENT ON TABLE ACCOUNTS IS 'Contains information about accounts and number of
projects';

COMMENT ON TABLE CUSTOMERS IS 'Contains information about customers in XML format';

COMMENT ON TABLE EMPLOYEES IS 'Contains information about employees';

COMMENT ON TABLE EMP_DETAILS IS 'Contains additional information on employees like
biography and picture';

COMMENT ON TABLE OFFICES IS 'Contains information about buildigs and offices';

ALTER TABLE DEPARTMENTS ADD CONSTRAINT PK_DEPT_CODE PRIMARY KEY ( "DEPT_CODE" );

ALTER TABLE ACCOUNTS ADD CONSTRAINT PK_ACCOUNTS PRIMARY KEY ( "DEPT_CODE", "ACCT_ID" );

ALTER TABLE EMPLOYEES ADD CONSTRAINT PK_EMPLOYEES PRIMARY KEY ( "EMP_ID" );

ALTER TABLE EMP_DETAILS ADD CONSTRAINT PK_EMP_DETAILS PRIMARY KEY ( "EMP_ID" );

ALTER TABLE OFFICES ADD CONSTRAINT PK_OFFICES PRIMARY KEY ( "OFFICE_ID" );

ALTER TABLE ACCOUNTS ADD CONSTRAINT FK_ACC_DEPT_CODE FOREIGN KEY ( "DEPT_CODE" )
REFERENCES DEPARTMENTS ( "DEPT_CODE" );

```

```

ALTER TABLE EMPLOYEES ADD CONSTRAINT FK_EMP_MGR_ID FOREIGN KEY ( "EMP_MGR_ID" )
REFERENCES EMPLOYEES ( "EMP_ID" );

ALTER TABLE EMPLOYEES ADD CONSTRAINT FK_EMP_OFFICE_ID FOREIGN KEY ( "OFFICE_ID" )
REFERENCES OFFICES ( "OFFICE_ID" );

ALTER TABLE EMPLOYEES ADD CONSTRAINT M_DEPT_CODE_ACCT_ID FOREIGN KEY ( "DEPT_CODE",
"ACCT_ID" ) REFERENCES ACCOUNTS ( "DEPT_CODE", "ACCT_ID" );

ALTER TABLE EMP_DETAILS ADD CONSTRAINT FK_EMP_DETAILS_ID FOREIGN KEY ( "EMP_ID" )
REFERENCES EMPLOYEES ( "EMP_ID" ) ON DELETE CASCADE;

ALTER TABLE EMPLOYEES ADD CONSTRAINT BAND_VALIDATION CHECK (BAND IN ('1', '2', '3', '4',
'5'));

CREATE INDEX ACCT_DEPT_IND ON ACCOUNTS (DEPT_CODE);

CREATE INDEX EMP_SEARCH_IND ON EMPLOYEES(LAST_NAME, FIRST_NAME, DEPT_CODE);

CREATE INDEX CUSTOMER_CITY_IND ON "CUSTOMERS" (CUST_DETAILS_XML)
GENERATE KEY USING XMLPATTERN '/customer-details/addr/city' AS SQL VARCHAR(50);

CREATE USER TEMPORARY TABLESPACE USERTEMP;

CREATE GLOBAL TEMPORARY TABLE TEMP_TABLE (
"NUM_COL" NUMBER,
"CHAR_COL" VARCHAR2(60));

CREATE OR REPLACE VIEW ORGANIZATION_STRUCTURE ("ORG_LEVEL", "FULL_NAME", "DEPARTMENT")
AS
/*
||-----
|| DESCRIPTION: Dispaly hierarchy of people in the organization structure
||
||
|| DEMO PURPOSE: Support of recursive SQL "START WITH ... CONNECT BY" along with LEVEL
|| keyword.
||
|| New built-in functions INITCAP and NVL.
|| New syntax for outer join - (+)
||-----
*/
SELECT
    LEVEL as ORG_LEVEL,
    SUBSTR((LPAD(' ', 4 * LEVEL - 1) || INITCAP(e.last_name) || ', ' ||
INITCAP(e.first_name)), 1, 40),
    NVL(d.dept_name, 'Uknown')
FROM
    EMPLOYEES e,
    DEPARTMENTS d
WHERE
    e.dept_code=d.dept_code(+)
START WITH emp_id = 1
CONNECT BY NOCYCLE PRIOR emp_id = emp_mgr_id ;

```

```

CREATE PUBLIC SYNONYM EMPLOYEE_SEQUENCE FOR SEQUENCE SALES.EMPLOYEE_SEQUENCE;

-- public synonyms on tables
CREATE PUBLIC SYNONYM DEPARTMENTS FOR TABLE SALES.DEPARTMENTS ;
CREATE PUBLIC SYNONYM EMPLOYEES FOR TABLE SALES.EMPLOYEES ;
CREATE PUBLIC SYNONYM EMP_DETAILS TABLE FOR SALES.EMP_DETAILS;
CREATE PUBLIC SYNONYM OFFICES FOR TABLE SALES.OFFICES ;

--public synonyms on views
CREATE PUBLIC SYNONYM ORGANIZATION_STRUCTURE FOR TABLE SALES.ORGANIZATION_STRUCTURE;

```

E.2.2 Package, procedure, and function

```

CREATE OR REPLACE PACKAGE "HELPER" AS
/*
||-----
|| DESCRIPTION: the purpose of this package is to create types that can be used by
|| stand-alone procedures and functions
||
|| DEMO PURPOSE: Definition of package for creating new types, Reference Cursor,
|| Record data type, Variable arrays
||-----
*/

-- type declaration
TYPE RCT1 IS REF CURSOR;

TYPE EMP_INFO_TYPE IS RECORD (
    EMP_ID          NUMBER(5 , 0),
    FIRST_NAME      VARCHAR2(20),
    LAST_NAME       VARCHAR2(20),
    BAND            CHAR(1));

TYPE emp_array_type IS VARRAY(10) OF EMP_INFO_TYPE;

END HELPER;
@

CREATE OR REPLACE PACKAGE ACCOUNT_PACKAGE AS
/*
||-----
|| DESCRIPTION: Account package contains the procedures to manage the accounts.
|| using the procedures in this package, users can add an account, remove
|| an account, provide account information in form of associative array
|| and display the account information using server output.
||
|| DEMO PURPOSE: Package header declaration, anchor datatypes like %TYPE and %ROWTYPE,
|| definition of associative array TYPE ... IS TABLE OF ... INDEX BY ...
||-----
*/

```

```

*/

-- type declaration
TYPE customer_name_cache IS TABLE OF EMPLOYEES%ROWTYPE INDEX BY PLS_INTEGER;

-- PROCEDURE declaration
PROCEDURE Add_Account(p_AccountId IN ACCOUNTS.acct_id%TYPE,
                     p_DeptCode IN ACCOUNTS.dept_code%TYPE,
                     p_AccountDesc IN ACCOUNTS.acct_desc%TYPE,
                     p_MaxEmployees IN ACCOUNTS.max_employees%TYPE);

PROCEDURE Remove_Account(p_AccountId IN ACCOUNTS.acct_id%TYPE,
                        p_DeptCode IN ACCOUNTS.dept_code%TYPE);

PROCEDURE Account_List(p_dept_code IN ACCOUNTS.dept_code%TYPE,
                      p_acct_id IN ACCOUNTS.acct_id%TYPE,
                      p_Employees_Name_Cache OUT Customer_Name_Cache);

PROCEDURE Display_Account_List(p_dept_code IN ACCOUNTS.dept_code%TYPE,
                              p_acct_id IN ACCOUNTS.acct_id%TYPE);

END ACCOUNT_PACKAGE;
@

CREATE OR REPLACE PACKAGE BODY ACCOUNT_PACKAGE AS

--Nested function has been placed in the package body definition
FUNCTION AVERAGE_BAND ( p_Deptament IN EMPLOYEES.dept_code%TYPE,
                        p_ACCT_ID IN EMPLOYEES.acct_id%TYPE)

RETURN CHAR AS
/*
||-----
|| DESCRIPTION: Nested procedure to derive the average band of employees in the
|| department
||
|| DEMO PURPOSE: Cursor definition, cursor attributes %NOTFOUND and %ROWCOUNT,
||               DECODE built-in function, user defined exception
||-----
||
*/

-- variable declaration
v_AverageBAND CHAR(1);
v_NumericBand NUMBER;
v_TotalBand NUMBER:=0;
v_NumberEmployees NUMBER;

-- CURSOR declaration
CURSOR c_Employees IS
SELECT band
FROM EMPLOYEES
WHERE dept_code = p_Deptament
AND acct_id = p_ACCT_ID;

```

```

BEGIN
    OPEN c_Employees;
    LOOP
        FETCH c_Employees INTO v_NumericBand;
        EXIT WHEN c_Employees%NOTFOUND;
        v_TotalBand := v_TotalBand + v_NumericBand;
    END LOOP;

    v_NumberEmployees:=c_Employees%ROWCOUNT;
    IF(v_NumberEmployees = 0) THEN
        RAISE_APPLICATION_ERROR(-20001, 'No employees exist for ' || p_Department || ' '
|| p_ACCT_ID);
    END IF;

    SELECT DECODE(ROUND(v_TotalBand/v_NumberEmployees), 5, 'A', 4, 'B', 3, 'C', 2,
'D', 1, 'E')
    INTO v_AverageBand
    FROM DUAL;

    RETURN v_AverageBand;

END AVERAGE_BAND;

PROCEDURE ADD_ACCOUNT(p_AccountId    IN ACCOUNTS.acct_id%TYPE,
                    p_DeptCode      IN ACCOUNTS.dept_code%TYPE,
                    p_AccountDesc    IN ACCOUNTS.acct_desc%TYPE,
                    p_MaxEmployees  IN ACCOUNTS.max_employees%TYPE) IS
/*
|-----
| DESCRIPTION: Add a new Employee into the specified Account
|
| DEMO PURPOSE: predefined exceptions like DUP_VAL_ON_INDEX and OTHERS
|
| EXAMPLE: EXEC ACCOUNT_PACKAGE.ADD_ACCOUNT(1, 1, 'Description', 10)
|-----
*/
BEGIN

    INSERT INTO ACCOUNTS (acct_id, dept_code, acct_desc, max_employees,
current_employees, num_projects)
    VALUES (p_AccountId, p_DeptCode, p_AccountDesc, p_MaxEmployees, 0, 0);

    DBMS_OUTPUT.PUT_LINE('Account ' || p_AccountId || ' successfully created .');

    EXCEPTION
    WHEN dup_val_on_index THEN
        DBMS_OUTPUT.PUT_LINE('Duplicate Account was rejected');
    WHEN others THEN

```



```

        DBMS_OUTPUT.PUT_LINE('SQLCODE: ' || SQLCODE);
        DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
        RAISE;

END ADD_ACCOUNT;

PROCEDURE REMOVE_ACCOUNT(p_AccountId IN ACCOUNTS.acct_id%TYPE,
                        p_DeptCode IN ACCOUNTS.dept_code%TYPE) IS
/*
||-----
|| DESCRIPTION: Removes the Account from database based on the account id and
|| department code
||
|| DEMO PURPOSE: Exception declaration
||
|| EXAMPLE: EXEC ACCOUNT_PACKAGE.REMOVE_ACCOUNT(1, 1)
||-----
*/

-- exception declaration
e_AccountNotRegistered EXCEPTION;
PRAGMA EXCEPTION_INIT (e_AccountNotRegistered, -20050);

BEGIN
    DELETE FROM ACCOUNTS WHERE acct_id = p_AccountId AND dept_code = p_DeptCode;

    IF SQL%NOTFOUND THEN
        RAISE e_AccountNotRegistered;
    END IF;

    DBMS_OUTPUT.PUT_LINE('Account ' || p_AccountId || ' was successfully removed.');
```

EXCEPTION

```

    WHEN e_AccountNotRegistered THEN
        DBMS_OUTPUT.PUT_LINE('Account ' || p_AccountId || ' does not exist.');
```

WHEN others THEN

```

        RAISE;

END REMOVE_ACCOUNT;

PROCEDURE ACCOUNT_LIST(p_dept_code          IN ACCOUNTS.dept_code%TYPE,
                      p_acct_id            IN ACCOUNTS.acct_id%TYPE,
                      p_Employees_Name_Cache OUT CUSTOMER_NAME_CACHE) IS
/*
||-----
|| DESCRIPTION: Stores all employees from particular department in the associative
```

```

|| array
||
|| DEMO PURPOSE: Cursor definition, Cursor iteration through LOOP, population of
|| associative array
||-----
*/

-- variable declaration
    v_NumEmployees NUMBER := 1;

-- cursor declaration
    CURSOR c_RegisteredEmployees IS
    -- Local cursor to fetch the registered Employees.
    SELECT *
    FROM EMPLOYEES
    WHERE dept_code = p_dept_code
    AND acct_id = p_acct_id;

BEGIN
/*  p_NumEmployees will be the table index. It will start at 0,
    and be incremented each time through the fetch loop.
    At the end of the loop, it will have the number of rows
    fetched, and therefore the number of rows returned in p_IDs.  */

    OPEN c_RegisteredEmployees;

    LOOP
        FETCH c_RegisteredEmployees INTO p_Employees_Name_Cache(v_NumEmployees);
        EXIT WHEN c_RegisteredEmployees%NOTFOUND;
        v_NumEmployees := v_NumEmployees + 1;
    END LOOP;

    CLOSE c_RegisteredEmployees;

END ACCOUNT_LIST;

PROCEDURE DISPLAY_ACCOUNT_LIST(p_dept_code IN ACCOUNTS.dept_code%TYPE,
                              p_acct_id   IN ACCOUNTS.acct_id%TYPE) IS
/*
||-----
|| DESCRIPTION: Displays the information about Employees and number of their project
|| assigned to specific account.
|| DEMO PURPOSE: Usage of associative arrays, DBMS_OUTPUT built-in package,
||                call to the procedure in the same package
||
|| EXAMPLE: EXEC ACCOUNT_PACKAGE.DISPLAY_ACCOUNT_LIST('A00', 1);
||-----
||
*/
    -- variable declaration
    v_customer_name_cache CUSTOMER_NAME_CACHE;
    k                     NUMBER := 1;

```

```

BEGIN
    DBMS_OUTPUT.PUT_LINE('List of employees');
    DBMS_OUTPUT.PUT_LINE('-----');
    account_list(p_dept_code, p_acct_id, v_customer_name_cache);

    FOR k IN 1..v_customer_name_cache.COUNT LOOP
        DBMS_OUTPUT.PUT_LINE('Record id          : ' || k );
        DBMS_OUTPUT.PUT_LINE('Employee          : ' ||
v_customer_name_cache(k).last_name);
        DBMS_OUTPUT.PUT_LINE('Number of projects : ' ||
v_customer_name_cache(k).Current_Projects);
        DBMS_OUTPUT.PUT_LINE('Average Band in department : ' ||
average_band(v_customer_name_cache(k).dept_code, v_customer_name_cache(k).acct_id));
    END LOOP;

    END DISPLAY_ACCOUNT_LIST;

END ACCOUNT_PACKAGE;
@

CREATE OR REPLACE PROCEDURE COUNT_PROJECTS (p_empID IN employees.emp_ID%TYPE,
                                             o_acct_id OUT employees.acct_id%TYPE,
out_return OUT NUMBER ) IS
/*
||-----
|| DESCRIPTION: Function that counts the project based on the employee id and also
||               returns information on total projects of the account to which employee
||               id belongs
|| DEMO PURPOSE: Function with OUT parameter, FOR LOOP over cursor
||
|| EXAMPLE: SELECT COUNT_PROJECTS(1, acct_id) FROM DUAL;
||-----
|| */
-- variable declaration
v_TotalProjects NUMBER;
v_AccountProjects NUMBER;

-- CURSOR declaration
CURSOR c_DeptAccts IS
SELECT dept_code, acct_id
FROM EMPLOYEES
WHERE emp_id = p_empID;

BEGIN
    FOR v_AccountRec IN c_DeptAccts LOOP
        o_acct_id:=v_AccountRec.acct_id;
        -- Determine the projects for this account.
        SELECT num_projects
        INTO v_AccountProjects
        FROM ACCOUNTS

```

```

        WHERE dept_code = v_AccountRec.dept_code
        AND acct_id = v_AccountRec.acct_id;

        -- Add it to the total so far.
        v_Totalprojects := v_Totalprojects + v_AccountProjects;
    END LOOP;
    -- different line for DB2 and Oracle
    out_return:= v_Totalprojects;

END COUNT_PROJECTS;
@

CREATE OR REPLACE PROCEDURE ADD_NEW_EMPLOYEE (
    p_FirstName    EMPLOYEES.first_name%TYPE,
    p_LastName     EMPLOYEES.last_name%TYPE,
    p_EmpMgrId     EMPLOYEES.emp_mgr_id%TYPE,
    p_DeptCode     EMPLOYEES.dept_code%TYPE,
    p_Account      EMPLOYEES.acct_id%TYPE,
    o_Employee OUT HELPER.EMP_INFO_TYPE,
    p_CreateDate   EMPLOYEES.create_date%TYPE DEFAULT SYSDATE,
    p_OfficeId     EMPLOYEES.office_id%TYPE DEFAULT 2
) AS
/*
|-----
| DESCRIPTION: Procedure to add a new employee
|-----
| DEMO PURPOSE: Default values in the procedure definition, Regular loops,
|                sequence keywords like NEXTVAL and CURVAL
|                EXECUTE IMMEDIATE
|-----
| EXAMPLE: EXEC ADD_NEW_EMPLOYEE('Max', 'Trenton', 2, 1, 1, emp_info)
|-----
*/

-- variable declaration
v_EmployeeId     EMPLOYEES.emp_id%TYPE :=1;
v_EmployeeIdTemp EMPLOYEES.emp_id%TYPE;

-- cursor declaration
CURSOR c_CheckEmployeeId IS
    SELECT 1
    FROM EMPLOYEES
    WHERE emp_id=v_EmployeeId;

CURSOR c_get_employee IS
    SELECT emp_id, first_name, last_name, band
    FROM EMPLOYEES
    WHERE emp_id=v_EmployeeId;

BEGIN
    -- Find Next available employee id from the employee sequence

```

```

LOOP
    SELECT employee_sequence.NEXTVAL INTO v_EmployeeId FROM DUAL;
    OPEN c_CheckEmployeeId;
    FETCH c_CheckEmployeeId INTO v_EmployeeIdTemp;
    EXIT WHEN c_CheckEmployeeId%NOTFOUND;
    CLOSE c_CheckEmployeeId;
END LOOP;

select employee_sequence.CURRVAL INTO v_EmployeeId FROM DUAL;

EXECUTE IMMEDIATE 'INSERT INTO EMPLOYEES(emp_id, first_name, last_name,
current_projects, emp_mgr_id, dept_code, acct_id, office_id, band, create_date)
VALUES ( ' || v_EmployeeId || ', UPPER('' || p_FirstName || ''),
UPPER('' || p_LastName || ''),
0, ' || p_EmpMgrId || ', ' || p_DeptCode || ', ' || p_Account ||
', ' || p_OfficeId || ', 1, ' || p_CreateDate || ' )';

OPEN c_get_employee;
FETCH c_get_employee INTO o_Employee;
CLOSE c_get_employee;

DBMS_OUTPUT.PUT_LINE('Employee record id ' || v_EmployeeId || ' was created
successfully. ');
EXCEPTION
    WHEN others THEN
        DBMS_OUTPUT.PUT_LINE('Employee record was not created. ');
        RAISE;

END ADD_NEW_EMPLOYEE;
@

CREATE OR REPLACE PROCEDURE GET_EMPLOYEE_RESUME (p_empID IN employees.emp_ID%TYPE,
o_resume OUT CLOB) AS
/*
||-----
|| DESCRIPTION: Builds employee's resume in the CLOB format based on the employee id
||
|| DEMO PURPOSE: DBMS_LOB built-in package
||
|| EXAMPLE: EXEC GET_EMPLOYEE_RESUME(1, clob_resume)
||-----
||
*/
-- variable declaration
v_education CLOB;
v_work_experience CLOB;
v_picture BLOB;
v_position NUMBER:=1;
BEGIN
    o_resume:=EMPTY_CLOB();
    SELECT education, work_experience

```

```

        INTO v_education, v_work_experience
        FROM EMP_DETAILS
        WHERE emp_id=p_empID;
DBMS_LOB.WRITE_CLOB(o_resume, 7, v_position, 'Resume' || chr(10));
v_position:=v_position+7;
DBMS_LOB.WRITE_CLOB(o_resume, 11, v_position, 'Education: ');
v_position:=v_position+11;
DBMS_LOB.APPEND_CLOB(o_resume, v_education);
v_position:=v_position+DBMS_LOB.GETLENGTH(v_education);
DBMS_LOB.WRITE_CLOB(o_resume, 12, v_position, 'Experience: ');
v_position:=v_position+12;
DBMS_LOB.APPEND_CLOB(o_resume, v_work_experience);
v_position:=v_position+DBMS_LOB.GETLENGTH(v_work_experience);

EXCEPTION
    WHEN others THEN
        DBMS_OUTPUT.PUT_LINE('Problems while building the employee resume');
        RAISE;
END GET_EMPLOYEE_RESUME;
@

CREATE OR REPLACE PROCEDURE ASSIGN_EMPLOYEE_TO_NEW_ACCOUNT (
    p_EmployeeId IN EMPLOYEES.emp_id%TYPE,
    p_DeptCode   IN ACCOUNTS.dept_code%TYPE,
    p_AcctId     IN ACCOUNTS.acct_id%TYPE) AS

    /*
    |-----
    | DESCRIPTION: Re-assigns employee to a new account
    |
    | DEMO PURPOSE: RAISE_APPLICATION_ERROR,
    |
    | EXAMPLE: EXEC ASSIGN_EMPLOYEE_TO_NEW_ACCOUNT(47, 'A01', 1)
    |-----
    */
    -- variable declaration
    v_CurrentEmployees NUMBER; -- Current number of employees assigned to account
    v_MaxEmployees     NUMBER; -- Maximum number of employees assigned to account

BEGIN

    SELECT current_employees, max_employees
    INTO v_CurrentEmployees, v_MaxEmployees
    FROM ACCOUNTS
    WHERE acct_id = p_AcctId
    AND dept_code = p_DeptCode;

    --Make sure there is enough room for this additional employee
    IF v_CurrentEmployees = v_MaxEmployees THEN
        RAISE_APPLICATION_ERROR(-20000, 'Can''t assign more employees to ' || p_DeptCode ||
        ' ' || p_AcctId);

```

```

END IF;

-- Add the employee to account
UPDATE ACCOUNTS
  SET current_employees = current_employees-1
 WHERE acct_id=(SELECT acct_id
                  FROM EMPLOYEES
                  WHERE emp_id=p_EmployeeId);

UPDATE EMPLOYEES
  SET acct_id = p_AcctId, dept_code = p_DeptCode
 WHERE emp_id=p_EmployeeId;

UPDATE ACCOUNTS
  SET current_employees = current_employees+1
 WHERE acct_id=p_AcctId;

EXCEPTION
  WHEN NO_DATA_FOUND THEN
    --Account information passed to this procedure doesn't exist. Raise an error
    RAISE_APPLICATION_ERROR(-20001, p_DeptCode || ' ' || p_AcctId || ' doesn't
    exist!');

END ASSIGN_EMPLOYEE_TO_NEW_ACCOUNT;
@

CREATE OR REPLACE PROCEDURE EMPLOYEE_DYNAMIC_QUERY (
  o_RefCur   OUT HELPER.RCT1,
  p_department1 IN EMPLOYEES.dept_code%TYPE DEFAULT NULL,
  p_department2 IN EMPLOYEES.dept_code%TYPE DEFAULT NULL) AS
/*
|-----
| DESCRIPTION: Search routine that returns the list of employees in the form of
|               reference cursor based on the input of department code.
|
| DEMO PURPOSE: Reference cursors, DBMS_SQL build-in package
|
| EXAMPLE: EXEC EMPLOYEE_DYNAMIC_QUERY(ref_cursor, 1, 2)
|-----
*/
-- variable declaration
v_CursorID   INTEGER;
v_SelectStmt VARCHAR2(500);
v_FirstName  EMPLOYEES.first_name%TYPE;
v_LastName   EMPLOYEES.last_name%TYPE;
v_DeptCode   EMPLOYEES.dept_code%TYPE;
v_Dummy      INTEGER;

BEGIN

  -- Open the cursor for processing.
  v_CursorID := DBMS_SQL.OPEN_CURSOR;

```

```

-- Create the query string.
v_SelectStmt := 'SELECT first_name, last_name, dept_code
                FROM EMPLOYEES
                WHERE dept_code IN (:d1, :d2)
                ORDER BY last_name';

-- Parse the query.
DBMS_SQL.PARSE(v_CursorID, v_SelectStmt, DBMS_SQL.NATIVE);

-- Bind the input variables.
DBMS_SQL.BIND_VARIABLE_CHAR(v_CursorID, ':d1', p_department1);
DBMS_SQL.BIND_VARIABLE_CHAR(v_CursorID, ':d2', p_department2);

-- Define the select list items.
DBMS_SQL.DEFINE_COLUMN_VARCHAR(v_CursorID, 1, v_FirstName, 20);
DBMS_SQL.DEFINE_COLUMN_VARCHAR(v_CursorID, 2, v_LastName, 20);
DBMS_SQL.DEFINE_COLUMN_CHAR(v_CursorID, 3, v_DeptCode, 30);

-- Execute the statement. We don't care about the return
-- value, but we do need to declare a variable for it.
v_Dummy := DBMS_SQL.EXECUTE(v_CursorID);

-- This is the fetch loop.
LOOP
    -- Fetch the rows into the buffer, and also check for the exit
    -- condition from the loop.
    v_Dummy:= DBMS_SQL.FETCH_ROWS(v_CursorID);
    IF v_Dummy = 0 THEN
        EXIT;
    END IF;

    -- Retrieve the rows from the buffer into PL/SQL variables.
    DBMS_SQL.COLUMN_VALUE_VARCHAR(v_CursorID, 1, v_FirstName);
    DBMS_SQL.COLUMN_VALUE_VARCHAR(v_CursorID, 2, v_LastName);
    DBMS_SQL.COLUMN_VALUE_CHAR(v_CursorID, 3, v_DeptCode);

    -- Insert the fetched data into temp_table
    INSERT INTO TEMP_TABLE (char_col)
    VALUES (v_FirstName || ' ' || v_LastName || ' is a ' || v_DeptCode || '
department.');
```

```

    END LOOP;

-- Close the cursor.
DBMS_SQL.CLOSE_CURSOR(v_CursorID);
OPEN o_RefCur FOR SELECT char_col FROM TEMP_TABLE;

EXCEPTION
    WHEN OTHERS THEN
        -- Close the cursor, then raise the error again.
        DBMS_SQL.CLOSE_CURSOR(v_CursorID);
        RAISE;
END EMPLOYEE_DYNAMIC_QUERY ;

```


@

```
CREATE OR REPLACE PROCEDURE SAVE_ORG_STRUCT_TO_FILE IS
/*
||-----
|| DESCRIPTION: Stores the hierarchy of organization in the OS file
||
|| DEMO PURPOSE: UTL_FILE built-in package
||
|| EXAMPLE: EXEC SAVE_ORG_STRUCT_TO_FILE
||-----
*/
-- variable declaration
v_filehandle      UTL_FILE.FILE_TYPE;
v_filename        VARCHAR2(100) DEFAULT 'catalog.out';
v_temp_line       VARCHAR2(100);

BEGIN
    UTL_DIR.CREATE_DIRECTORY('MYDIR', '/tmp');
    v_filehandle := UTL_FILE.FOPEN('MYDIR',v_filename,'w');
    IF (UTL_FILE.IS_OPEN( v_filehandle ) = FALSE ) THEN
        DBMS_OUTPUT.PUT_LINE('Cannot open file');
    END IF;
    FOR i IN (SELECT org_level, full_name, department
              FROM ORGANIZATION_STRUCTURE)
    LOOP
        UTL_FILE.PUT_LINE(v_filehandle, 'Level: ' || i.org_level || ' ' || i.full_name
|| ' Department: ' || i.department);
    END LOOP;
    UTL_FILE.FCLOSE(v_filehandle);
    UTL_DIR.DROP_DIRECTORY('MYDIR');

    EXCEPTION
        WHEN others THEN
            DBMS_OUTPUT.PUT_LINE('Some problem with saving organization structure to
file');

END SAVE_ORG_STRUCT_TO_FILE;
@
```

```
CREATE OR REPLACE PROCEDURE INSERT_CUSTOMER_IN_XML (cust_in IN VARCHAR2)
IS
/*
||-----
|| DESCRIPTION: This procedure selects the customer information stored in XML data
|| type and process it in a cursor loop.
||
|| DEMO PURPOSE: Procedure that utilizes the power of XML and Xquery to process XML
|| data. It demonstrates a comparison between DB2 and Oracle syntax.
||-----
*/
```

```

||
|| EXAMPLE: EXEC Insert_Customer_in_XML
|| ('<customerinfo xmlns="http://posample.org" Cid="1000">
||         <name>Kathy Smith</name><addr country="Canada">
||         <street>5 Rosewood</street>
||         <city>Toronto</city><prov-state>Ontario</prov-state>
||         <pcode-zip>M6W 1E6</pcode-zip></addr>
||         <phone type="work">416-555-1358</phone></customerinfo>')
|| -----
*/

v_cust_id PLS_INTEGER;
v_city VARCHAR2(50);

BEGIN

SELECT customer_sequence.nextval INTO v_cust_id FROM DUAL;
INSERT INTO CUSTOMERS VALUES (v_cust_id, cust_in, SYSDATE);

SELECT XMLCAST(XMLQUERY('$customer/customer-details/addr/city' PASSING
cust_details_xml AS "customer") AS VARCHAR(50))
INTO v_city
FROM CUSTOMERS
WHERE cust_id=v_cust_id;

DBMS_OUTPUT.PUT_LINE('Customers located in the same city: ');
FOR i IN (SELECT XMLCAST(XMLQUERY('$customer/customer-details/addr/city' PASSING
cust_details_xml AS "customer") AS VARCHAR(50)) AS cust_name
FROM CUSTOMERS
WHERE XMLCAST(XMLQUERY('$customer/customer-details/addr/city' PASSING
cust_details_xml AS "customer") AS VARCHAR(50))=v_city
AND cust_id<>v_cust_id)
LOOP
DBMS_OUTPUT.PUT_LINE(i.cust_name);
END LOOP;
EXCEPTION
WHEN others THEN
DBMS_OUTPUT.PUT_LINE('Problem with inserting XML data in customers table');
END Insert_Customer_in_XML;
@

```

E.2.3 Trigger

```

CREATE OR REPLACE TRIGGER UPDATE_ACC_ON_NEW_EMPL
/*
|| -----
|| DESCRIPTION: Trigger to update accounts and employees tables
||              upon addition of new employee
||
|| DEMO PURPOSE: Showcase PL/SQL support in triggers
|| -----
||

```

```

        */
        AFTER INSERT ON EMPLOYEES FOR EACH ROW
        BEGIN
            -- Add one to the number of employees in the project.
            UPDATE ACCOUNTS
                SET current_employees = current_employees + 1
                WHERE dept_code = :new.dept_code
                AND acct_id = :new.acct_id;

        END UPDATE_ACC_ON_NEW_EMPL;

    @

CREATE OR REPLACE TRIGGER UPDATE_DEPARTMENTS
/*
||-----
|| DESCRIPTION: Trigger to keep the entries in the managers, employees, and accounts
|| tables in sync. When a record is inserted
||
|| DEMO PURPOSE: Showcase PL/SQL support in triggers
||-----
*/
    AFTER INSERT ON employees FOR EACH ROW
    BEGIN
        UPDATE DEPARTMENTS
            SET total_projects=total_projects+:new.current_projects,
            total_employees=total_employees+1
            WHERE dept_code=:new.dept_code;
    END UPDATE_DEPARTMENTS;
    @

CREATE OR REPLACE TRIGGER UPDATE_DEPARTMENTS
/*
||-----
|| DESCRIPTION: Trigger to keep the entries in the managers, employees, and accounts
|| tables in sync. When a record is inserted
||
|| DEMO PURPOSE: Showcase PL/SQL support in triggers
||-----
*/
    AFTER DELETE ON employees FOR EACH ROW
    BEGIN
        UPDATE DEPARTMENTS
            SET total_projects=total_projects-:old.current_projects,
            total_employees=total_employees-1
            WHERE dept_code=:old.dept_code;

    END UPDATE_DEPARTMENTS;
    @

CREATE OR REPLACE TRIGGER UPDATE_DEPARTMENTS

```

```

/*
||-----
|| DESCRIPTION: Trigger to keep the entries in the managers, employees, and accounts
|| tables in sync. When a record is inserted
||
|| DEMO PURPOSE: Showcase PL/SQL support in triggers
||-----
*/
AFTER UPDATE ON employees FOR EACH ROW
BEGIN
    UPDATE DEPARTMENTS
        SET total_projects=total_projects+:new.current_projects-:old.current_projects
        WHERE dept_code IN (:old.dept_code, :new.dept_code);

END UPDATE_DEPARTMENTS;
@

```



Additional material

This book refers to additional material that can be downloaded from the Internet as described below.

Locating the Web material

The Web material associated with this book is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser at:

<ftp://www.redbooks.ibm.com/redbooks/SG247736>

Alternatively, you can go to the IBM Redbooks Web site at:

ibm.com/redbooks

Select the **Additional materials** and open the directory that corresponds with the IBM Redbooks form number, SG247736.

Using the Web material

The additional Web material that accompanies this book includes the following files:

<i>File name</i>	<i>Description</i>
sg247636testcase.zip	Zipped test code samples for enablement practice. The zip file includes the following: Oracle_testcase DB2_testcase

System requirements for downloading the Web material

The following system configuration is recommended:

Hard disk space:	0.5 MB minimum
Operating System:	Windows 2000/Linux SUSE or Red Hat
Processor:	Intel® 386 or higher
Memory:	16 MB

How to use the Web material

Create a subdirectory (folder) on your workstation, and unzip the contents of the Web material zip file into this folder.

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this book.

IBM Redbooks

For information about ordering these publications, see “How to get Redbooks” on page 322. Note that some of the documents referenced here may be available in softcopy only.

- ▶ *Oracle to DB2 Conversion Guide for Linux, UNIX, and Windows*, SG24-7048

Other publications

These publications are also relevant as further information sources:

IBM DB2 for Linux, UNIX, and Windows manuals

- ▶ *Administrative API Reference*, SC27-2435
- ▶ *Administrative Routines and Views*, SC27-2436
- ▶ *Call Level Interface Guide and Reference, Volume 1*, SC27-2437
- ▶ *Call Level Interface Guide and Reference, Volume 2*, SC27-2438
- ▶ *Command Reference*, SC27-2439
- ▶ *Data Movement Utilities Guide and Reference*, SC27-2440
- ▶ *Data Recovery and High Availability Guide and Reference*, SC27-2441
- ▶ *Database Administration Concepts and Configuration Reference*, SC27-2442
- ▶ *Database Monitoring Guide and Reference*, SC27-2458
- ▶ *Database Security Guide*, SC27-2443
- ▶ *DB2 Connect User's Guide*, SC27-2434
- ▶ *DB2 Text Search Guide*, SC27-2459
- ▶ *Developing ADO.NET and OLE DB Applications*, SC27-2444
- ▶ *Developing Embedded SQL Applications*, SC27-2445

- ▶ *Developing Java Applications*, SC27-2446
- ▶ *Developing Perl, PHP, Python, and Ruby on Rails Applications*, SC27-2447
- ▶ *Developing User-defined Routines (SQL and External)*, SC27-2448
- ▶ *Getting Started with Database Application Development*, GI11-9410
- ▶ *Getting Started with DB2 Installation and Administration on Linux and Windows*, GI11-9411
- ▶ *Globalization Guide*, SC27-2449
- ▶ *Information Integration: Administration Guide for Federated Systems*, SC19-1020
- ▶ *Information Integration: ASNCLP Program Reference for Replication and Event Publishing*, SC19-1018
- ▶ *Information Integration: Configuration Guide for Federated Data Sources*, SC19-1034
- ▶ *Information Integration: Introduction to Replication and Event Publishing*, SC19-1028
- ▶ *Information Integration: SQL Replication Guide and Reference*, SC19-1030
- ▶ *Installing and Configuring DB2 Connect Personal Edition*, SC27-2432
- ▶ *Installing and Configuring DB2 Connect Servers*, SC27-2433
- ▶ *Installing DB2 Servers*, GC27-2455
- ▶ *Installing IBM Data Server Clients*, GC27-2454
- ▶ *Message Reference Volume 1*, SC27-2450
- ▶ *Message Reference Volume 2*, SC27-2451
- ▶ *Net Search Extender Administration and User's Guide*, SC27-2469
- ▶ *SQL Procedural Languages: Application Enablement and Support*, SC23-9838
- ▶ *Partitioning and Clustering Guide*, SC27-2453
- ▶ *pureXML Guide*, SC27-2465
- ▶ *Query Patroller Administration and User's Guide*, SC27-2467
- ▶ *Spatial Extender and Geodetic Data Management Feature User's Guide and Reference*, SC27-2468
- ▶ *SQL Procedural Language Guide*, SC27-2470
- ▶ *SQL Reference, Volume 1*, SC27-2456
- ▶ *SQL Reference, Volume 2*, SC27-2457
- ▶ *Troubleshooting and Tuning Database Performance*, SC27-2461

- ▶ *Upgrading to DB2 Version 9.7*, SC27-2452
- ▶ *Visual Explain Tutorial*, SC27-2462
- ▶ *What's New for DB2 Version 9.7*, SC27-2463
- ▶ *Workload Manager Guide and Reference*, SC27-2464
- ▶ *XQuery Reference*, SC27-2466

Online resources

These Web sites are also relevant as further information sources:

DB2

- ▶ Database and Information Management home page:
<http://www.ibm.com/software/data/>
- ▶ DB2 developerWorks:
<http://www.ibm.com/developerworks/db2/>
- ▶ DB2 Information Center:
<http://publib.boulder.ibm.com/infocenter/db2luw/v9r5/>
- ▶ DB2 for Linux:
<http://www.ibm.com/software/data/db2/linux/>
<http://www.ibm.com/software/data/db2/linux/validate/>
- ▶ DB2 Product Family Library:
<http://www.ibm.com/software/data/db2/library/>
- ▶ DB2 Technical Support:
http://www.ibm.com/software/data/db2/support/db2_9/
- ▶ DB2 Universal Database V9 Application Development:
<http://www.ibm.com/software/data/db2/ad/>
- ▶ Planet DB2:
<http://www.planetdb2.com/>

Other resources

- ▶ Apache HTTP Server Project:
<http://httpd.apache.org>

- ▶ Comprehensive Perl Archive Network:
<http://www.cpan.org>
http://www.cpan.org/modules/by-category/07_Database_Interfaces/DBI
- ▶ DB2 Perl Database Interface:
<http://www.ibm.com/software/data/db2/perl>
- ▶ DBI.perl.org:
<http://dbi.perl.org>
- ▶ IBM Tivoli System Automation for Multiplatforms:
http://publib.boulder.ibm.com/tividd/td/ITSAFL/SC33-8272-01/en_US/PDF/HALBAU01.pdf
- ▶ Perl.apache.org:
<http://perl.apache.org/docs/1.0/guide/>
- ▶ PHP scripting language:
<http://www.php.net/>

How to get Redbooks

You can search for, view, or download Redbooks, Redpapers, Technotes, draft publications and Additional materials, as well as order hardcopy Redbooks publications, at this Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Index

A

- access_denied 264
- analyze_database procedure 262
- anonymous block 28, 39, 43, 47, 51, 64, 74
- anonymous PL/SQL block 41
- API 174, 176–177, 180–181, 196–197, 200–201, 211
- application design 151
- application types 5
- array element 39
- array type 30, 87
- assignment statement 38–39, 75
- associative array 30, 32, 88
- associative array index 30
- associative array type 87
- atomic 41
- auto_reval 160
- automatic revalidation 161
- automatic storage 162
- autonomous transaction 80, 94
- availability 180–181

B

- background thread 154
- backup 150
- bigint 193
- bind_variable_blob procedure 258
- bind_variable_char procedure 258
- bind_variable_clob procedure 258
- bind_variable_date procedure 258
- bind_variable_double procedure 258
- bind_variable_int procedure 258
- bind_variable_number procedure 258
- bind_variable_raw procedure 258
- bind_variable_timestamp procedure 258
- bind_variable_varchar procedure 259
- binding 209
- bitmap index 106
- bldmapp.bat 199
- bldrtn 82, 84
- Boolean 43, 47
- buffer pool 162
- buffer_size 196

- built-in 191
- built-in data type 55
- built-in package
 - dbms_alert 15
 - dbms_job 15
 - dbms_lob 15
 - dbms_output 15
 - dbms_pipe 15
 - dbms_sql 15
 - dbms_utility 15
 - utl_file 15
 - utl_mail 15
 - utl_smtp 15
- built-in packages 21

C

- C 173, 192
- C layer 202
- C++ 192
- call statement 64, 137
- case expression 45, 47
- case statement 45–47, 71, 111
- charsetmismatch 264
- close_cursor procedure 259
- cobol 178–179, 191
- collection method 27–28, 32
- column_value_blob procedure 259
- column_value_char procedure 259
- column_value_clob procedure 259
- column_value_date procedure 259
- column_value_double procedure 259
- column_value_int procedure 259
- column_value_long procedure 259
- column_value_number procedure 259
- comma_to_table procedure 262
- command window console 164
- command-line interface 145
- command-line mode 144
- compatibility 180–181, 209
- compatibility feature 60
- compatibility mode 150
- compatible mode 27
- compile_schema procedure 262

- concat 65
- configuration parameter setting 162
- connect by 24, 80, 94–95, 111
- connection 182, 191, 200–202, 204, 209, 211, 214, 218, 220–224, 229, 231
- console option 145
- constraint 107–108, 143
- create_pipe function 253
- current_date 65
- current_timestamp 65
- cursor 39, 49, 55–57, 59, 64, 76, 88, 101
- cursor factory 56
- cursor stability 10
- cursor type 87

D

- data access 151
- data clustering 162
- data definition language 60
- data manipulation language 60
- data movement 147
- data organization scheme 121
- data partitioning 115
- data type 24–25, 27, 30, 33, 35–36, 39, 55, 65, 69, 87, 102, 112
- data types 21
- database enablement 21
- data-centric 151
- date 24, 27, 38, 68, 75, 92, 102, 113, 115, 121, 141–142
- DB_version procedure 262
- DB2 Call Interface 209
- DB2_COMPATIBILITY_VECTOR 23, 160
- DB2_DEFERRED_PREPARE_SEMANTICS 160
- DB2CI 209–210
- dbms_alert 89
- dbms_alert module 251
- dbms_output module 250
- dbms_pipe module 252
- dbms_sql module 257
- dbms_utility module 261
- DDL 23, 60, 102, 139, 164–165
- decfloat 26
- decflt_rounding 160
- declared variable 36, 38
- decode function 71–72
- deferred_force 160
- define_column_blob procedure 259

- define_column_clob procedure 259
- define_column_double procedure 259
- define_column_long procedure 260
- define_column_raw procedure 260
- define_column_timestamp procedure 260
- delete_failed 264
- deploy wizard 154
- deployment 5, 151
- deployment log 149
- deployment operation 149
- descriptor function 209
- development 151
- disable procedure 250
- distinct type 87, 98
- DML 60, 62, 139
- dot notation 33, 36
- driver manager 174, 176, 202, 211
- dynamic SQL execution 41, 60, 63

E

- education 180, 182
- EJB 201
- enable procedure 250
- error functions 64
- etNumberVal() 183
- exception 39, 41, 43, 50, 52–53, 55, 67, 70, 75, 88, 102
- EXEC SQL 178–179, 192–198
- exec_ddl_statement procedure 262
- executable statement 40
- execute immediate 38, 40–41, 60–62, 64, 81, 105
- execute procedure 260
- execute_and_fetch procedure 260
- exit statement 47
- exit when condition 47, 49
- export file 82
- expression 38–39, 45, 47–48, 75, 104, 106, 117
- extended support 27, 102–103
- extentsize 115
- external procedure 82
- extract 165
- extraction process 146

F

- federated database 125
- fetch_rows procedure 260
- file I/O 209
- file systems 162

file_open 264
fixed-length 193
floating point 26
for loop 49, 58
fremove procedure 264
function 26, 39, 47, 65, 67, 70–74, 78, 80–85, 88,
94, 103, 106, 110, 132, 148

G

generator 181
get_cpu_time function 262
get_lines procedure 250
getClobVal() 183
getConnection method 202, 204
getNamespace() 183
getStringVal() 183

H

handle function 209
hash memory 162
hash partitioning 162
hierarchical query 95
host variable 178–179, 192–194, 197–198

I

IBMDDataMovementTool.cmd 145
IBMDDataMovementTool.jar 145
IBMDDataMovementTool.sh 145
if statement 44
implicit casting 42
implicit cursor 58
include column index 105
index 143, 158
index value 30
initcap 42, 65, 76, 79, 95, 111
inout 197–198
input parameter 74
input string 64
interactive deploy 149
interactive mode 144
internal_error 264
invalid_filehandle 265
invalid_filename 265
invalid_maxlinesize 265
invalid_mode 265
invalid_offset 265
invalid_operation 265

invalid_path 265
is_open procedure 260

J

J2SE 200
Java layer 202
Java ServerPage 201
javac 84–85
JDBC OCI driver 202–204

L

large object 209, 256
last_row_count procedure 260
lname_array 263
lock memory 162
logic element 87
logical unit numbers 162
LPAD 65, 111
LUN 162

M

MAIL procedure 268
materialized query tables 162
materialized views 162
mechanism 195
message_buffer 196
module 25, 86, 89
multi-dimensional clustering 162

N

name_resolve procedure 262
named set 87
namespace 87
new_line procedure 250, 264
next value expression 104
next_item_type function 253
noop procedure 268
null statement 40
number 24, 27, 38, 54, 66, 68, 80–81, 102, 109,
136, 140–142
nvl function 72

O

object definition 87
OCI 173, 202, 204, 209–210, 217
ODBC 173–174, 176, 182, 210–211, 218, 220,
222, 225–226

open database connectivity 210–211
open_connection function 269
open_connection procedure 269
open_cursor procedure 260
open_data procedure 269
Oracle Call Interface 209, 217
outer join 99

P

pack_message function 253
pack_message_raw procedure 253
package 25, 27–28, 35, 52, 56, 60, 78–79, 86–89, 93, 110–111, 136, 148
package body 35, 79, 88
package specification 87–88
packages and language elements 21
parameter marker 198, 215
parameterized cursor 59
parse procedure 260
performance 151
PL/SQL language 39
placeholder 40
precompiler 177–178, 191–193, 195, 211
predetermined number 49
prepare 25, 41
primary key 107
Pro*C 209
procedure 148
pseudo-column 96–97
public element 88
published object 87
published prototype 87
purge procedure 253
put procedure 250, 264
put_line procedure 250, 264

Q

quit procedure 269

R

RAID 162
raise statement 53
raise_application_error procedure 53
range partitioning 115–116, 122, 162
raw 27
raw device 162
RawToHex 72

RCPT procedure 269
read_error 265
receive_message function 253
record variable 35
recursion 80, 94
Redbooks Web site 322
 Contact us xiv
redundant array of independent disk 162
REF 55, 58, 77–78
registry 162
registry variable 23
remove_pipe function 253
removeall procedure 251
rename_failed 265
repository 181, 187
reset_buffer procedure 253
result set 49, 55–57, 71, 74, 136, 138, 209
return clause 57
returning into clause 41–42
returning into values 41
reverse keyword 49
roll-out 182
rollout 211
routine 80, 82, 87, 103
routine prototype 87
row compression 5
row type 34, 87
RPAD 65, 130
rset procedure 269
runtime 177, 196

S

scalar function 68, 94
scalar functions 21
schema evolution 103
searched case statement 47
self-tuning memory manager 162
send_attach_raw procedure 267
send_message procedure 253
sequence 25, 47, 76, 104–105, 110
server authentication 163
servers edition 2
servlet 201
set serveroutput on 29, 98, 250
set_defaults procedure 251
signal procedure 251
silent mode 158
simple case statement 45

- sort memory 162
- spserver.sqc 82
- SQL communication area 196
- SQL operation 40
- SQL PL routine 151
- SQL statement 41, 43, 60, 63, 74, 84, 93, 95–96, 100, 111, 130, 135
- SQL*NET 202
- SQL/XML 182–183, 185–188
- SQL%FOUND 34, 37, 43
- SQL%NOTFOUND 43, 52, 56, 77–78
- SQL%ROWCOUNT 43–44
- SQLCA 195–196, 198
- SQLCODE function 70
- SQLDA 198
- SQLERRM function 70
- sqlint32 192–194
- sqlint64 193
- SQLj 200
- sqlquery 186
- static cursor 60
- STMM 162
- stored procedure 75, 82–83, 94
- stripe 162
- striping 162
- synonym 23, 96, 110
- syntax 181, 185, 198, 202, 208–209, 211, 214

T

- table 143
- table_to_comma procedure 262
- target database 165–166
- temporary table space 161
- test database 151
- time zone 65, 67
- to_char 65
- to_date 65
- to_number 65
- to_timestamp 65
- transaction control 60, 75, 209
- transaction log 162
- tree view 149
- trigger 39, 74, 90, 92, 94, 148
- truncate statement 24, 93, 100, 109
- type construct 78

U

- UDF 84–86

- uncl_array 263
- Unicode 161
- unique_session_name function 253
- unpack_message procedure 253
- update statement 41
- user-defined record type 35
- userenv function 73
- utilemb.h 199
- utilemb.sqc 199
- utl_dir 89
- utl_file package 264
- utl_mail 89
- utl_mail module 267
- utl_smtp module 268

V

- validate procedure 262
- varchar2 24, 27–28, 30–31, 34–36, 38, 40, 42, 44, 46, 56, 58–59, 61–64, 66–67, 72, 74, 77–81, 98, 102, 109, 112, 130, 140–142
- variable 178–179, 192–194, 197–198, 222–224
- variable assignment 39
- variable_value_blob procedure 260
- variable_value_char procedure 260
- variable_value_clob procedure 260
- variable_value_date procedure 260
- variable_value_double procedure 260
- varray 27–28, 32
- version 150
- vrfy procedure 269

W

- waitany procedure 251
- weakly typed cursor 56
- web service 151
- while loop statement 48
- while statement 47
- wildcard 185
- working directory 149
- workloads 5
- write_data procedure 269
- write_error 265
- write_raw_data procedure 269

X

- XML 182–186, 188–189, 191, 194
- XMLAGG 184

- XMLAgg 183
- XMLAttribute 183–184
- XMLCAST 184, 186
- XMLCast 183–184
- xmlcolumn 186–187, 189
- XMLCONCAT 184
- XMLConcat 183
- XMLELEMENT 184
- XMLElement 183
- XMLEXISTS 184–185, 188
- XMLFOREST 185
- XMLForest 183
- XMLPARSE 184
- XMLParse 183
- XMLQUERY 184–185, 187, 191
- XMLSERIALIZE 184
- XMLSerialize 183
- XMLTABLE 184, 188
- XMLType 183, 186
- XQuery 151
- XQuery statement 104



Oracle to DB2 Conversion Guide: Compatibility Made Easy

(0.5" spine)
0.475" <-> 0.875"
250 <-> 459 pages



Oracle to DB2 Conversion Guide: Compatibility Made Easy

**Move Oracle to DB2
efficiently and
effectively**

**Learn about DB2 9.7
Oracle Database
compatibility
features**

**Use Oracle PL/SQL
skills directly with
DB2 9.7**

In this IBM Redbooks publication, we describe the new DB2 Oracle Database compatibility features. The latest version of DB2 includes extensive native support for the PL/SQL procedural language, new data types, scalar functions, improved concurrency, built-in packages, OCI, SQL*Plus, and more. These features dramatically improve the ease of developing applications that run on both DB2 and Oracle, and simplify the process of moving from Oracle to DB2.

In addition, IBM now provides rich tooling to simplify the enablement process, such as the highly scalable IBM Data Movement Tool for moving schema and data into DB2, and an Editor and Profiler for PL/SQL provided by way of the Optim Data Studio tool suite.

This Oracle to DB2 migration guide describes new technology, best practices for moving to DB2, and how to handle some common scenarios.

Within six weeks of DB2 9.7 becoming generally available, scores of companies had already reported very high compatibility between the code they developed for Oracle and what they were experiencing with DB2 9.7. These companies reported out-of-the-box compatibility rates between 90%-100% (measured by the percentage of statements that required modification), with typical compatibility in the high 90s. These statistics come from dozens of early adopters who shared their code and their experiences with IBM.

INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks