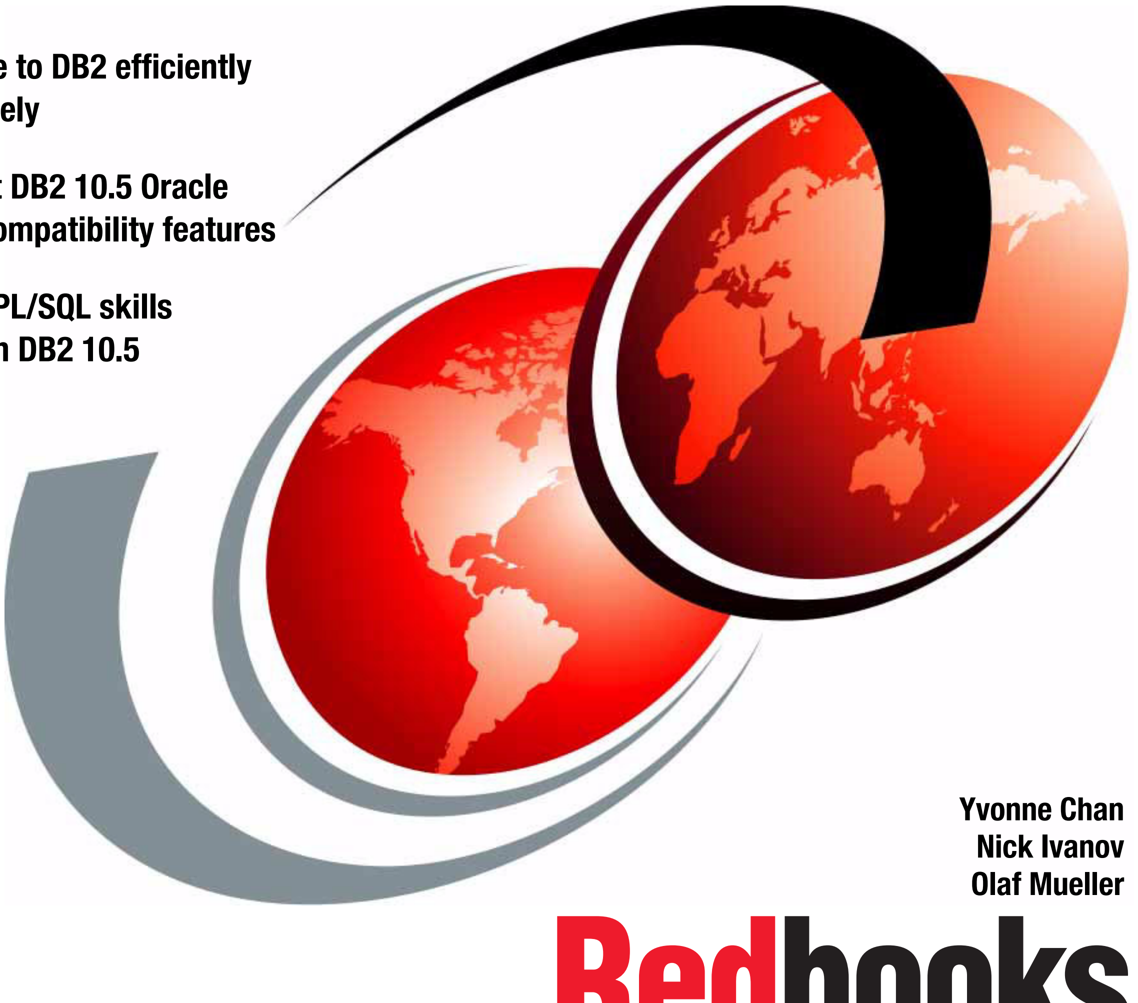
IBM® Information Management Software

Oracle to DB2 Conversion Guide: Compatibility Made Easy

Move Oracle to DB2 efficiently and effectively

Learn about DB2 10.5 Oracle Database compatibility features

Use Oracle PL/SQL skills directly with DB2 10.5



Yvonne Chan
Nick Ivanov
Olaf Mueller

Redbooks

ibm.com/redbooks

International Technical Support Organization

Oracle to DB2 Conversion Guide: Compatibility Made Easy

September 2013

SG24-7736-02

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

Third Edition (September 2013)

This edition applies to IBM DB2 for Linux, UNIX, and Windows Version 10.5.

Contents

Notices ix
Trademarks x

Preface xi
Authors xii
  Acknowledgements xiii
Now you can become a published author, too! xiv
Comments welcome xiv
Stay connected to IBM Redbooks publications xv

Summary of changes xvii
September 2013, Third Edition xvii

Chapter 1. Introduction 1
1.1 DB2 family of products 2
  1.1.1 DB2 editions 2
  1.1.2 IBM DB2 10.5 Advanced Enterprise Edition features 6
  1.1.3 DB2 10 autonomic computing features 7
  1.1.4 Introduction to PureData 9
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 Oracle Call Interface and Pro*C 16
1.3 DB2 educational resources 16
  1.3.1 IBM professional certification 18
  1.3.2 Other resources 18
  1.3.3 DB2 10 videos and topics 19

Chapter 2. Language compatibility features 21
2.1 DB2 compatibility features references 22
  2.1.1 SQL compatibility setup 22
  2.1.2 PL/SQL record and collection types 27
  2.1.3 Subtypes 45

2.1.4 Basic procedural statements 47
2.1.5 Control of flow statements. 53
2.1.6 Condition (exceptions) handling 59
2.1.7 Cursor data type 63
2.1.8 Static and dynamic SQL support. 69
2.1.9 Support for built-in scalar functions. 73
2.1.10 Routines, procedures, and functions compatibility 84
2.1.11 PL/SQL packages 96
2.1.12 Triggers 100
2.1.13 SQL statements 103
2.2 Schema compatibility features 113
2.2.1 Extended data type support 113
2.2.2 Flexible schema changes in DB2 114
2.2.3 Sequences 115
2.2.4 Index enablement 117
2.2.5 Constraints enablement 118
2.2.6 Created global temporary tables 120
2.2.7 Synonyms 121
2.2.8 Views and Materialized Views. 122
2.2.9 Object types 123
2.2.10 Partitioning and MDC 126
2.2.11 Oracle database links 137
2.2.12 Oracle Data Dictionary compatible views 140
2.3 DB2 command-line utilities 145
2.3.1 The command line processor plus user interface 145
2.3.2 Using the DB2 command line processor. 153

Chapter 3. Conversion process and enablement tools 159
3.1 The conversion process 160
3.2 Enablement tools. 162
3.2.1 IBM Data Studio 162
3.2.2 IBM Database Conversion Workbench 165
3.3 Getting started with DCW 169
3.3.1 Creating a DCW project 169
3.3.2 DCW Task Launcher. 171
3.4 DDL extraction using DCW 171
3.4.1 DDL extraction using a connection 172
3.4.2 DDL extraction using a custom script 177
3.4.3 Importing the DDL file 178
3.5 Assessment and conversion using DCW 178
3.5.1 Evaluating an Oracle DDL. 179
3.5.2 Converting Oracle DDL to DB2 compatible syntax 185
3.5.3 The Split DDL function 189

3.6 Preparing your DB2 database for data movement 190
3.6.1 Creating the target DB2 database. 191
3.6.2 Deploying the DDL objects that are required for data movement . . 191
3.7 Data movement using DCW 194
3.7.1 Data movement using flat files 194
3.7.2 Data movement using pipes 197
3.7.3 Data movement using IBM InfoSphere Federation Server 200
3.7.4 Data movement using IBM InfoSphere Change Data Capture 203
3.7.5 Manual data deployment. 209
3.7.6 Selecting the appropriate data movement method 210
3.7.7 Verifying data movement 211
3.8 Deploying remaining objects on the target DB2 database 211
3.9 Conclusion. 212

Chapter 4. Enablement scenario 213
4.1 Installing DB2 and creating an instance 214
4.2 Enabling SQL compatibility 217
4.3 Creating and configuring the target DB2 database 217
4.4 Defining a new database user. 220
4.5 Using IBM Database Conversion Workbench 220
4.5.1 Getting started. 220
4.5.2 Extracting DDL and PL/SQL objects 223
4.5.3 Compatibility evaluation 225
4.5.4 Conversion 228
4.5.5 Splitting DDL 230
4.5.6 Deploying objects 231
4.5.7 Extracting and loading data from files 234
4.5.8 Deploying PL/SQL objects 237
4.5.9 Resolving incompatibilities with Interactive Deploy. 238
4.6 Verifying enablement. 240
4.7 Summary 241

Chapter 5. Application conversion 243
5.1 DB2 application development introduction 244
5.1.1 Driver support 244
5.1.2 Embedded SQL. 246
5.2 Application enablement planning 249
5.2.1 Checking software and hardware availability and compatibility . . . 249
5.2.2 Educating developers and administrators. 250
5.2.3 Analyzing application logic and source code 250
5.2.4 Setting up the target environment. 250
5.2.5 Changing vendor-specific database API use 250
5.2.6 Application testing. 251

5.2.7 Application tuning 252
5.2.8 Production rollout procedures 252
5.2.9 User education 252
5.3 Converting XML features. 252
5.3.1 SQL/XML. 253
5.3.2 XQuery 256
5.3.3 Modifying XML data 259
5.4 Converting Oracle Pro*C applications to DB2. 261
5.4.1 Connecting to the database 261
5.4.2 Host variable declaration. 262
5.4.3 Exception handling 265
5.4.4 Error messages and warnings 266
5.4.5 Passing data to a stored procedure from a C program 267
5.4.6 Building a C/C++ DB2 application. 269
5.5 Converting Oracle Java applications to DB2. 270
5.5.1 Java access methods to DB2 271
5.5.2 JDBC driver for DB2 271
5.5.3 JDBC driver declaration 272
5.5.4 New binary XML API 274
5.5.5 Stored procedure calls 275
5.6 Converting Oracle Call Interface applications 279
5.7 Converting Open Database Connectivity applications 282
5.7.1 Introduction to DB2 CLI. 282
5.7.2 Setting up the DB2 CLI environment. 283
5.8 Converting Perl applications 284
5.8.1 DB2 Connect method syntax 286
5.8.2 Syntax for calling a DB2 stored procedures 286
5.9 Converting PHP applications. 288
5.9.1 Connecting to Oracle using PDO 289
5.9.2 Connecting to DB2 using PDO 289
5.9.3 Connecting to an Oracle database using PHP (OCI8) 290
5.9.4 Connecting PHP applications to a DB2 database. 291
5.10 Converting .NET applications 296
5.10.1 Supported development software for .NET Framework applications (DB2 9.7) 296
5.10.2 Supported development software for .NET Framework applications (DB2 10.5) 297
5.10.3 Supported deployment software for .NET Framework applications (in general). 297
5.10.4 .NET Data Providers 297
5.10.5 Visual Basic .NET conversion example 298

Appendix A. Terminology mapping 305

Appendix B. Data types 309
B.1 Supported SQL data types in C/C++ 310
B.2 Supported SQL data types in Java 314
B.3 Data types available in PL/SQL 316
B.4 Mapping Oracle data types to DB2 data types 319

Appendix C. Built-in modules 321
C.1 DBMS_ALERT 322
C.2 DBMS_DDL 323
C.3 DBMS_JOB 324
C.4 DBMS_LOB 326
C.5 DBMS_OUTPUT 328
C.6 DBMS_PIPE 329
C.7 DBMS_SQL 331
C.8 DBMS_UTILITY 335
C.9 UTL_DIR 338
C.10 UTL_MAIL 339
C.11 UTL_SMTP 341

Appendix D. DB2CI sample program 345

Appendix E. Code samples 351
E.1 Code disclaimer 352
E.2 Oracle DDL statements 352
E.2.1 Tables and views 352
E.2.2 Packages, procedures, and functions 357
E.2.3 Triggers and anonymous blocks 370
E.3 DB2 DDL statements 373
E.3.1 Tables and views 373
E.3.2 PL/SQL packages, procedures, and functions 378
E.3.3 Triggers 394
E.4 Deep nested objects sample 395
E.4.1 Creating a database 396
E.4.2 Creating schema objects (default schema) 396
E.4.3 JUnit tests (class file excerpt that includes only relevant information for the printed book) 413
E.4.4 Cleaning up 424

Appendix F. Additional material 425
Code disclaimer 425
Locating the web material 426
Using the web material 426
System requirements for downloading the web material 426
Downloading and extracting the web material 427

Related publications 429
IBM Redbooks publications 429
Other publications 429
Online resources 431
Help from IBM 432

Notices

Trademarks

Preface

This IBM® Redbooks® publication describes IBM DB2® SQL 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 can help with developing applications that run on both DB2 and Oracle and can help simplify the process of moving from Oracle to DB2.

In addition, IBM now provides tools 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 the IBM Data Studio tool suite.

This Oracle to DB2 migration guide describes new technology, preferred practices for moving to DB2, and common scenarios that can help you as you move from Oracle to DB2. This book is intended for IT architects and developers who are converting from Oracle to DB2.

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

Authors

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

Yvonne Chan is the DB2 IBM pureScale® Principal on the IBM PureData™ Ecosystem team helping customers implement DB2 pureScale and IBM PureData System for Transactions solutions. Before her current role, she spent several years on the DB2 pureScale kernel team, and led the DB2 Linux Team that is responsible for porting DB2 to several Linux platforms, including x86_64, IBM System z®, and Itanium Yvonne has been with IBM for over 14 years and earned a Bachelor of Applied Science degree from the University of Toronto in Computer Engineering

Nick Ivanov is a DB2 Solutions Migration Consultant at the IBM Toronto Software Laboratory. Before joining IBM Canada Labs in 2010, Nick Ivanov worked for many years as an independent DBA consultant for companies in financial and retail industries and the public sector. Nick has experience working with DB2, Oracle, MS SQL Server, and other database systems.

Olaf Mueller is the WW Principal of DB2 Conversion in the IBM Information Management Technical Enablement organization. He is based at the Toronto Lab in Canada. With his more than 20 years of experience, he helps major IBM customers convert their existing Oracle -based applications to DB2. Olaf is also the chief architect of the IBM Database Conversion Workbench. This workbench helps automate the whole database conversion process. Olaf is an experienced IBM Redbooks publications author. He holds a degree in Chemistry from Johannes-Gutenberg-Universitaet at Mainz, Germany.

Acknowledgements

The authors would like to express their great appreciation to the following contributing authors:

Sabyasachi Routray is a senior software developer at IBM India Software Labs. After starting his career with IBM in 2005, Sabyasachi has been involved in software testing and development across different projects. Sabyasachi has experience in Core Java, Realtime Java, Perl, XML, and DB2.

Guilherme Gevaerd is a Software Engineer with the IBM Information Management PureData Ecosystem organization. He is based at the São Paulo Laboratory in Brazil. He has over five years of experience working on software development. He is a developer of IBM Database Conversion Workbench. This workbench helps automate the whole database conversion process. He holds a degree in Computer Science from Univerdade do Estado de Santa Catarina at Joinville, Brazil.

Tapas Gupta is a software developer in the IBM India Software Labs. He has more than six years of experience in software development. Tapas has worked on various telecom and banking domain projects. Tapas has experience in Java, C++, Perl, Oracle, and DB2.

The authors also thank the following people for their contributions to this project:

Joshua Kim
Project Manager, IBM Information Management

Esteban Laver
Software Engineer, IBM Information Management

Robert Matchett
Senior Software Engineer, IBM Information Management

Jana Palmer
Application Architect, IBM Information Management

Fraser McArthur
Technical Enablement Specialist, IBM Information Management

Whei-Jen Chen
Project Leader, IBM International Technical Support Organization

Thanks to the authors of the previous editions of this book.

- The authors of the second edition, *Oracle to DB2 Conversion Guide: Compatibility Made Easy*, published in September 2012, were:

  Nick Ivanov
  Romeo Lupascu
  Olaf Mueller

Now you can become a published author, too!

Comments welcome

- Mail your comments to:

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

Stay connected to IBM Redbooks publications

- Find us on Facebook:

  `http://www.facebook.com/IBMRedbooks`
- Follow us on Twitter:

  `http://twitter.com/ibmredbooks`
- Look for us on LinkedIn:

  `http://www.linkedin.com/groups?home=&gid=2130806`
- Explore new Redbooks publications, residencies, and workshops with the IBM Redbooks weekly newsletter:

  `https://www.redbooks.ibm.com/Redbooks.nsf/subscribe?OpenForm`
- Stay current on recent Redbooks publications with RSS Feeds:

  `http://www.redbooks.ibm.com/rss.html`

Summary of changes

This section describes the technical changes that were made in this edition of the book and in previous editions. This edition might also include minor corrections and editorial changes that are not identified.

Summary of Changes
for SG24-7736-02
for Oracle to DB2 Conversion Guide: Compatibility Made Easy
as created or updated on July 2, 2014.

September 2013, Third Edition

This revision reflects the addition, deletion, or modification of new and changed information described below.

New information

- New product packaging for DB2 10.5 (Chapter 1)
- New compatibility features (Chapters 1 and 2)
- IBM Database Conversion Workbench (Chapter 3)

Changed information

- Improvements to compatibility features (Chapter 2)
- Description of the latest version of enablement tools (Chapter 3)
- Enablement scenario modified to use updated enablement tools (Chapter 4)

1 Introduction

IBM DB2 10.5 for Linux, UNIX, and Windows operating systems offers extensive native support for Oracle compatibility, including native support for Oracle SQL and PL/SQL dialects. This support allows many applications that are written against Oracle to run against DB2 with minimal or no changes, allowing you to easily migrate your applications to DB2.

In this book, we explore the preferences and practices for migrating your application from Oracle to DB2.

Terminology note: In this book, references to *DB2 UDB* or *DB2* are used generically to mean *IBM DB2 10.5 for Linux, UNIX, and Windows.*

This chapter includes the following topics:

- DB2 family of products (including an introduction to PureData)
- DB2 Oracle database compatibility features overview
- DB2 educational resources

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 to provide customers with a robust and scalable enterprise-wide solution.

1.1.1 DB2 editions

IBM offers database solutions that run on all platforms, including Microsoft Windows, IBM AIX®, Solaris, HP-UX, Linux, IBM AS/400®, IBM OS/390®, and IBM z/OS® operating systems. In addition, DB2 technologies support both 32-bit and 64-bit environments, providing support for 32-bit operating systems on Linux on x86 and Windows systems, and 64-bit operating systems on Linux, UNIX, and Windows systems.

The DB2 product family is composed of various editions and features that provide customers with choices based on business need. With DB2 10.5, to simplify the decision about which edition to buy, the number of available choices has been reduced to seven editions and one feature.

Express-C

IBM DB2 Express®-C is available as a no cost, entry-level edition of the DB2 data server for developers and the IBM Business Partner community. It can be up and running in minutes and includes self-management features. It includes the following capabilities of DB2 for Linux, UNIX, and Windows operating systems:

- IBM pureXML® storage
- Oracle Compatibility
- Replication tools
- Time Travel Query
- IBM Data Studio
- Federation with DB2 and IBM Informix Data Server

Solutions that are developed using DB2 Express-C can be seamlessly deployed using scalable DB2 editions without modifications to the application code.

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 number of processors and memory and is optimized to use up to a maximum of two processor cores and 16 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.

Express Edition

DB2 Express is a full-function DB2 data server, which provides attractive entry-level pricing for the small and medium business (SMB) market. It is offered in per Authorized User, Processor 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. Although it is easy to upgrade to the other editions of DB2 10.5, 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. DB2 Express adds the following additional features to the Express-C edition:

- pureXML
- Web services federation
- DB2 Homogeneous Federation
- High Availability and Disaster Recovery (HADR)
- Homogeneous SQL replication between DB2 for Linux, UNIX, and Windows, and Informix
- IBM Tivoli® System Automation

DB2 Express can be deployed on pervasive SMB operating systems, such as Linux, Windows, or Solaris systems. If licensed as a yearly subscription (DB2 Express FTL), it also includes a high availability feature if both primary and secondary servers in the high availability cluster are licensed. If licensed under the Limited Use Virtual Server metric, DB2 Express uses up to eight cores on the server. The DB2 data server cannot use more than 64 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.

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, Processor 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. Included with this edition is the IBM Data Studio tool, which can be installed separately.

DB2 Workgroup can be deployed in Linux, UNIX, and Windows server environments and uses up to 16 cores and 128 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.

Enterprise Server Edition

DB2 Enterprise Server Edition 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 Server Edition is an ideal foundation for building on demand enterprise-wide solutions, such as high-performing 24x7 available high-volume transaction processing business solutions or web-based solutions. It is the data server back-end system of choice for the following types of industry-leading ISVs building enterprise solutions:

- Business intelligence
- Content management
- E-commerce
- Enterprise Resource Planning
- Customer Relationship Management
- Supply Chain Management

Additionally, DB2 Enterprise Server Edition offers connectivity, compatibility, and integration with other enterprise DB2 and IBM Informix® data sources. DB2 Enterprise Server Edition includes the following additional features:

- Materialized Query Table (MQT)
- Multi-Dimensional Clustering (MDC)
- Multi-Temperature Storage
- Query Parallelism

DB2 Enterprise Server Edition 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.

Advanced Enterprise Server Edition

DB2 Advanced Enterprise Server Edition is the flagship of DB2 editions. It is designed to meet the data server needs of 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 Advanced Enterprise Server Edition is an ideal foundation for building on demand enterprise-wide solutions, such as high-performing 24x7 available high-volume transaction processing business solutions or web-based solutions. Additionally, DB2 Advanced Enterprise Server Edition offers connectivity, compatibility, and integration with other enterprise DB2 and Informix data sources. It provides the following additional features.

- DB2 Storage Optimization Feature
- Continuous Data Ingest
- Federation with DB2 for Linux, UNIX, and Windows, and Oracle
- IBM InfoSphere® Optim™ Configuration Manager
- IBM InfoSphere Optim Performance Manager Extended
- IBM InfoSphere Optim Query Workload Tuner
- IBM InfoSphere Data Architect
- Q Replication with two other DB2 for Linux, UNIX, and Windows servers
- Workload management

DB2 Advanced Enterprise Server Edition 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.

Advanced Workgroup Server Edition

This edition is similar to the DB2 Advanced Enterprise Server Edition except that it has limits on processor, memory, and database size and is suitable for deployment in a departmental, workgroup, or medium-sized business environment.

DB2 Advanced Workgroup Server Edition is available on either a Processor Value Unit or per Authorized User Single Install 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. DB2 Advanced Workgroup Server Edition can be deployed in Linux, UNIX, and Windows server environments with up to 16 cores and 128 GB of memory.

For more information about DB2 database product editions, go to the following website:

`http://pic.dhe.ibm.com/infocenter/db2luw/v10r5/index.jsp`

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 all the DB2 server editions and DB2 Connect Enterprise Edition so you can build solutions that use the latest data server technologies.

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

1.1.2 IBM DB2 10.5 Advanced Enterprise Edition features

Although DB2 10 includes capabilities that can serve the needs of most typical 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 as different product editions to provide the flexibility to purchase only what you need.

The following features are included with the DB2 10.5 Advanced Workgroup Edition, DB2 10.5 Advanced Enterprise Edition, and DB2 10.5 Developer Edition.

IBM DB2 column organized tables

DB2 10.5 introduces a new processing paradigm and data format that can accelerate analytic processing. Compressed column-organized tables for DB2 databases provide broad support of data mart workloads that involve complex queries that are characterized by multi-table joins, grouping, and aggregation and table scans over a star schema.

The IBM DB2 pureScale feature

You can reduce the risk and cost that is associated with growing your distributed database solution by providing extreme capacity and application transparency. Designed for continuous availability and high availability capable of exceeding even the strictest industry standard, the IBM DB2 pureScale feature tolerates both planned maintenance and component failure with ease.

The DB2 pureScale feature was first introduced in DB2 9.8. DB2 10.5 builds on DB2 pureScale feature support by adding the highly available and disaster recovery feature (HADR), increasing the availability characteristics, and improving workload balancing. In addition, restore mobility is provided between the DB2 pureScale feature and the DB2 Enterprise Server Edition.

The IBM DB2 Advanced Recovery Feature

This is the only feature that can be additionally purchased with any DB2 10.5 edition except for the Express-C edition. This feature provides advanced database backup, recovery, and data extraction capabilities through the following DB2 tools:

- IBM DB2 Merge Backup for Linux, UNIX, and Windows
- IBM DB2 Recovery Expert for Linux, UNIX, and Windows
- IBM DB2 Optim High Performance Unload for DB2 for Linux, UNIX, and Windows

1.1.3 DB2 10 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.

DB2 autonomic computing includes the following features:

- Automatic features

  Automatic features assist you in managing a 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 change 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
  - 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 places your data. Then, the database manager manages 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 that is 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 to ensure 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 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 you must 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).

1.1.4 Introduction to PureData

IBM PureData Systems are optimized to deliver data to today's demanding applications. With built-in expertise, integration by design, and simplified experience in mind, these systems provide a well-tuned, working database environment for workloads that expect little to no downtime in a 24x7 environment.

One of these systems is the PureData System for Transactions, which includes a DB2 pureScale cluster allowing for a continuously available DB2 database environment. This platform helps reduce complexity, accelerates the time to value, and helps lower ongoing data management costs for any OLTP workload.

For more information, go to the following website:

http://www-01.ibm.com/software/data/puredata/transactions/

1.2 DB2 Oracle database compatibility features overview

To allow an application that is written for one relational database management system (RDBMS) to run on another RDBMS unchanged, many pieces must fall into place. Different locking mechanisms, data types, SQL, procedural language on the server, and even the client interfaces that are used by the application itself must be aligned in syntax and in semantics.

Starting with Version 9.7, DB2 understands and runs applications that are written for Oracle. In DB2 10.5, additional Oracle compatibility features are introduced.

1.2.1 Concurrency control

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

There is no semantic reason why a transaction that runs 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 is implemented in DB2 so that DB2 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 is not committed yet. If the row is written out and is also overwritten in the log buffer, DB2 knows exactly where to find it so that a single I/O retrieves the wanted version.

Figure 1-1 on page 11 shows an application that updates a name in an employee table. Before that application commits the change, another application scans that table. Traditionally, the second user waits for the first application to commit or roll back. Because it is reading currently committed data, the scan for the second application retrieves the version of the row from the log buffer that does not contain the first user's changes.

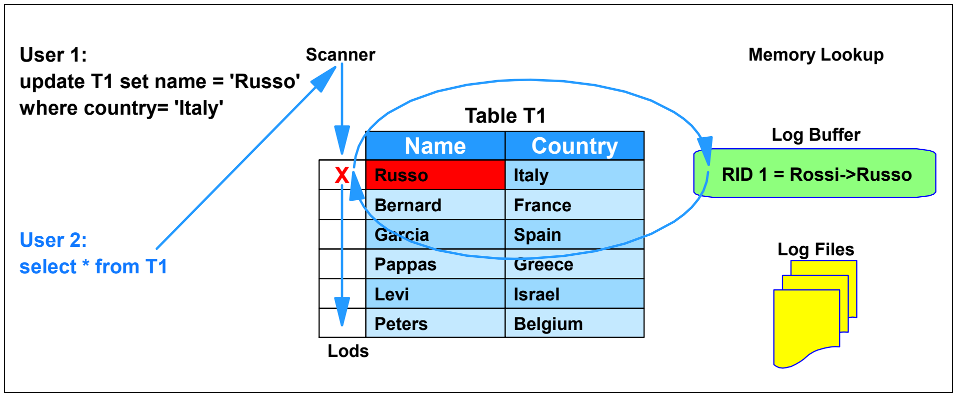


Figure 1-1 Concurrency control

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

This new behavior cannot cause a situation similar to “Snapshot too old” because in the unlikely event that the necessary log file is archived, DB2 falls back and waits for the lock to go away. It is a rare occurrence for this situation to occur, as it requires archival of a log file while a transaction is still uncommitted.

In addition to these changes, additional lock avoidance techniques were introduced in to 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 your ability to enable an application to another RDBMS. To allow Oracle applications to run on DB2, you must 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 PL/SQL that are added in DB2. (For more information, see B.3, “Data types available in PL/SQL”.)

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.

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

In addition, untyped NULLs can be used in many more places, and untyped parameter markers can be used nearly anywhere, thanks to deferred prepare, where DB2 does not resolve the type of a parameter marker until it sees the first actual value.

DB2 also supports defaulting procedure parameters and 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 started 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 implements many nonstandard keywords and semantics. DB2 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 introduced native PL/SQL support. Figure 1-2 on page 13 shows that the DB2 engine now includes a PL/SQL compiler with the SQL PL compiler. Both compilers produce virtual machine code for DB2 SQL Unified Runtime Engine. It is important to note that monitoring and development tools such as Optim Database Tools are hooked into DB2 at the runtime engine level. DBAs and application programmers develop and debug their PL/SQL source.

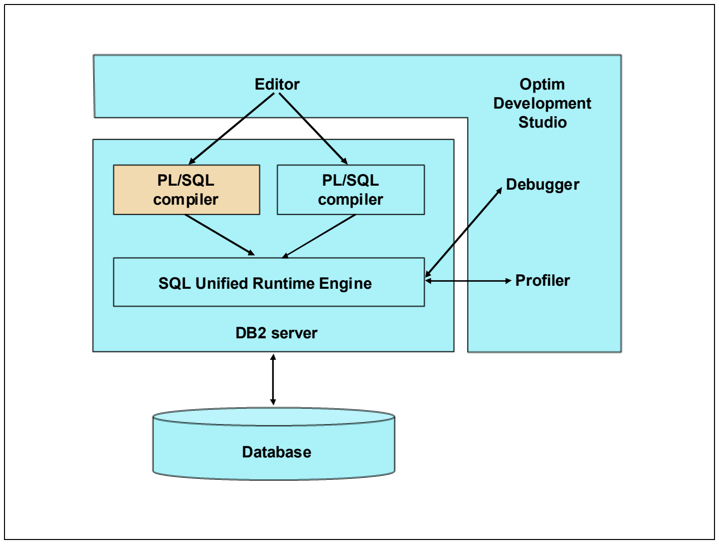


Figure 1-2 SQL compiler

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 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 the DB2 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 IBM Data Studio.

PL/SQL syntax details

DB2 supports the following common constructs of PL/SQL:

- 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 run 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 objects 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 and 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 and non-procedural objects, such as variables and types.
- CREATE [OR REPLACE] PACKAGE BODY, which implements all private and public routines and 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 use packages that are provided by the RDBMS. In particular, libraries that provide reporting, email, 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
- DBMS_DDL
- UTL_FILE
- UTL_MAIL
- UTL_SMTP
- UTL_DIR

1.2.7 Oracle specific JDBC extensions

JDBC is a standard Java client interface. However, extensions were added to the Oracle JDBC driver to support specific nonstandard data types. To maximize the level of compatibility for Java technology-based applications, the DB2 10.5 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 reports are written using the SQL*Plus command line processor. To make it easier to transfer these scripts and the skills of developers that write them, DB2 provides an SQL*Plus compatible command line processor, called *CLPPlus*. The tool has the following functionality:

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

1.2.9 Oracle Call Interface and Pro*C

Oracle Call Interface (OCI) and Pro*C are Oracle client APIs that allow access to an Oracle database from C and C++ programs.

DB2CI is a callable SQL interface to the DB2 database servers. It is a C and C++ API for DB2 database access that uses function calls to connect to databases, manage cursors, and perform SQL statements.

The DB2CI interface provides support for over 120 OCI APIs. This support reduces the complexity of enabling existing OCI applications so that they work with DB2 databases. The IBM Data Server Driver for DB2CI is the driver for the DB2CI interface.

DB2 Embedded SQL is the equivalent to Oracle Pro*C. It is extended to understand the Oracle Pro*C dialect as well.

1.3 DB2 educational resources

IBM has offerings to support training needs, enhance skills, and boost success with IBM software. IBM offers a range of training options from traditional classroom to instructor-led online (ILO) training to meet your demanding schedule. ILO training 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 onsite training for groups as small as three or as large as 14. Choose from the same quality training that is 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 website or with your training representative for the most recent training schedule.

Table 1-1 lists the DB2 educational offerings that are available.

Table 1-1 DB2 educational offerings

Course title	Classroom Code
DB2 Family Fundamentals	CE031
SQL Workshop	CE121
SQL Workshop for Experienced Users	1E131
Fast Path to DB2 10.1 for Experienced Relational DBAs	CL284
DB2 10.1 for Linux, UNIX, and Windows Quickstart for Exp. Relational DBAs	CL484
DB2 10.1 for Linux, UNIX, and Windows New Feat. and DB Migr. Considerations	CL313
DB2 10 for Linux, UNIX, and Windows: Basic Administration for Linux and Windows	CL2X3
DB2 9 pureScale Implementation and Ctrl for DB2 for Linux, UNIX, and Windows Admins	CL800
Query XML Data with DB2 9	CL121
Manage XML Data with DB2 9	CL141
Query and Manage XML Data with DB2 9	CL131
DB2 Performance Tuning and Monitoring	CL413
DB2 Advanced Database Recovery	CF492
Oracle to DB2 Enablement Workshop	CL720

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

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

For scheduling and enrollment, call IBM training at 800-IBM-TEACH (426-8322) or go to:

http://www.ibm.com/training

1.3.1 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. IBM professional certification includes the following offerings:

- Exam 610, DB2 10.1 Fundamentals: IBM Certified Database Associate - DB2 10.1 Fundamentals
- Exam 611, DB2 10.1 DBA for Linux, UNIX, and Windows: IBM Certified Database Administrator - DB2 10.1 DBA for Linux, UNIX, and Windows
- Exam 543, DB2 9.7 Application Developer: IBM Certified Application Developer - DB2 9.7
- Exam 545, DB2 9.7 SQL Procedure Developer: IBM Certified Solution Developer - DB2 9.7 SQL Procedure

For additional information, go to:

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

1.3.2 Other resources

Here are some additional DB2 educational resources:

- IBM DB2 manuals:

  http://www-01.ibm.com/support/docview.wss?rs=71&uid=swg27009474
- IBM Redbooks publications:

  http://www.redbooks.ibm.com
- IBM Press books on DB2 and other products:

  http://www.redbooks.ibm.com/ibmpress/
- IBM DB2 Express-C Edition is a version of DB2 that is available at no cost and can be downloaded from the following address:

  http://www.ibm.com/developerworks/downloads/im/udbexp/?S_TACT=105AGX01&S_CMP=HP
- IBM developerWorks®:

  http://www.ibm.com/developerworks

1.3.3 DB2 10 videos and topics

The following videos and topics provide introductions to some of the features available in DB2 10.

Videos

The following videos are available from YouTube:

- Native PL/SQL support:

  http://www.youtube.com/watch?v=EnpDMvobUmE

- Moving to DB2 is easy:

  http://www.youtube.com/watch?v=HJx3KZ5byN0

- CLPPlus:

  http://www.youtube.com/watch?v=3PndCKWlpJk

- Online Schema change:

  http://www.youtube.com/watch?v=wvM0xl9OXyE

- DB2 10.1 - Time Travel Query:

  http://www.youtube.com/watch?v=7JrQdzdYw0A

Topics

Run Oracle applications on DB2 10.1 for Linux, UNIX, and Windows is available from IBM developerWorks at:

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

2

Language compatibility features

IBM DB2 10.5 for Linux, UNIX, and Windows operating systems includes Structured Query Language (SQL) and Procedural Language SQL (PL/SQL) capabilities that facilitate database enablement from Oracle. These features 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 same speed and efficiency that the DB2 product offers.

This chapter introduces the language features that DB2 supports and provides examples of their use. It includes the following sections:

- DB2 compatibility features references
- Schema compatibility features
- DB2 command-line utilities

2.1 DB2 compatibility features references

The DB2 SQL compatibility features eliminate the need to convert most Oracle database objects and Oracle SQL to DB2 syntax. This section provides examples of typical Oracle syntax constructs that are supported in DB2. It also describes how to perform manual conversions for Oracle objects and features that are not natively supported on DB2.

About the examples in this section: The examples in this section illustrate techniques for handling exceptions in the database enablement process using both PL/SQL and SQL Procedural Language (SQL PL). To understand these examples, you need prior application development experience in either Oracle PL/SQL or DB2 SQL PL.

If no DDL is specified, the examples in this book are based on the tables and other database objects that are included in Appendix E, “Code samples” on page 351 or from the DB2 Sample database that is provided at DB2 installation time.

2.1.1 SQL compatibility setup

DB2 compatibility features ease the task of moving applications that are written for Oracle, Sybase, and MySQL to DB2. You can use the **`DB2_COMPATIBILITY_VECTOR`** registry variable 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. You enable individual DB2 compatibility features by specifying a hexadecimal value for the registry variable or by using the symbolic values to take full advantage of the DB2 compatibility features.

Run **`db2set`** to set the value to one of the symbolic values that are listed in Table 2-1.

Table 2-1 DB2_COMPATIBILITY_VECTOR symbolic values

Symbol	Description
ORA	Oracle full compatibility set
SYB	Sybase full compatibility set
MYS	MySQL full compatibility set

You can also selectively enable specific compatibility features by setting specific pieces of the **DB2_COMPATIBILITY_VECTOR** registry variable. For best results, when established, keep the selected compatibility level for the life of the database. Table 2-2 presents the possible variable settings for the **DB2_COMPATIBILITY_VECTOR** registry variable.

Table 2-2 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.
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 support for the VARCHAR2 and NVARCHAR2 data types and associated character string processing.
7 (0x40)	DATE data type[a]	Enables the interpretation of the DATE data type as the TIMESTAMP(0) data type, a combined date and time value. For example, “VALUES CURRENT DATE” in date compatibility mode returns a value such as 2011-02-17-10.43.55.
8 (0x80)	TRUNCATE TABLE	Enables alternative 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 runs 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*) refers to element *i* of array1.

Bit position	Compatibility feature	Description
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.
13 (0x1000)	Insensitive cursors	Enables cursors that are defined with WITH RETURN to be insensitive if the select-statement does not explicitly specify FOR UPDATE.
14 (0x2000)	INOUT parameters	Enables the specification of `DEFAULT` for **`INOUT`** parameter declarations
15 (0x8000)	LIMIT and OFFSET clauses	Enables the use of the MySQL- and PostgreSQL-compatible LIMIT and OFFSET clauses on fullselect, UPDATE, and DELETE statements.
17 (0x10000)	SQL data-access-level enforcement	Enables routines to enforce SQL data-access levels at run time.
18 (0x20000)	Oracle database link syntax	Enables Oracle database link syntax for accessing objects in other databases.

a. Applicable only during database creation. Enabling or disabling this feature affects only later created databases.
b. For more information, see:
`http://pic.dhe.ibm.com/infocenter/db2luw/v10r5/index.jsp?topic=/com.ibm.db2.luw.apdv.plsql.doc/doc/c0053608.html`

You must set the **`DB2_COMPATIBILITY_VECTOR`** registry variable to the wanted level before you create a database, as shown in Example 2-1. In addition, you must also restart the DB2 instance after the value is changed for it to take effect.

Example 2-1 Setting DB2_COMPATIBILITY_VECTOR

```
db2set DB2_COMPATIBILITY_VECTOR=ORA
```

The **`DB2_DEFERRED_PREPARE_SEMANTICS`** registry variable also enhances compatibility between Oracle and DB2 user applications, such as those written in Java. By setting this registry variable to `YES`, dynamic SQL statements are not evaluated at the PREPARE step, but rather are evaluated on **`OPEN`** or **`EXECUTE`** calls. You can use this setting to take advantage of the DB2 implicit data type casting feature. It also 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) facility. This facility is turned off by default, although you can still define and modify jobs (tasks).

To enable the ATS, set the variable that is shown in Example 2-3.

Example 2-3 Setting DB2_ATS_ENABLE

```
db2set DB2_ATS_ENABLE=YES
```

Example 2-4 demonstrates the commands and the correct sequence of these steps.

Example 2-4 Setting DB2_COMPATIBILITY_VECTOR

```
db2inst1> db2set DB2_COMPATIBILITY_VECTOR=ORA
db2inst1> db2set DB2_DEFERRED_PREPARE_SEMANTICS=YES
db2inst1> db2set DB2_ATS_ENABLE=YES
db2inst1> db2set -all
[i] DB2_COMPATIBILITY_VECTOR=ORA
[i] DB2_DEFERRED_PREPARE_SEMANTICS=YES
[i] DB2_ATS_ENABLE=YES
[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"
```

For more information, see the Information Center at:

http://pic.dhe.ibm.com/infocenter/db2luw/v10r5/index.jsp?topic=%2Fcom.ibm.db2.luw.admin.gui.doc%2Fdoc%2Ft0054396.html

The following database configuration parameters can also simplify the enablement process:

- **AUTO_REVAL**

  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 (dropping of underlined table), an attempt to revalidate it is attempted automatically the next time it is referenced. Changing the **AUTO_REVAL** parameter value to DEFERRED_FORCE also allows new objects to be created successfully, even though they might depend on invalid objects.

  The DEFERRED_FORCE value also enables the CREATE value with an error feature. For example, a procedure is created successfully even though it relies on a table that does not exist yet. The procedure is marked as invalid until the table is created, and the procedure is revalidated automatically on first use. This process is especially convenient when you create large numbers of objects where it is difficult to execute the scripts in the correct dependency order.

- **DECFLT_ROUNDING**

  You can use the **DECFLT_ROUNDING** database configuration parameter to specify the rounding mode for a decimal floating point (DECFLOAT). This parameter defaults to round-half-even, but you can set it to round-half-up to more closely match the Oracle rounding mode.

  For the change to take effect, you must deactivate the database after you update the database configuration (Example 2-5).

  Example 2-5 DB2 database parameter settings

  ```
  connect to testdb;
  db2 update db cfg using auto_reval DEFERRED_FORCE;
  db2 update db cfg using decflt_rounding ROUND_HALF_UP;
  db2 update db cfg using nchar_mapping CHAR_CU32;
  db2 update db cfg using extended_row_sz ENABLE;
  db2 connect reset;
  db2 deactivate db testdb;
  db2 connect to testdb;
  ```

- **NCHAR_MAPPING**

  This parameter determines the data type mapping for national character string data types in Unicode databases. It optimizes NCHAR types to predominantly multi-byte or single-byte Unicode character environments. In Europe, Africa, and the Americas, UTF-8 (1 - 4 bytes encoding) is more efficient, and in Asia, UCS-2 (2 or 4 bytes encoding) is often more efficient.

NCHAR_MAPPING applies to all usages of NCHAR types in the database, for example:

```
CREATE TABLE emp(name NVARCHAR(20), resume NCLOB(1M));
```

- **EXTENDED_ROW_SZ**

  This is an important parameter that is introduced with DB2 10.5. The maximum database page size in DB2 is 32 KB, and every row must fit into one page. If your Oracle table has a larger row length, you must change your table definition, for example, convert a VACHAR2 column into a CLOB column. With **EXTENDED_ROW_SZ** enabled, this change is no longer necessary. The maximum row length of a table can now be up to 1 MB.

  As the page size limit is not lifted, DB2 converts the large string to LOB internally. For the application, this behavior is transparent. For example:

  ```
  CREATE TABLE emp(
     name    VARCHAR(4000),
     address VARCHAR(4000),
     cv      VARCHAR(32000));
  ```

  The page size limit is not enforced only during table creation, but also during statement execution because the row length of a result set is bound to the 32 KB page size limit. This limit is now lifted by using the same internal DB2 mechanism as for table definitions. The following statement runs successfully with **EXTENDED_ROW_SZ** enabled:

  ```
  SELECT name, address, cv FROM emp ORDER BY name;
  ```

  The only other thing you must do is define a 32 KB system temp table space in your database.

2.1.2 PL/SQL record and collection types

DB2 provides extended support for SQL data types, such as NUMBER, VARCHAR2, NVARCHAR2, and DATE, and some PL/SQL scalar types, including BOOLEAN, BINARY_INTEGER, PLS_INTEGER, and RAW This extended support ensures a simplified enablement process so that you can run 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 309.

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

Record types

Record types are supported in PL/SQL contexts when declared as part of PL/SQL packages (header or body). Alternatively, you can create the same type of constructs outside of a PL/SQL package and still reference them inside of routines and packages.

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. Here is the syntax of a user-defined record type:

```
TYPE <type_name> IS RECORD (fiel1 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 2-6 shows a package specification that creates a user-defined record type. This type is immediately used as an **`IN OUT`** parameter in the test_record_sp procedure. One of the fields in the records is defined with the `%TYPE` attribute.

Example 2-6 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;
/
```

The %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. 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 is assigned to the variable that is being declared with the `%TYPE` attribute.

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

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

Example 2-7 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` using the `%TYPE` attribute. The **`IN`** parameter `p_id1` is also defined with `%TYPE` attribute and has the same data type as the ID column of the `Test_table`.

Example 2-7 Using %TYPE in a 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;
/
```

The previous example is equivalent to Example 2-8, where simple data type declarations are used.

Example 2-8 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 2-9 shows another example of the %TYPE attribute.

Example 2-9 Using the %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
```

Collection types

The DB2 data server supports collection types, such as the `VARRAY` collection type and associative arrays in PL/SQL. 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 using a subscript notation within parentheses. When the database is in Oracle compatible mode, you can use collection methods to obtain information about collections or to modify them.

The VARRAY collection type

DB2 supports the `VARRAY` collection type, where each element of a scalar data type is referenced by a positive integer, called the *array index*. The maximum cardinality of the `VARRAY` collection type (the maximum value of the array index) is defined in the type definition of the `VARRAY` collection but cannot exceed 2147483647.

DB2 also supports nested array and row types. A *nested* type is a complex data type that references another complex data type. You can nest the following types:

- An array type as an element of an array type
- An array or row type as a field of a row type

The maximum nesting level for arrays and row types is 16. Performing inserts or updates in deeply nested array and row types requires careful use of the syntax.

In partitioned environments, support is available only for top-level `SET` and `CALL` statements that reference objects defined in nested types. Do not reference objects with nested types in a subquery.

You cannot use chained access operators to access deep nested objects. Instead, you must create a local variable and use one level access operator at the time. For example, if you have an array of arrays of integers that are called `int_int_array` that accesses the integer with the index 1 inside of the array index 2, you cannot access the array through the expression `int_int_array(2)(1)`. You must define a local variable (called `int_array` for example) and access it in two steps, first using `int_array:=int_int_array(2)` and then using `int_array(1)` to access the nested value.

The PL/SQL `VARRAY` syntax is as follows:

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

Table 2-3 summarizes the `VARRAY` collection methods that are supported by the DB2 data server in a PL/SQL context.

Table 2-3 VARRAY collection methods that are 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 before 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, do not define the `VARRAY` keyword either inside the PL/SQL packages or in stand-alone **`CREATE TYPE`** statements.

Example 2-10 shows how to declare a VARRAY as part of a PL/SQL package and how to start this VARRAY and its collection methods in an anonymous block. This syntax is similar to the syntax that is used in Oracle. The example also demonstrates how to insert multiple rows in a table.

Example 2-10 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);
   TYPE emp_group_typ is VARRAY(10) of emp_arr_typ; -- nesting arrays
END;
/

SET SERVEROUTPUT ON
/

DECLARE
   emp_arr        Types_package.emp_arr_typ;
   emp_arr_1      Types_package.emp_arr_typ;
   emp_arr_2      Types_package.emp_arr_typ;
   emp_grp        Types_package.emp_group_typ;

    CURSOR emp_cur IS SELECT ename FROM emp
      WHERE ROWNUM <= 5;
    i               INTEGER := 0;
    k               INTEGER := 0;
    l               INTEGER := 0;
   h           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;

   -- Excercise nested collections:
   h := emp_arr.LAST/2;
    FOR j IN emp_arr.FIRST..emp_arr.LAST LOOP
      if j < h then
         emp_arr_1[j] := emp_arr[j];
      else
         emp_arr_1[j-h+1] := emp_arr[j];
      end if;
   END LOOP;

   -- create the nested array
   emp_grp[1]:=emp_arr_1;
   emp_grp[2]:= emp_arr_2;

   -- to access the second group elements we use one of the existing emp_arr_x to 
hold the temporary value (any variable of same tile would do)

   emp_arr_1 := emp_grp[2];
   FOR j IN emp_arr_1.FIRST..emp_arr_1.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;
/
```

You can also use the DB2 SQL PL syntax to declare an array and use it in a PL/SQL procedure in the same fashion as with `VARRAY` (Example 2-11). In this example, `myVarrayType` is defined as a stand-alone array type with DB2 syntax and used later in the `TEST_ARRAY` PL/SQL procedure.

Example 2-11 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 array (index by tables) data types

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

An *associative array* data type is a data type that is 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, 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. You can continue to add elements to the array without concern for a maximum size, which is useful if you do not know in advance how many elements constitute a set.
- The array index value can be a non-integer data type. The `VARCHAR` and `INTEGER` values 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. There are not necessarily index elements for all possible index values between the lowest and highest value. This feature is useful, for example, if you want to create a set that stores names and phone numbers. You can add pairs of data to the set in any order and sort using the data item in the pair that is 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 using a set of available scalar functions.
- Associative arrays are supported in SQL PL contexts.

You can use associative arrays to manage and pass sets of values of the same kind in the form of a collection. With this method, you can avoid the following types of situations:

- Reducing the data to scalar values and using one-element-at-a-time processing, which can cause network traffic problems.
- Using cursors that are passed as parameters.
- Reducing the data to scalar values and reconstituting these values as a set using a `VALUES` clause.

Use the following syntax for the PL/SQL associative array:

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

Table 2-4 summarizes the associative array collection methods that are supported by the DB2 data server in a PL/SQL context. In general, DB2 arrays are not bounded and can increase dynamically. Therefore, EXTEND() is similar to a NO-OP.

Table 2-4 Associative arrays collection methods that are 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.

Collection method	Description
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 before the specified element.

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

Example 2-12 shows how to create, initialize, and display the values of an associative array. The `ROWTYPE` attribute is used to define `emp_arr_typ`.

Example 2-12 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;
/
```

Performance improvements using deep-nested objects

Nested object collections are frequently used by the OLTP applications as a design pattern that is intended to minimize the total time that is required by the application to retrieve full or partial complex data trees that are stored in classic relational structures (data shredding). You can use the DB2 pureXML feature to store this type of information. However, in certain cases, such as OLTP environments when the workload requires sparse leaf field updates in the data tree, the shredded form of storage might be the best storage form.

In these types of cases, retrieving the structure of the whole tree requires the application to run multiple queries and to fetch data from multiple results sets, each of which has different structures (heterogeneous queries). While under load when the application goes from one result set to another, the network operations and server-side session activity handling can introduce supplemental time because of activity fragmentation.

When the data retrieval time for a full data tree is expected to be in tenths of a millisecond range and a data tree be composed of tenths of distinct result sets (queries), each millisecond (or less) counts. As a consequence, Oracle OLTP applications tend to use trees of deep-nested objects and collections as a fast transport method by wrapping multiple individual rows sets in one single response object.

In DB2 10.5, you can use DB2 deep-nested object trees to achieve the same result for cases when the Oracle deep-nested objects collections are used. Slight differences in the type of objects that can be nested require changes to the PL/SQL code, but the application side is less impacted. When migrating deep-nested object collections, keep in mind the following rules:

- Replace Oracle structured Object types with DB2 Row types.
- Replace Oracle structured Object type constructors and methods with DB2 user-defined functions (UDFs).

Example 2-13 exemplifies this conversion strategy. The application must retrieve the data tree that is represented in Example 2-13 (for better readability) as an XML document.

Example 2-13 XML encoding of the sample data model

```
<?xml version="1.0" encoding="UTF-8"?>
<customer> <!-- Object, deep nested L0 -->
   <info> <!-- Object, deep nested L1 -->
      <name> <!-- Object L2 -->
         <first_name>John</first_name> <!-- Leaf value -->
         <last_name>Smith</last_name>
      </name>
      <birth_date>1963-03-29</birth_date>
   </info>
   <adresses> <!-- Object Array, deep nested L1 -->
      <address> <!-- Object, deep nested L2 -->
         <phones> <!-- Object Array, deep nested L3 -->
            <phone> <!-- Object L4 -->
               <phone_provider>Bell</phone_provider>
               <phone_number>000-111-2222</phone_number>
            </phone>
            <phone>
               <phone_provider>Rogers</phone_provider>
               <phone_number>000-111-2223</phone_number>
            </phone>
         </phones>
         <country>Canada</country>
         <country_div>Ontario</country_div>
         <city>Toronto</city>
         <street>Warden Ave.</street>
         <number>8200</number>
         <code>M2H-2P7</code>
         <building_address> <!-- Object, deep nested L3 -->
            <entry>A1</entry>
            <floor>2</floor>
            <apartment_number>120</apartment_number>
         </building_address>
      </address>
   </adresses>
</customer>
```

Using the DB2 nested objects and object instance factory UDFs, you can build the same document at the server side for retrieval in one single query (Example 2-14).

Example 2-14 DB2 "object notation" encoding of the model from the previous example

```
CUSTOMER(
   INFO(
      NAME('John','Smith'),
      '1967-02-23'
   ),
   ADDRESSES(
      ADDRESS(
         PHONES(
            PHONE(
               'Rogers',
               '000-111-2223'
            )
         ), -- phone array
         'Canada', -- country
         'Ontario', -- country_div
         'Toronto', -- city
         'Warden Ave.', -- street
         'M2H-2P7', -- code
         BUILDING_ADDRESS(
            'A1', -- entry
            3, -- floor
            120 -- apartment
         ) -- building_ddress
      ) -- address
   )-- address array
)
```

This structure can be retrieved at the application side using an application code equivalent to the code in the JUnit test case (Example 2-15).

Example 2-15 JUnit test client sample that retrieves the model from the previous example

```
public class ReadNestedObjectTest extends TestBase {
   ...
   public void testReadNestedObjectPackage()
     throws SQLException, ClassNotFoundException {
      Connection con = getConnection();
      CallableStatement cstmt = null;
      try {
         cstmt = con.prepareCall(
             "CALL
dnobj.generate_customer_sample_object_array(?,?,?,?,?)"
         );

         cstmt.registerOutParameter(1, java.sql.Types.ARRAY);
         cstmt.setInt(2, 1);// using same seed to keep the test happy
:)
         cstmt.setInt(3, 1);
         cstmt.setInt(4, 1);
         cstmt.setInt(5, 1);

         cstmt.execute();

         // simply dump of the received data tree by walking
         // the tree down for any "A" array or "S" structure
         // entity detected, leaf field discrimination is positional
         String result = sqlObjectToText("customers->",
cstmt.getArray(1));
         ...
         // verify by comparing to the expected value
         String expected=
              "customers->A[0]S[0]S[0]S[0]=John\n"
            + "customers->A[0]S[0]S[0]S[1]=Smith\n"
            + "customers->A[0]S[0]S[1]=2012-02-23 22:04:31.0\n"
            + "customers->A[0]S[1]A[0]S[0]A[0]S[0]=Rogers\n"
            + "customers->A[0]S[1]A[0]S[0]A[0]S[1]=000-111-2223\n"
            + "customers->A[0]S[1]A[0]S[1]=Canada\n"
            + "customers->A[0]S[1]A[0]S[2]=Ontario\n"
            + "customers->A[0]S[1]A[0]S[3]=Toronto\n"
            + "customers->A[0]S[1]A[0]S[4]=Warden Ave.\n"
            + "customers->A[0]S[1]A[0]S[5]=M2H-2P7\n"
            + "customers->A[0]S[1]A[0]S[6]S[0]=A1\n"
            + "customers->A[0]S[1]A[0]S[6]S[1]=3\n"
            + "customers->A[0]S[1]A[0]S[6]S[2]=120\n"
         ;
         assertEquals(expected,result);
      } finally {
         if(cstmt != null) cstmt.close();
         if(con   != null) con.close();
      }
   }
   ...
   // a method useful for walking a deep nested object and
   // convert it in a simple textual representation
   // the text representation is used for inspection and validation
   // of the result
   // In this representation Arrays are named with “A” and structures
“S”
   public String sqlObjectToText(String name, Object obj) {
      ByteArrayOutputStream bos = new ByteArrayOutputStream();
      PrintStream out = new PrintStream(bos);

      if (obj instanceof java.sql.Array) {
         try {
            Object[] a = (Object[]) ((Array) obj).getArray();
            for (int i = 0; i < a.length; ++i) {
               out.print(sqlObjectToText(name + "A[" + i + "]", a[i]));
            }
         } catch (SQLException e) {
            out.println(name + "Array:-exception->" + e.getMessage());
         }

      } else if (obj instanceof java.sql.Struct) {
         Struct a = (Struct) obj;
         try {
            Object[] attrs = a.getAttributes();
            for (int i = 0; i < attrs.length; ++i) {
               out.print(sqlObjectToText(name + "S[" + i + "]",
attrs[i]));
            }
         } catch (SQLException e) {
            out.println(name + "struct:-exception->" + e.getMessage());
         }
      } else {
         out.println(name + "=" + obj);
      }
      out.flush();
      return bos.toString();
   }
```

Object (structured) types are not supported in DB2. However, equivalent structures exist to support the same functions. When you convert Oracle object types (Example 2-16), you can use the DB2 ROW type to port the data structure and use UDFs to port object methods.

Example 2-16 Oracle object sample

```
create or replace
TYPE  PHONE_TYPE AS OBJECT
   (
     PHONE_PROVIDER VARCHAR2(32),
     PHONE_NUMBER VARCHAR2(32),
     CONSTRUCTOR FUNCTION PHONE_TYPE (
        p_phone_provider VARCHAR2,
        p_phone_nr       VARCHAR2
     )
      RETURN SELF AS RESULT
   )
/
CREATE OR REPLACE
 TYPE BODY  PHONE_TYPE AS
   CONSTRUCTOR FUNCTION PHONE_TYPE (
     p_phone_provider VARCHAR2,
     p_phone_nr       VARCHAR2
   )
      RETURN SELF AS RESULT
   AS
   BEGIN
        PHONE_PROVIDER := p_phone_provider;
        PHONE_NUMBER   := p_phone_nr;
      RETURN;
   END;
END;
/
```

Example 2-17 shows the code that is converted to DB2 compatible syntax.

Example 2-17 Conversion to DB2 row type to allow deep nesting of the structures

```
create or replace
   TYPE  PHONE_TYPE AS ROW
   (
     PHONE_PROVIDER VARCHAR2(32),
     PHONE_NUMBER VARCHAR2(32)
   )
/
create or replace
   FUNCTION PHONE (
     PHONE_PROVIDERVARCHAR2(32),
     PHONE_NUMBERVARCHAR2(32)
   ) RETURNS PHONE_TYPE
   LANGUAGE SQL
   BEGIN
      declare OBJ PHONE_TYPE;
    set OBJ.PHONE_PROVIDER= PHONE_PROVIDER;
```

```
      set OBJ.PHONE_NUMBER= PHONE_NUMBER;
      RETURN OBJ;
   END
/
```

To convert Oracle collections of structured objects (Example 2-18), you must change the syntax of the type definition (Example 2-19).

Example 2-18 Oracle collection of a structured object sample

```
CREATE OR REPLACE
  TYPE PHONE_TYPE_ARRAY AS
  TABLE OF PHONE_TYPE
/
```

Example 2-19 Array of rows for the equivalent type definition of Oracle type

```
CREATE OR REPLACE
  TYPE PHONE_TYPE AS
  ARRAY[]
/
```

The %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 2-20 demonstrates a declaration of a variable that is associated with the table T1 that is defined with the `%ROWTYPE` attribute. Note the calls to the record fields with a dotted notation of <*variable*>.<*table_field*>.

Example 2-20 Using the %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 2-21 shows another example of the `%ROWTYPE` attribute.

Example 2-21 Using the %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;
/
```

2.1.3 Subtypes

A subtype is a definition of a type that is based on a built-in type. Subtypes provide a layer of abstraction between variables and parameters and the data types that they use. This layer allows you to concentrate any changes to the data types in one location. You can add constraints to subtypes so that they cannot be nullable or limited to a specific range of values.

Subtypes can be defined in the following statements as user-defined data types for variables and parameters:

- `CREATE PACKAGE (PL/SQL)`
- `CREATE PACKAGE BODY (PL/SQL)`
- `CREATE PROCEDURE (PL/SQL)`
- `CREATE FUNCTION (PL/SQL)`
- `CREATE TRIGGER (PL/SQL)`
- Anonymous blocks (PL/SQL)

Here is the syntax for subtypes:

```
SUBTYPE type-name IS built-in-type RANGE start-value .. end-value NOT
NULL
```

- `type-name` specifies an identifier for the subtype.
- `Built-in-type` specifies the data type that the subtype is based on. You cannot specify `BOOLEAN` as the built-in type.
- `RANGE start-value .. end-value` is an optional clause and defines a range of values within the domain of the subtype that is valid for the subtype.
- `NOT NULL` is optional too and defines that the subtype is not nullable.

Example 2-22 explains the usage of the subtypes in PL/SQL code.

Example 2-22 Example of subtype usage

```
DECLARE
   SUBTYPE tinyint IS INTEGER RANGE -256..255 NOT NULL;
   val tinyint := 255;
BEGIN
   val := val + 1;
END;
/
```

At run time, you receive the following error message:

```
SQL20552N The cast or assignment failed because the value does not
conform to the data type constraint of the user-defined type.
User-defined type: "TINYINT". Value: "256".
```

This error message appears when the values of a parameter or a variable that is based on the subtype violate the boundaries of the defined range of that subtype.

2.1.4 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 2-23 shows the assignment syntax in several combinations, such as variables, parameters, and constants.

Example 2-23 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;
/
```

The 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 2-23 on page 47, Example 2-24 demonstrates the variable assignment, where the cursor `cur0` is assigned the current row.

Example 2-24 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 block structures

PL/SQL block structures can be included within a PL/SQL procedure, function, or trigger definitions, or can be 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
- 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 2-25 demonstrates the usage of a PL/SQL block.

Example 2-25 PL/SQL block

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

For more details about anonymous blocks, see "Anonymous blocks" on page 84.

SQL statements

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

Table 2-5 lists the SQL statements that can be run by the DB2 server within PL/SQL contexts.

Table 2-5 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.

The NULL statement

The **NULL** statement can act as a placeholder where an executable statement is required, but an SQL operation is not wanted. Example 2-26 demonstrates a **NULL** statement that is used as a minimum executable statement that must exist between the BEGIN-END block; otherwise, a syntax error occurs.

Example 2-26 NULL statement usage

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

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

The 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, the **EXECUTE IMMEDIATE** statement can execute an anonymous PL/SQL block. Because the **EXECUTE IMMEDIATE** statement is used primarily for dynamic SQL execution, this topic is described in 2.1.8, "Static and dynamic SQL support" on page 69.

The RETURNING INTO clause

A useful feature of SQL execution is the **RETURNING INTO** clause that can be optionally appended to the **INSERT**, **DELETE**, or **UPDATE** statements, which typically does row by row processing. You can use this feature to access the changed rows in an atomic manner without making an additional selection, which can have locking implications, because the selection happens later and makes the SQL code more efficient. If there is not a changed row, the **RETURNING INTO** values are undefined.

Example 2-27 shows how the **RETURNING INTO** clause is used with the **INSERT**, **DELETE**, and **UPDATE** statements that return one value. This example also demonstrates an arithmetic operation with implicit casting in the **VALUES** clause of the **SELECT** statement.

Example 2-27 DELELE, INSERT, or 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 formated 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;
/
```

Assume that you have a denormalized EMP table 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.

The BULK COLLECT and FORALL statements

DB2 supports **`BULK COLLECT`** and **`FORALL`** syntax, including **`INDICES OF`** and **`VALUES OF`** clauses in **`FORALL`**. **`BULK COLLECT`** and **`FORALL`** are PL/SLQ statements that optimize processing and fetching of collection information from associative arrays and varrays.

Statement attributes

You can use the following PL/SQL attributes to determine the effect of an SQL statement:

- `SQL%FOUND`

  This attribute has a Boolean value that returns `TRUE` if at least one row is affected by an **`INSERT`**, **`UPDATE`**, or **`DELETE`** statement, or if a **`SELECT INTO`** statement retrieved one row.

Example 2-28 shows an anonymous block in which a row is inserted and a status message is displayed.

Example 2-28 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 are 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 a `NO_DATA_FOUND` exception rather than `SQL%NOTFOUND` in this case.

  Example 2-29 shows how to use `SQL%NOTFOUND`.

Example 2-29 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 are affected by an **`INSERT`**, **`UPDATE`**, or **`DELETE`** statement. Example 2-30 illustrates how to use `SQL%ROWCOUNT`.

Example 2-30 Using SQL%ROWCOUNT

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

2.1.5 Control of flow statements

The control of flow statements are programming constructions that you can use to group, relate, and organize different SQL or PL/SQL statements through internal procedural logic rather than relying on their sequential execution. These statements control the execution logic (“flow”) of the basic procedural statements, statement blocks, routines, and so on. As such, they are an important part of the programming concepts in PL/SQL and are supported by both Oracle and DB2.

Decision making statement: IF and CASE statements

You can use the **IF** statement within PL/SQL contexts to run PL/SQL statements that are based on certain criteria.

Example 2-31 shows an **IF THEN ELSIF END IF** routing statement.

Example 2-31 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, **CASE** statements and expressions execute one or a set of statements when a specified search condition is true. The **CASE** statement is a stand-alone statement that is distinct from the CASE expression, which must appear as part of an expression.

CASE statements and expressions can take one of the following forms:

- Simple **CASE** statement

  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 2-32 to Example 2-34 on page 55 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 2-32 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 2-33 shows the CASE statement from Example 2-32 in PL/SQL.

Example 2-33 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');
      ELSE
        DBMS_OUTPUT.PUT_LINE('Not IT related');
    END CASE;
  END LOOP;
END;
/
```

Example 2-34 shows a simple CASE expression in PL/SQL.

Example 2-34 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`** statement

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

  Example 2-35 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 2-32 on page 54; only a comparison expression is used to find a match.

Example 2-35 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 2-36 shows basic **LOOP** and **EXIT** statements within an anonymous block.

Example 2-36 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 if 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 2-37 shows a **WHILE LOOP** statement.

Example 2-37 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 2-38 shows a simple **FOR** (integer variant) statement (or **LOOP**) that loops 10 times.

Example 2-38 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** statement 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 2-39 shows a **FOR LOOP** statement over a cursor statement that uses the cursor values mapped from the cursor column for display.

Example 2-39 A FOR LOOP statement 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

Do not use **GOTO** instructions outside of assembler code. However, you can use the **GOTO** and **LABLE** statements in case of errors to unconditionally redirect the programming logic to a statement or block label. DB2 supports the **GOTO *label_name*** statement, where the label must be defined with the <<*label_name*>> syntax and must be contained in the same PL/SQL block, either at the same level or higher up in the hierarchy.

Example 2-40 shows how to use a **GOTO** statement to exit the loop prematurely to the label **exit_perm**.

Example 2-40 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_prem; END IF;
         DBMS_OUTPUT.PUT_LINE(i.table_name);
      END LOOP;
    <<exit_prem>>
   DBMS_OUTPUT.PUT_LINE('done');
END;
/
```

2.1.6 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`).

Example 2-41 shows the general syntax for exception handling in a `BEGIN` block.

Example 2-41 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

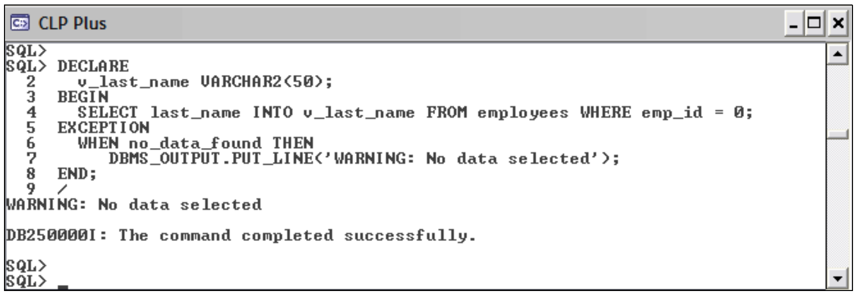
DB2 provides support for the following Oracle predefined exceptions:

- 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
- ZERO_DIVIDE

These exceptions use the following general syntax:

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

Figure 2-1 demonstrates an anonymous block that returns a warning as a result of caching a `NO_DATA_FOUND` exception in CLPPlus.



```
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 2-42, custom exceptions are defined with an exception declaration and the `PRAGMA EXCEPTION_INIT` syntax.

Example 2-42 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. Example 2-43 declares a user-defined exception named `AccountNotRegistered` in the package header. A separate procedure named `Remove_Account` reuses this global exception by calling it with `<package_name>.<exception_name>`, as shown in Example 2-44.

Example 2-43 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 2-44 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

You can use the **`RAISE`** statement to raise a previously defined condition. For details, see Example 2-43 and Example 2-44.

RAISE_APPLICATION_ERROR

The **RAISE_APPLICATION_ERROR** procedure provides the capability to intentionally stop 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 previously. 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 procedure uses the following general syntax:

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

Example 2-45 shows a procedure that is 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 2-45 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
DB250000I: The command completed successfully.
```

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

http://publib.boulder.ibm.com/infocenter/db2luw/v10r5/topic/com.ibm.db2.luw.apdv.plsql.doc/doc/r0055262.html

2.1.7 Cursor 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. You can use a cursor in the following contexts:

- User-defined cursors types
- Global variables
- Parameters to procedures and functions
- Local variables
- Return types of functions

To use cursors, follow this general process:

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, 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 database resources.

An application (PL/SQL program or client program) consumes the records that are produced by the cursors by fetching the records from the cursor one by one. Cursors are record-at-a-time lengthy operations. Thus, they are less efficient than a *set*-at-a-time operation. Cursors hold resources while opened. When a cursor is the main construct that is used for passing information from RDBMS to client applications, you must manage them properly. Close cursors as soon as they are no longer in use or minimize cursors use for better performance.

Depending on the cursor type, the operations that you do on cursors might be slightly different.

REF CURSOR

`REF CURSOR` is a type in PL/SQL that allows you to define cursor variables. These variables hold the pointers to the cursors. Cursor variables are frequently used to pass the result sets from the queries between various PL/SQL objects. In DB2, `REF CURSOR` must be created inside a package.

Example 2-46 shows `REF CURSOR` with the `Cur0` cursor variable.

Example 2-46 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 not bound to a particular result set or type. If you need a weakly typed cursor, use the `SYS_REFCURSOR` type. DB2 supports weakly typed cursors.

You can use weakly typed cursors when the definition of the result set is created dynamically or at run time. Weakly typed cursors have limited or no type checking; therefore, their row structures are discovered only at run time by applications. This limited checking is often used to flow cursors around or to pass cursors back and forth as parameters between procedures and functions. Weakly typed cursor variables are ideal for holding different cursor types at different points of time. For example, a procedure can be a "cursor factory" and can return different cursor types in to the same **`OUT`** parameter based on various values or logic, such as a **`CASE`** statement.

Example 2-47 shows a procedure using a weakly typed cursor.

Example 2-47 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.deptname);
   END LOOP;
   CLOSE v_ref_cur;
END;
/
```

Strongly typed cursors

Strongly typed cursors are cursors with a `RETURN` clause that define the number of columns and the types that 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. These types of cursors are used mainly for static business logic.

Strongly typed cursors are advantageous because their structures and result sets type can be verified or checked by the compilation process or at assignment times. 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 2-47 on page 65 cannot use a strongly typed cursor because the strongly typed cursor by definition is tied only to single table or view. In Example 2-48, the reference cursor is anchored to the entire EMPLOYEE table, and it retrieves all the columns from that table.

Example 2-48 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 that are 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 2-49 demonstrates a usage of the implicit cursor. It does not include an explicit cursor declaration.

Example 2-49 Using an 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 the parameters are associated with the cursor in the definition, as shown in Example 2-50. DB2 requires that you specify the length of the parameter in the parameterized cursor, `name0 VARCHAR2(30)`.

Example 2-50 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. Example 2-51 modifies the parameterized cursor example (Example 2-50 on page 67) to illustrate the static cursor.

Example 2-51 Using a 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;
/
```

A word about cursor usage in OLTP environments

An OLTP environment workload profile usually implies many small-size transactions in a short period (high workload fragmentation). This workload also implies that the database server must handle a high number of connections that run diverse SQL statements, each handling a relatively small amount of data.

For these types of environments (or workloads), each byte of memory in the database server counts, even if those machines usually have hundreds of gigabytes of dynamic memory. This state or process is because of the large number of connections and high rate of transactions per second required to be handled by the server.

In these types of cases, returning a cursor to the application layer (outside of the database) poses special performance challenges. The developer must evaluate the long-term impact of using the cursor pattern in a production environment.

Avoid returning cursor variables to the application layer or, if that is not possible, use insensitive or forward only cursor types. The application code must read the full amount of data from the cursors and close (release the server-side resources) as soon as possible. This development pattern can provide much better throughput and overall application performance. Also, if more than one cursor is returned by the SQL statement, wrapping all the data using XML publishing or deep-nested objects (a combination of rows and arrays) in one single entity can improve the overall throughput with the database server. This case is specific to applications that retrieve the full data model of a complex entity (such as a customer profile) and then work with it in the application server memory.

You want to avoid the worst case scenario where the application layer opens a cursor for update and keeps the cursor open for an undetermined amount of time. This pattern might work well in non-OLTP environments, but in an OLTP environment, it can lead to low performance and locking issues.

2.1.8 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. *Dynamic* SQL is compiled at run time unless the compiled package happens to be cached in memory when the SQL is started.

The DB2 compatibility feature for PL/SQL allows both static and dynamic SQL. The static SQL statements that are supported can be in data definition language (DDL), data manipulation language (DML), or a transaction control (**COMMIT** or **ROLLBACK**).

Using the EXECUTE IMMEDIATE statement

The **EXECUTE IMMEDIATE** statement is the primary mechanism for running a dynamic statement. With this statement, SQL statements are not fully defined until run time. In addition, specific static DDL SQLs cannot be run otherwise. The **EXECUTE IMMEDIATE** statement can also run a PL/SQL block.

It is sometimes convenient to execute DDL statements from within PL/SQL. Example 2-52 shows how to use the **EXECUTE IMMEDIATE** statement to make dynamic changes to an existing table.

Example 2-52 The EXECUTE IMMEDIATE statement

```
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 you manipulate data within a database server, it is usually better to use statically compiled code. In this case, the database compiles, verifies, and optimizes the SQL statements immediately.

Example 2-53 shows a simple procedure with a few static SQL statements for manipulating data.

Example 2-53 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;
/
```

In this case, using the **EXECUTE IMMEDIATE** statement (Example 2-54) to achieve the same purpose provides no advantage. In addition, if a syntax error is present in one of these **EXECUTE IMMEDIATE** statements, it is discovered at their run time, which is too late, instead of being caught at compilation time, which is early in the development process.

Example 2-54 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;
/
```

Alternatively, the **EXECUTE IMMEDIATE** statement is often used to run a DML that is constructed, as shown in Example 2-55. It also demonstrates how variables can be passed between procedures.

Example 2-55 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 2-56 shows an equivalent form of Example 2-55.

Example 2-56 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 2-57 shows another dynamic SQL execution where the SQL statement is constructed dynamically and run using **`EXECUTE IMMEDIATE`**.

Example 2-57 Using EXECUTE IMMEDIATE

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

`EXECUTE IMMEDIATE` can also run a PL/SQL block, in addition to the dynamic or static SQL statements. In this case, we must ensure that a complete block is specified. An anonymous block example that shows the execution of a **`CALL`** statement that is wrapped in a block is shown in Example 2-58.

Example 2-58 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 procedure has arguments, use the following syntax
   --- EXECUTE IMMEDIATE 'BEGIN add_emp3(:1, :2 ,:3); END'
   --- using id, LName, FName;
END;
/
```

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

The EXECUTE IMMEDIATE statement usage patterns

The **EXECUTE IMMEDIATE** statement is useful for cases when the algorithm of the data processing is not known at the time the code was built. Using it as a way to save some of the developer's time in OLTP or cases when the logic is then used by other logic (computer to computer call) is usually not justified.

You can use the **EXECUTE IMMEDIATE** statement for cases when parts of the actual statement or statements are not known until run time. Limit those cases to a minimum for better performance and security levels.

The performance can be impacted in different ways. For example, the added compile and optimization time adds to the performance impact. In addition, if you have a DDL statement that drops and re-creates tables that are then filled with data (staging tables, for example), you must also run the statistic collection on those tables and rebind the dependent statements. The worst case scenario is when those tables have no statistics or the statistics are captured when the new table is empty.

Security of the application can also be impacted by allowing the user to inject (untested) code at run time.

2.1.9 Support for built-in scalar functions

DB2 provides built-in functions that increase the compatibility of applications that were 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, and table functions.

Character and string functions

The following character and string functions primarily deal with manipulation and processing of strings to produce a modified output that is based on the specific action that is performed on the input string:

- ASCII
- CHR, CHAR
- CONCAT
- CONCAT with ||
- INITCAP
- INSTR
- INSTRB

- LENGTH
- LOWER
- LPAD
- LTRIM
- REPLACE
- RPAD
- TO_SINGLE_BYTE
- RTRIM
- SOUNDEX
- SUBSTR
- SUBSTR4
- TRANSLATE
- TRIM
- UPPER

Conversion functions

DB2 extends the **`TO_CHAR`**, **`TO_NUMBER`**, **`TO_DATE`**, and **`TO_TIMESTAMP`** conversion functions. These functions are identical to their respective Oracle functions, except for the support of the NLS parameter.

The following conversion functions are supported in DB2:

- **`CAST`**
- **`CONVERT`**
- **`TO_CHAR`**
- **`TO_DATE`**
- **`TO_TIMESTAMP`**
- **`TO_CLOB`**
- **`TO_NUMBER`**

Date calculation functions

The date calculation functions manipulate the date and time. `TO_DATE` and `TO_CHAR` (especially useful for date manipulations) are also listed under conversion functions. The following functions are the most widely used data manipulation functions:

- **`ADD_MONTHS`**
- **`CURRENT_DATE`**
- **`CURRENT_TIMESTAMP`**
- **`LAST_DAY`**
- **`TIMESTAMPDIFF`**
- **`LOCALTIMESTAMP`**
- **`MONTHS_BETWEEN`**
- **`NEXT_DAY`**
- **`ROUND (`*`date`*`)`**

- **SYSDATE**
- **TO_CHAR**
- **TO_DATE**
- **TRUNC (*date*)**

Time zone functions

Oracle supports several functions that provide time zone information. In DB2, you can implement this function with custom functions that use the DB2 Timestamp data type. The next examples demonstrate how to use the DB2 date scalar functions to obtain the same output.

Example 2-59 shows how to implement the Oracle **FROM_TZ** function in DB2 to provide time zone information.

Example 2-59 From_TZ implementation in DB2

```
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 2-60 shows a simple implementation for Oracle **TZ_OFFSET** function that returns the zone offset for a time zone. The exception checking is not included for simplicity. Also, not all the zones are shown, because there are more than 400 zones.

Example 2-60   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**, you can develop a DB2 version of the **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*)**. This function returns the correct output only if the database is in Oracle compatibility mode.

Example 2-61 shows a simple implementation (assuming that you use the correct inputs).

Example 2-61 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 use Interval Year To Date and Interval Day To Second data types could be implemented in DB2 by using the DB2 timestamp data type, which components are considered as intervals, not a point in time. 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, see *Oracle to DB2 Conversion Guide for Linux, UNIX, and Windows*, SG24-7048.

Mathematical functions

DB2 also provides the following set of mathematical functions:

- **ABS**
- **ACOS**
- **ASIN**
- **ATAN**
- **ATAN2**
- **AVG**
- **BITAND**
- **BITANDNOT**
- **BITOR**
- **BITXOR**
- **CEIL**
- **COS**
- **COSH**
- **COUNT**
- **DENSE_RANK**
- **EXP**
- **EXTRACT**
- **FLOOR**
- **GREATEST**
- **LEAST**
- **MAX**
- **MIN**
- **MOD**
- **LN**
- **POWER**
- **RANK**
- **ROUND (*numbers*)**
- **SIGN**
- **SIN**
- **SINH**
- **STDDEV**
- **SUM**

- **TAN**
- **SQRT**
- **TANH**
- **TRUNC (*numbers*)**
- **VAR_POP**
- **VAR_SAMP**
- **VARIANCE**

Error functions

The **SQLCODE** and **SQLERRM** error functions raise exception or retrieve error codes. These functions can be used only in the EXCEPTION section of PL/SQL blocks.

SQLCODE

The **SQLCODE** function returns the SQLCODE value that is associated with the raised exception. Example 2-62 shows a usage of the **SQLCODE** function.

Example 2-62 SQLCODE function

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

SQLERRM

The **SQLERRM** function returns the error message that is associated with the raised exception. Example 2-63 shows a usage of the **SQLERRM** function.

Example 2-63 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 the following Oracle functions:

- **CARDINALITY**
- **COALESCE**
- **CURRENT_USER**
- **DECODE**
- **LAG**
- **LEAD**

- **NULLIF**
- **NVL**

The following examples illustrate the **DECODE** and **NVL** functions for reference to show that there is no difference in how these functions operate on DB2 and Oracle. The examples also demonstrate how to implement the **USERENV** function in DB2.

DECODE

DB2 supports the **DECODE** function in the same way Oracle does, which is also similar to the **CASE** statement. In Example 2-64, the query selects the number of current projects per employee and displays a conditional message that is based on this number.

Example 2-64 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 2-65 demonstrates part of the result set that is returned by the query in **DECODE**.

Example 2-65 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 2-66 returns a specific message when department information is not yet assigned to the employee.

Example 2-66 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 2-67 presents the output of this query. Note the department description for the NEW department code.

Example 2-67 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 that are used in the database industry, and each RDBMS provides specific functions that are 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 can develop a corresponding DB2 equivalent 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 2-68 shows an example of RawToHex implementation in DB2.

Example 2-68 RawToHex implementation

```
CREATE OR REPLACE FUNCTION RAW2HEX
     (x1 VARCHAR(100) FOR BIT DATA)
```

```
      RETURNS VARCHAR2(100)
      SOURCE HEX(VARCHAR(100) FOR BIT DATA)
```

USERENV

The **USERENV** function is specific to Oracle. You can use it to retrieve information about the current Oracle session. The function accepts several different input parameters, as listed in Table 2-6.

Table 2-6 Parameters that could be passed to the USERENV function

Parameter	Returned value
CLIENT_INFO	The user session information that is inserted using the DEMS_APPLICATION_INFO package.
ENTRYID	Current session ID for auditing session
SESSIONID	Current session ID
INSTANCE	Oracle instance ID
ISDBA	TRUE or FALSE depending on whether the user is granted DBA privileges
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

To enforce the language statement, Example 2-69 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 2-69 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;
```

2.1.10 Routines, procedures, and functions compatibility

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 might not return a value, but a function takes zero or more inputs/outputs and always returns an output value. During run time, the procedure is started as a PL/SQL statement from the command line, another procedure, or function, trigger, or anonymous block. A function is always started 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 are not 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. During the execution of anonymous blocks, if an exception occurs and is caught, the transaction control can be handled in the exception section. If the exception is not caught, all statements before the exception are rolled back to the previous commit point.

Many examples throughout the book illustrate the use of anonymous blocks. Example 2-70 shows a simple anonymous block to illustrate the basic construct.

Example 2-70 Simple anonymous block

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

For more information about anonymous blocks, see the developerWorks topic *DB2 9.7: Using PL/SQL anonymous blocks in DB2 9.7*, which is available at:

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

You can also refer to the DB2 Information Center at:

http://publib.boulder.ibm.com/infocenter/db2luw/v10r5/topic/com.ibm.db2.luw.apdv.plsql.doc/doc/c0053848.html

Procedures compatibility

DB2 PL/SQL supports a wide range of Oracle PL/SQL features, syntactically and semantically. Within this compatible context, an Oracle PL/SQL procedure can be compiled and started directly on DB2.

Example 2-71 shows a PL/SQL procedure. With DB2 PL/SQL support, you can directly compile and run this procedure in DB2 without any modification.

Example 2-71 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 , INTICAP(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;
/
```

The DB2 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 of changes. For those few unavailable language elements you encounter, the PL/SQL code requires modification to provide the equivalent results in DB2.

Before DB2 10, nested routines (functions) should be implemented in DB2 as stand-alone database objects or inside PL/SQL packages. DB2 10 allows for local procedure definitions just as they are defined in Oracle.

In Example 2-72, the `nestedproc1` procedure contains the declaration of another procedure and one function.

Example 2-72 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 2-73 illustrates how you implement the nested routines from Example 2-72 in a PL/SQL package in a way that works on both DB2 and Oracle. localProc1 is allowed as local procedure (new in DB2 10) and localFunc2 is defined inside the package body, which makes it private to the package and makes these procedures correspond to the way they were nested in nestedproc1 in Example 2-72.

Example 2-73 DB2 nested procedure

```
CREATE OR REPLACE PACKAGE nestedpack1
IS
   FUNCTION localFunc2(p_arg4 IN VARCHAR2) RETURN VARCHAR2;
   PROCEDURE nestedproc1(p_arg1 IN VARCHAR2, p_arg2 OUT VARCHAR2);
END nestedpack1;
/

CREATE OR REPLACE PACKAGE BODY nestedpack1
IS
```

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

   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;

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

END nestedpack1;
/
```

Functions compatibility

The building blocks for PL/SQL functions are similar to the PL/SQL procedures, except, as noted earlier, regarding 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, because DB2 implementation details of procedures and functions are a bit different, in some exceptional cases, a handful of constructs that work in procedures might 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 to slightly syntax differences because of implementation differences.

Example 2-74 shows a simple Oracle recursive function (**REPORT_CHAIN**) that calls itself.

Example 2-74 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 delay the dependency checking that is done between compile time and run time by running the **set_routine_opts()** DB2 system procedure (Example 2-75). You can obtain the same result by setting the **AUTO_REVAL** database registry parameter to DEFERRED_FORCE. You can add a call to the **SYSPROC.ADMIN_REVALIDATE_DB_OBJECTS** administrative procedure to check the invalid object count and error messages after the full database deployment is finished. This call can catch any legitimate errors and fix them before you enable the production environment.

Example 2-75 DB2 recursive function

```
call set_routine_opts(‘VALIDATE RUN’)
/
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;
/
-- clear the registry variable or set a new option for the next
-- routine
call set_routine_opt('')
/
-- expect and ignore this type of message...
DB21034E  The command was processed as an SQL statement because it was
not a
valid Command Line Processor command.  During SQL processing it
returned:
SQL20481N  The creation or revalidation of object
"DB2INST1.REPORT_CHAIN"
would result in an invalid direct or indirect self-reference.  LINE
NUMBER=32.
SQLSTATE=429C3

-- ... after all code is deployed, run the command below to revalidate
all objects
call SYSPROC.ADMIN_REVALIDATE_DB_OBJECTS(NULL,NULL,NULL)
/
-- check the invalid object count
select count(1) from syscat.invalidobjects
/
-- which should be 0 !
```

Pipelined table functions

The purpose of pipelined table functions is to incrementally produce a result set for consumption on demand. A pipelined table function computes a table one row at a time and can be referenced in the **`FROM`** clause of **`SELECT`** statements. To enable this functionality, the **`PIPE`** statement was introduced. It enables user-defined functions to become a pipelined table function. This statement can be embedded in a compound SQL (compiled) statement of an SQL table function. It is not an executable statement and cannot be dynamically prepared.

Example 2-76 shows how to set up a pipelined table function and how it can be used by your application code.

Example 2-76 Pipelined table function

```
-- Create a type to hold the result set
CREATE TYPE datearray AS TABLE OF DATE;

-- Create the function as PIPELINED
CREATE FUNCTION dates(dt IN DATE) RETURN datearray PIPELINED AS
BEGIN
   LOOP
      PIPE ROW(dt);
      dt := dt + 1;
   END LOOP;
   RETURN;
END;
/

-- Use the function in a FROM clause.
SELECT * FROM TABLE(dates('2013-01-01')) WHERE ROWNUM <= 2;

-- Results
COLUMN_VALUE
2013-01-01 00:00:00
2013-01-02 00:00:00
```

External routines

External procedures and functions that are 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 are 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

To create a stored procedure that is written in C, complete the following steps:

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 export all global functions in the library. Here is an example of an AIX export file for the `outlanguage` procedure that exists in the file `spserver.sqc`:

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

   The `spserver.exp` export file lists the `outlanguage` stored procedure. The linker uses `spserver.exp` to create the shared library spserver that contains the `outlanguage` stored procedure.
3. On Windows operating systems, a DEF file is required, which has a similar purpose. See the `sqllib/samples/c/spserver.def` file 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 `sqllib/function` path.
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 accesses it. When you attempt 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 might 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 scripts are 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.

The following procedure creates and catalogs a stored procedure that is 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) the **`BONUS_INCREASE`** procedure is written.

1. Create and save the C source file (Example 2-77), with embedded SQL, as `outlanguage.sqc`.

   Example 2-77 Stored procedure in C

   ```
   #include <stdio.h>
   #include <string.h>
   #include <stdlib.h>
   #include <sqlda.h>
   #include <sqlca.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. Run the `bldrtn` build script 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. Run the script to catalog the procedure by running this command:

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

   DB2 responds with the following 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 the following command:

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

   The results are:

   ```
   Value of output parameters
   --------------------------
   Parameter Name  : LANGUAGE
   Parameter Value : SQL
   Return Status = 0
   ```

Building Java routines

To create an external Java user-defined function (UDF) from the DB2 command window, complete the following steps:

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

   ```
   javac your_javaFileName.java
   ```

2. Copy the `your_javaFileName.class` file to the `sqllib\function` directory on the Windows operating systems or to the `sqllib/FUNCTION` directory on UNIX.
3. Connect to the database by running the following 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 creates a Java UDF that retrieves the system name from the DB2 Registry variable DB2SYSTEM:

1. The Java source file that is shown in Example 2-78 is saved as `db2system_nameUDF.java`.

   Example 2-78 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 by running the following 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 by running the following command:

   ```
   db2 connect to db2_emp
   ```

6. Run the script to register the UDF with the database by running the following command:

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

7. Test the UDF by running the following command:

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

   The results are:

   ```
   ------------------------------------------------
   smpoaix

   1 record(s) selected.
   ```

2.1.11 PL/SQL packages

Both Oracle and DB2 support the concept of grouping database-related objects in a single unit, which is known in Oracle as a *package* and in DB2 as a *module*. In addition to its native modules, DB2 provides support for the full set of PL/SQL modular features that are extended to both user and system defined (built-in) packages. Therefore, the definition of a package or 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. This book uses the terms *module* and *package* interchangeably.

The following package features can be useful database objects:

- Extend schema support by grouping, 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 so 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 be referenced only 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 that are 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 that are defined within the package.
- Drop the package.
- Manage who can reference objects in a package by granting and revoking the EXECUTE privilege for the package.

User-defined packages

You can create user-defined packages in DB2 using the same syntax that you use in Oracle. As expected, these packages have the same structure and requirements. Similar to Oracle, DB2 user-defined packages have schema extensions 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. You can create a package specification 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 that are defined in the package specification (variable, constant, exception, function, and procedure) with the following 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, and 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 a basic package, called *c*, is in Appendix C, “Built-in modules” on page 321. This package example demonstrates the traditional PL/SQL functionality that 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 other functions.

Support for built-in packages

DB2 provides support for the most widely used built-in Oracle packages through system-defined modules (built-in packages). These packages include routines and procedures that can simplify the procedural and application logic. The packages provide 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 email.

Table 2-7 lists these packages and their descriptions.

Table 2-7 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 alternative 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_DDL	The DBMS_DDL module provides the capability to obfuscate DDL objects, such as routines, triggers, views, or PL/SQL packages.
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 email.
UTL_SMTP	Provides a set of routines for sending email using the Simple Mail Transfer Protocol (SMTP).

Detailed references about these packages, their method, and some examples are in Appendix C, “Built-in modules” on page 321. More detailed information about the built-in packages (system-defined module) is available in the DB2 Information Center at:

http://pic.dhe.ibm.com/infocenter/db2luw/v10r5/topic/com.ibm.db2.luw.apdv.sqlpl.doc/doc/c0053670.html

2.1.12 Triggers

A trigger is a database object that consists of set of SQL statements that are 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 various actions, for example, updates to other tables, automatically generating or transforming values for inserted or updated rows, or started functions to perform tasks, such as issuing alerts.

Triggers can be fired once for the FOR EACH ROW or FOR EACH STATEMENT triggers, and before, instead of, or after the triggered operation occurs. In PL/SQL, DB2 supports FOR EACH ROW BEFORE and AFTER triggers.

You can use Oracle to define a single trigger that can contain triggered actions for **INSERT**, **DELETE**, and **UPDATE** actions on the table, which is known as a *multi-action trigger*.

Example 2-79 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. DB2 supports the **SELECT** statement on the same table on which the trigger is defined, which Oracle does not.

Example 2-79 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;
/
```

When you create a trigger with the **CREATE TRIGGER** statement, you can define the following attributes:

- Include more than one operation in the trigger event clause. You can use **UPDATE**, **DELETE**, and **INSERT** operations together in a single clause. This capability means that the trigger is activated by the occurrence of any of the specified events. One, two, or all three trigger events can be arbitrarily specified in a **CREATE TRIGGER** statement. However, an operation cannot be specified more than once.
- Identify the event that activated a trigger. The trigger event predicates of **UPDATING**, **INSERTING**, and **DELETING** can be used as Boolean conditions for identifying trigger actions. Trigger event predicates can be used only in the trigger action of a **CREATE TRIGGER** statement that uses a compound SQL (compiled) statement.
- Use statement level triggers in PL/SQL. You can create triggers that fire only one time per statement regardless of the number of rows affected.

Example 2-80 demonstrates how to separate the multi-action trigger from Example 2-79 on page 100 into three separate triggers (one for each action of **INSERT**, **DELETE**, and **UPDATE**) for DB2 versions before DB2 9.10.

Example 2-80 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;
/
```

Often, inside of the trigger, there are common statements that are 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 preferred practice, extract complex logic from a trigger, place it in a procedure, and started the procedure from the trigger.

Oracle supports the enablement and disablement of triggers globally using the **`ALTER TRIGGER`** statement. In DB2, triggers can either be dropped and re-created instead, or global or package variables can be used to control whether the triggers are fired in a specific context.

DB2 has a feature that is equivalent (to a certain degree) to Oracle database triggers. Basically, you can instruct DB2 to run a stored procedure each time a new connection to the database is created. You can use this generic feature to configure database session environments by setting session parameters.

For more information, see the description for the **`connect_proc`** database configuration parameter at:

http://pic.dhe.ibm.com/infocenter/db2luw/v10r5/index.jsp?topic=%2Fcom.ibm.db2.luw.admin.dbobj.doc%2Fdoc%2Fc0057372.html

2.1.13 SQL statements

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

The TRUNCATE table SQL statement

DB2 includes a **`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 that the **`IMMEDIATE`** keyword is mandatory. When the `DB2_COMPATIBILITY_VECTOR` registry variable is set, the **`IMMEDIATE`** keyword is no longer required.

Example 2-81 shows the usage of **TRUNCATE**.

Important: If your database has the **BLOCKNOTLOGGED** parameter enabled, a **TRUNCATE** statement waits until the current backup operation on the table space where the table is defined finishes.

Because DB2 uses one thread per table space for the backup process, if you have a table space that is much larger than the other table spaces (as in the case of a staging table), the overall backup process time depends on the size of the largest table space. You cannot execute a **TRUNCATE** (or other operations that are not logged) during the online backup. If the **TRUNCATE** operation is part of your business logic, the **TRUNCATE** can negatively impact the transaction response time and can lead to timeouts in the client application.

Example 2-81 TRUNCATE table

```
CREATE TABLE trunct1(c1 int);
INSERT INTO truncT1 VALUES (1), (2), (3);
SELECT * FROM truncT1;
C1
-----------
1
2
3
3 record(s) selected.
TRUNCATE truncT1;
SELECT * FROM truncT1;
C1
-----------
0 record(s) selected.
```

Autonomous transaction

DB2 provides support for autonomous transaction, a mechanism that you can use to run a block of statements (or a separate transaction) independently of the outcome of the started transaction. This feature is useful when you move 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, 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** and **PRIOR**), and the **SYS_CONNECT_BY_PATH** scalar function.

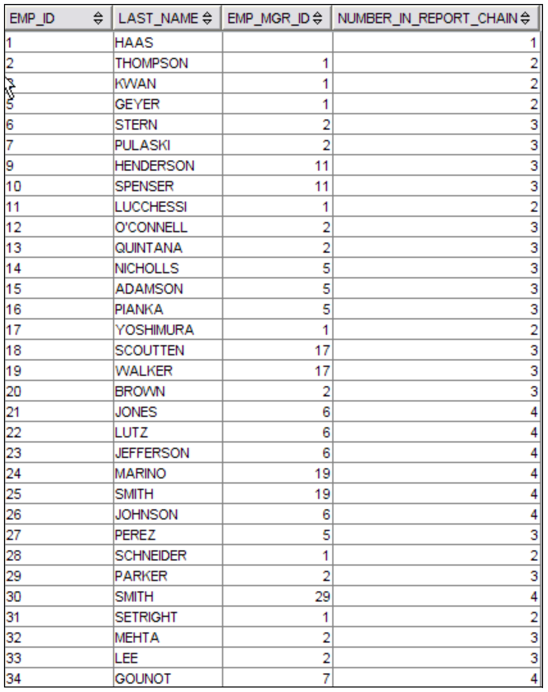
CONNECT BY recursion uses the same subquery for the seed (start) and the recursive step (connect). This combination provides a concise method of representing recursions, such as reports-to-chains presented in **CONNECT BY** recursion.

The query in Example 2-82 relies on the child-parent relationship between emp_id and emp_mgr_id in the 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 2-82 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 2-2 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
5	GEYER	1	2
6	STERN	2	3
7	PULASKI	2	3
9	HENDERSON	11	3
10	SPENSER	11	3
11	LUCCHESSI	1	2
12	O'CONNELL	2	3
13	QUINTANA	2	3
14	NICHOLLS	5	3
15	ADAMSON	5	3
16	PIANKA	5	3
17	YOSHIMURA	1	2
18	SCOUTTEN	17	3
19	WALKER	17	3
20	BROWN	2	3
21	JONES	6	4
22	LUTZ	6	4
23	JEFFERSON	6	4
24	MARINO	19	4
25	SMITH	19	4
26	JOHNSON	6	4
27	PEREZ	5	3
28	SCHNEIDER	1	2
29	PARKER	2	3
30	SMITH	29	4
31	SETRIGHT	1	2
32	MEHTA	2	3
33	LEE	2	3
34	GOUNOT	7	4

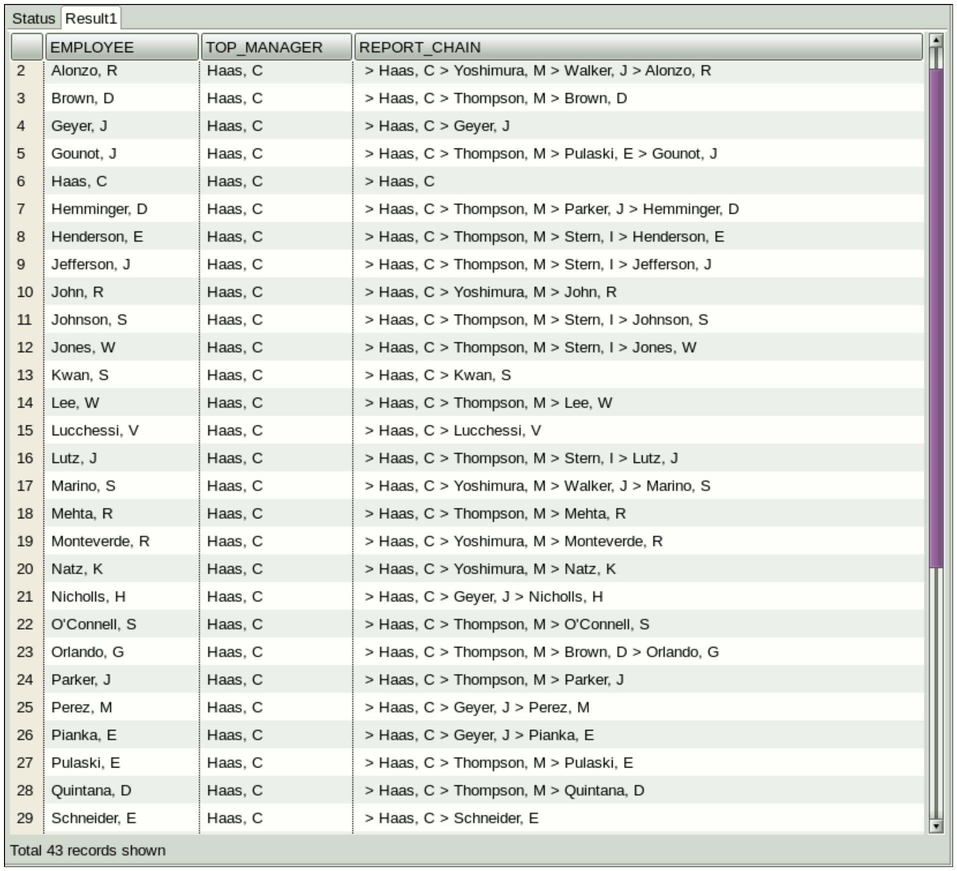
Figure 2-2 Output of a CONNECT BY recursive SQL statement

Example 2-83 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 run successfully on both Oracle and DB2.

Example 2-83 Demonstrating CONNECT_BY_ROOT and SYS_CONNECT_BY_PATH

```
SELECT
   (INITCAP(last_name) || ', ' || substr(first_name,1,1)) as employee,
   CONNECT_BY_ROOT (INITCAP(last_name) ||', '|| substr(first_name,1,1))
      as top_manager,
   SYS_CONNECT_BY_PATH ((INITCAP(last_name) ||', '||
   substr(first_name,1,1)),' > ') as report_chain
FROM employees
   START WITH band = '5'
   CONNECT BY nocycle 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 2-3. The names shown in Figure 2-3 are fictitious. These names are used for instructional purposes only.”


Status | Result1

	EMPLOYEE	TOP_MANAGER	REPORT_CHAIN
2	Alonzo, R	Haas, C	> Haas, C > Yoshimura, M > Walker, J > Alonzo, R
3	Brown, D	Haas, C	> Haas, C > Thompson, M > Brown, D
4	Geyer, J	Haas, C	> Haas, C > Geyer, J
5	Gounot, J	Haas, C	> Haas, C > Thompson, M > Pulaski, E > Gounot, J
6	Haas, C	Haas, C	> Haas, C
7	Hemminger, D	Haas, C	> Haas, C > Thompson, M > Parker, J > Hemminger, D
8	Henderson, E	Haas, C	> Haas, C > Thompson, M > Stern, I > Henderson, E
9	Jefferson, J	Haas, C	> Haas, C > Thompson, M > Stern, I > Jefferson, J
10	John, R	Haas, C	> Haas, C > Yoshimura, M > John, R
11	Johnson, S	Haas, C	> Haas, C > Thompson, M > Stern, I > Johnson, S
12	Jones, W	Haas, C	> Haas, C > Thompson, M > Stern, I > Jones, W
13	Kwan, S	Haas, C	> Haas, C > Kwan, S
14	Lee, W	Haas, C	> Haas, C > Thompson, M > Lee, W
15	Lucchessi, V	Haas, C	> Haas, C > Lucchessi, V
16	Lutz, J	Haas, C	> Haas, C > Thompson, M > Stern, I > Lutz, J
17	Marino, S	Haas, C	> Haas, C > Yoshimura, M > Walker, J > Marino, S
18	Mehta, R	Haas, C	> Haas, C > Thompson, M > Mehta, R
19	Monteverde, R	Haas, C	> Haas, C > Yoshimura, M > Monteverde, R
20	Natz, K	Haas, C	> Haas, C > Yoshimura, M > Natz, K
21	Nicholls, H	Haas, C	> Haas, C > Geyer, J > Nicholls, H
22	O'Connell, S	Haas, C	> Haas, C > Thompson, M > O'Connell, S
23	Orlando, G	Haas, C	> Haas, C > Thompson, M > Brown, D > Orlando, G
24	Parker, J	Haas, C	> Haas, C > Thompson, M > Parker, J
25	Perez, M	Haas, C	> Haas, C > Geyer, J > Perez, M
26	Pianka, E	Haas, C	> Haas, C > Geyer, J > Pianka, E
27	Pulaski, E	Haas, C	> Haas, C > Thompson, M > Pulaski, E
28	Quintana, D	Haas, C	> Haas, C > Thompson, M > Quintana, D
29	Schneider, E	Haas, C	> Haas, C > Schneider, E

Total 43 records shown

Figure 2-3 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 that are 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 by the **`FETCH FIRST`** ***`n`*** **`ROWS ONLY`** statement:

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

This DB2 syntax is equivalent (synonymous) to the previous ROWNUM example. Starting with DB2 10, the optimizer automatically considers the ROWNUM row limiting expressions for optimization, such as ROWNUM < 10 in the previous example, to match the native equivalent **`FETCH FIRST 9 ROWS ONLY`**. This optimization can help to optimize query plans based on the reduced number of rows that are returned by the query.

`UPDATE` and **`DELETE`** statements can also be executed on both Oracle and DB2, as shown in Example 2-84. 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 2-84 UPDATE and DELETE statements using 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;
```

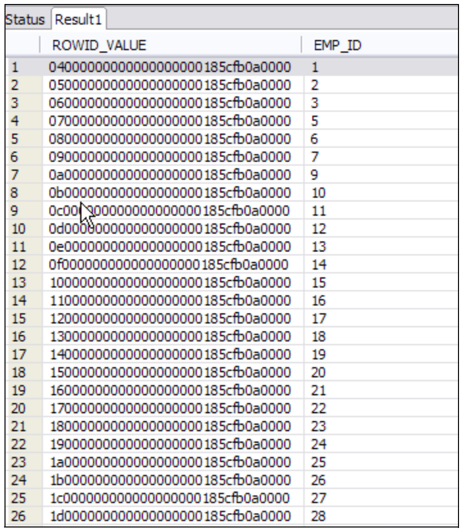
Row identifier (ROWID)

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. Figure 2-4 shows the ROWID from the EMPLOYEE table that is retrieved with the following query:

```
SELECT ROWID, emp_id FROM employees;
```



Status | Result1

	ROWID_VALUE	EMP_ID
1	04000000000000000000185cfb0a0000	1
2	05000000000000000000185cfb0a0000	2
3	06000000000000000000185cfb0a0000	3
4	07000000000000000000185cfb0a0000	5
5	08000000000000000000185cfb0a0000	6
6	09000000000000000000185cfb0a0000	7
7	0a000000000000000000185cfb0a0000	9
8	0b000000000000000000185cfb0a0000	10
9	0c000000000000000000185cfb0a0000	11
10	0d000000000000000000185cfb0a0000	12
11	0e000000000000000000185cfb0a0000	13
12	0f000000000000000000185cfb0a0000	14
13	10000000000000000000185cfb0a0000	15
14	11000000000000000000185cfb0a0000	16
15	12000000000000000000185cfb0a0000	17
16	13000000000000000000185cfb0a0000	18
17	14000000000000000000185cfb0a0000	19
18	15000000000000000000185cfb0a0000	20
19	16000000000000000000185cfb0a0000	21
20	17000000000000000000185cfb0a0000	22
21	18000000000000000000185cfb0a0000	23
22	19000000000000000000185cfb0a0000	24
23	1a000000000000000000185cfb0a0000	25
24	1b000000000000000000185cfb0a0000	26
25	1c000000000000000000185cfb0a0000	27
26	1d000000000000000000185cfb0a0000	28

Figure 2-4 ROWID values in DB2

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 works without changes. If you want to store a ROWID in a PL/SQL variable, create the following DB2 user distinct types:

```
CREATE DISTINCT TYPE ROWID AS VARCHAR(16) FOR BIT DATA WITH
COMPARISONS;
CREATE FUNCTION CHARTOROWID(VARCHAR(16) FOR BIT DATA)
       RETURNS ROWID SOURCE ROWID(VARCHAR());
CREATE FUNCTION ROWIDTOCHAR(ROWID)
       RETURNS VARCHAR(16) FOR BIT DATA SOURCE VARCHAR(ROWID);
```

Example 2-85 shows a way to store a ROWID in a PL/SQL variable.

Example 2-85 ROWID distinct type

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

SET SERVEROUTPUT ON
/
DECLARE
   x ROWID;
   ln VARCHAR2(30);
BEGIN
   SELECT ROWID INTO x FROM emp WHERE rownum=1;
END;
/
```

Outer join operator

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

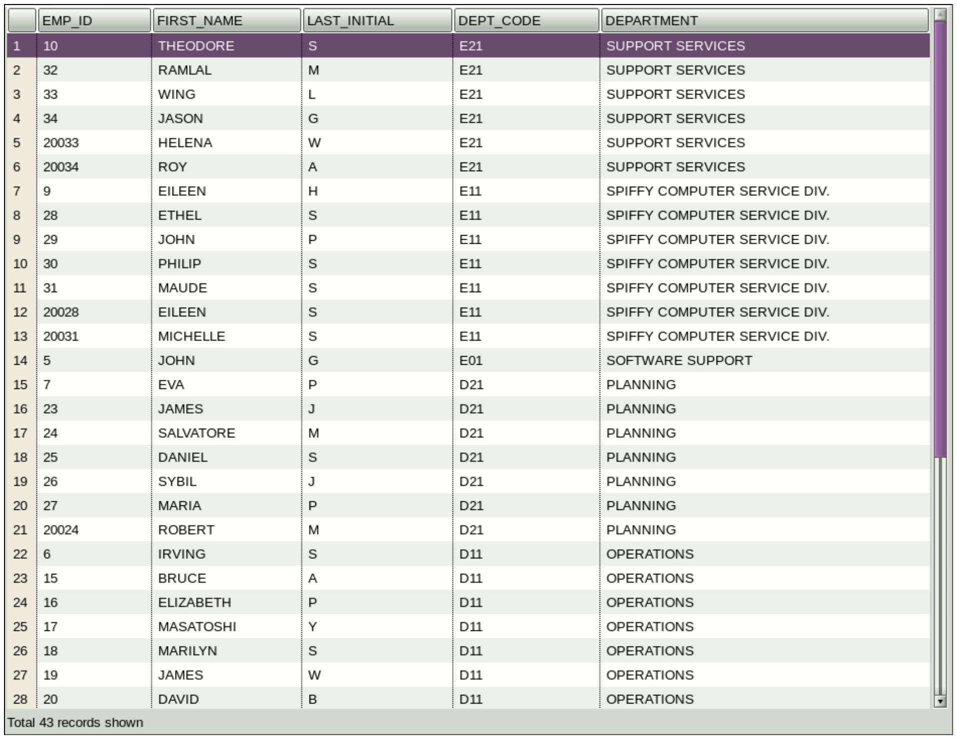
The outer join operator can be specified only within predicates of the **`WHERE`** clause on columns that are associated with table references specified in the **`FROM`** clause of the same subselect.

In Example 2-86, the right outer join correctly skips the records that have no corresponding entries, which demonstrates the outer join (+) operator syntax.

Example 2-86 Outer join

```
SELECT
   e.emp_id, e.first_name, substr(e.last_name, 1, 1) last_initial,
   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 2-5 shows the output.



	EMP_ID	FIRST_NAME	LAST_INITIAL	DEPT_CODE	DEPARTMENT
1	10	THEODORE	S	E21	SUPPORT SERVICES
2	32	RAMLAL	M	E21	SUPPORT SERVICES
3	33	WING	L	E21	SUPPORT SERVICES
4	34	JASON	G	E21	SUPPORT SERVICES
5	20033	HELENA	W	E21	SUPPORT SERVICES
6	20034	ROY	A	E21	SUPPORT SERVICES
7	9	EILEEN	H	E11	SPIFFY COMPUTER SERVICE DIV.
8	28	ETHEL	S	E11	SPIFFY COMPUTER SERVICE DIV.
9	29	JOHN	P	E11	SPIFFY COMPUTER SERVICE DIV.
10	30	PHILIP	S	E11	SPIFFY COMPUTER SERVICE DIV.
11	31	MAUDE	S	E11	SPIFFY COMPUTER SERVICE DIV.
12	20028	EILEEN	S	E11	SPIFFY COMPUTER SERVICE DIV.
13	20031	MICHELLE	S	E11	SPIFFY COMPUTER SERVICE DIV.
14	5	JOHN	G	E01	SOFTWARE SUPPORT
15	7	EVA	P	D21	PLANNING
16	23	JAMES	J	D21	PLANNING
17	24	SALVATORE	M	D21	PLANNING
18	25	DANIEL	S	D21	PLANNING
19	26	SYBIL	J	D21	PLANNING
20	27	MARIA	P	D21	PLANNING
21	20024	ROBERT	M	D21	PLANNING
22	6	IRVING	S	D11	OPERATIONS
23	15	BRUCE	A	D11	OPERATIONS
24	16	ELIZABETH	P	D11	OPERATIONS
25	17	MASATOSHI	Y	D11	OPERATIONS
26	18	MARILYN	S	D11	OPERATIONS
27	19	JAMES	W	D11	OPERATIONS
28	20	DAVID	B	D11	OPERATIONS

Total 43 records shown

Figure 2-5 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 2-87, 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 2-87 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 that is called DUAL, which is frequently used to retrieve system information. Although DB2 has its own equivalent that is called SYSIBM.SYSDUMMY1, support for calls to DUAL is 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. Because of 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 ignores the hints and no changes to the code are necessary.

The 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 at:

http://pic.dhe.ibm.com/infocenter/db2luw/v10r5/topic/com.ibm.db2.luw.admin.perf.doc/doc/c0060612.html

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. Unlike the **DELETE** statement, the **TRUNCATE** statement cannot be rolled back, which corresponds to the support provided by Oracle, Sybase, and Microsoft SQL Server. You can find a detailed discussion and an example in “The TRUNCATE table SQL statement” on page 103.

SELECT INTO statement with the 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 that consists of at most one row, and assigns the values in that row to host variables. The **FOR UPDATE** clause specifies that the selected row from the underlying table is 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 2-88 shows an **SELECT INTO** with **FOR UPDATE** example.

Example 2-88 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

DB2 schema compatibility features can help when you change schemas and the corresponding functions.

2.2.1 Extended data type support

When you provide extended data type support, you can use DB2 to 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 composed of year, month, and day) as a TIMESTAMP(0) data type (composed 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.

You can now run Oracle DDL to create tables directly in DB2, except for only a few data types, and it still requires correct mapping to the appropriate DB2 data types. For example, depending on the application requirements for number (38) data type, you might choose to change it to a large integer, decfloat (34), decimal (14.0), or simply to number (31).

Important: If you transfer Oracle DATE in DB2 TIMESTAMP(0), all the time parts of the TIMESTAMP data are filled with “0”. This automatic fill induces a lower selectivity for that field and can, in turn, induce suboptimal plans. If so, you must ensure that detailed statistics are collected for the column to provide the optimizer with correct input. Failing to do so can impact request response time and the overall database performance.

For more information about the new data types, see the following topics at the DB2 Information Center:

- VARCHAR2 data type:

  http://publib.boulder.ibm.com/infocenter/db2luw/v10r5/topic/com.ibm.db2.luw.apdv.porting.doc/doc/r0052880.html
- NUMBER data type:

  http://publib.boulder.ibm.com/infocenter/db2luw/v10r5/topic/com.ibm.db2.luw.apdv.porting.doc/doc/r0052879.html
- DATE data type based on TIMESTAMP(0):

  http://publib.boulder.ibm.com/infocenter/db2luw/v10r5/topic/com.ibm.db2.luw.apdv.porting.doc/doc/r0053667.html
- NCHAR data types:

  http://pic.dhe.ibm.com/infocenter/db2luw/v10r5/topic/com.ibm.db2.luw.sql.ref.doc/doc/r0057004.html

2.2.2 Flexible schema changes in DB2

The schema changes features in DB2 are designed to increase the availability, minimize database downtime, and simplify the administration tasks when schema changes are needed.

Maximum length of DB2 identifier names for schema objects

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

Altering 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. You can change the database with a minimal or no outage. This functionality, sometimes referenced as *schema evolution*, includes features, such as the ability to change column data types, rename a column or index, or add a default.

Altering objects

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

CREATE OR REPLACE allows for dynamic replacement for definitions of views, sequences, routines, and packages, which could be altered in any order and then be revalidated automatically the first time they are used. This situation 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 that you alter, both the view and the function definition are invalidated when the change in the table definition occurs. DB2 revalidates automatically both the function and the view the first time when they are called.

You can also revalidate the database objects manually by running a single database procedure named **ADMIN_REVALIDATE_DB_OBJECTS**. In the following example, all objects in the schema that is called MY_SCHEMA are revalidated:

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

To see different options of this procedure, see the DB2 Information Center at:

http://publib.boulder.ibm.com/infocenter/db2luw/v10r5/index.jsp?topic=/com.ibm.db2.luw.sql.rtn.doc/doc/r0053626.html

The revalidation option depends on a database configuration parameter that is called **AUTO_REVAL**, which is described in 2.1.1, “SQL compatibility setup” on page 22.

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 use the identity column functionality when you must automatically generate values for the table.

Sequence characteristics

The sequence includes the following characteristics:

- 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, two expressions are used to control the retrieval of the next value in the specified sequence and the retrieval of the value that is generated previous to the statement that is 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 you create a sequence that is called EMPLOYEE_SEQUENCE, see Example 2-89 for typical PL/SQL code calls to this sequence. This code can be run in both Oracle and DB2.

Example 2-89 Sequence

```
CREATE SEQUENCE EMPLOYEE_SEQUENCE
   MINVALUE 1
   MAXVALUE 999999999999999999999999999
   INCREMENT BY 1
   START WITH  2
   CACHE 20 NOCYCLE  NOORDER ;

CREATE PUBLIC SYNONYM EMPLOYEE_SEQUENCE  FOR SEQUENCE SALES.EMPLOYEE_SEQUENCE;
   SELECT employee_sequence.NEXTVAL INTO v_EmployeeId FROM dual;
   ...
   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 || ''')';
   ...
   DBMS_OUTPUT.PUT_LINE('Employee record id ' || v_EmployeeId || '
      was created successfully.');
```

For a complete sequence usage example, see the ADD_NEW_EMPLOYEE procedure in our test case procedure that is shown in Appendix C, “Built-in modules” on page 321.

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. In this section, we describe similarities and differences in the index implementation in Oracle and DB2.

Including a column index

The DB2 INCLUDE column index is a concept that is also available on some other database systems. When you create a unique index, you can include extra columns to the index using the **INCLUDE** clause. The INCLUDE columns are stored with the index but are not 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 2-90 shows the creation of an index with INCLUDE columns.

Example 2-90 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 the data of the index.

The DB2 clustering index provides performance enhancements when a query scans most of the data in the same order as the data 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. Each table can have only one clustered index because a table can be only in the same physical order as one index.

Example 2-91 provides an example statement that is used to create a clustering index.

Example 2-91 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 few key values (low cardinality), for example, gender or state. In DB2, this type of index is not required because the DB2 optimizer might create dynamic bitmap indexes during the execution of certain types of queries (if needed).

Indexing expressions

In Oracle, this functionality is known as a function-based index. It computes the value of the function or expression and stores it in the index. DB2 provides the exact same functionality. Here is its syntax:

```
CREATE INDEX emp_name ON emp(UPPER(name));
```

This feature can also be applied to multi-column indexes and it supports unique indexes and INCLUDE columns.

Partitioned table indexes

In DB2, the indexes that are created on partitioned tables can be both local and global. For more details about these indexes, see “Indexes on partitioned tables” on page 136.

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 that are specified in a unique constraint in DB2 must be defined as `NOT NULL`, while Oracle allows `NULL` values in a unique constraint. You overcome this issue in DB2 by defining a unique index on these columns instead of defining the unique constraint in the **`CREATE TABLE`** statement. Before DB2 10.5, you can have only one NULL key value in your unique index. With DB2 10.5, a new clause, `EXCLUDE NULL KEYS`, is introduced. This clause prevents NULL key values from being considered for the uniqueness of an index. A NULL key value in a multi-column index is defined as all key parts containing the `NULL` value.

In Oracle, you are not required to add a `NOT NULL` attribute to a column in a table 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 referential integrity, both DB2 and Oracle support foreign key constraints, in which a parent table's primary key is referenced by the child table. Additionally, you can use DB2 to enforce the referential integrity for any unique constraints (not just for the primary key). This constraint is especially useful when the unique constraint is a composite key and it can automatically apply more complex dependencies between the tables.

When you must perform operations, such as loading data into a table, altering a table by adding constraints, or adding a generated column, you usually want to temporarily disable the constraints, complete the operation, and then revalidate and enable the constraints. You can use the DB2 **`SET INTEGRITY`** mechanism to perform this manual integrity processing, as shown in Example 2-92.

Example 2-92 How to use SET INTEGRITY in DB2

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

For information about the options that are related to the **`SET INTEGRITY`** statement, see the DB2 Information Center at:

http://pic.dhe.ibm.com/infocenter/db2luw/v10r5/topic/com.ibm.db2.luw.sql.ref.doc/doc/r0000998.html

DB2 offers one more type of constraint, which is 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 more verification of data. Rather, they are used to improve query performance.

2.2.6 Created global temporary tables

Oracle supports the concept of global temporary tables, which correspond to the created global temporary tables in DB2, and thus provide 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 type of user-defined temporary table in DB2. An application session can 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 truncates only the specific session and has no effect on the data of other sessions.
- Indexes can be created on temporary tables.

Example 2-93 shows how to create a created global temporary table in DB2 with the same syntax as in Oracle.

Example 2-93 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 2-94 on page 121 using the DB2 **LIKE** syntax. The results of inserting into and selecting from the global temporary table are the same in both examples.

Example 2-94 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, substr(last_name,1,1) last_initial
FROM Employees_Temp_table;
```

Example 2-95 shows the result of this script.

Example 2-95 Result set - select from a global temp table

```
EMP_ID  FIRST_NAME           LAST_INITIAL
------- -------------------- ------------
    23. JAMES                J
    26. SYBIL                J
    24. SALVATORE            M
 20024. ROBERT               M
    27. MARIA                P
     7. EVA                  P
    25. DANIEL               S
```

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. When you create a synonym in DB2 for a package, nickname, sequence, view, or another synonym, specify the type of the database object on which the synonym is defined. Note the following **`FOR SEQUENCE`** clause:

```
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 2-96 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 2-96 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 similar in Oracle and DB2. Depending on the SQL used in the view definition, most of the **CREATE VIEW** (**CREATE OR REPLACE VIEW**) statements can be run against DB2 without changes.

The organization_structure view, which is shown in Example 2-97, compiles on both Oracle and DB2. It uses the following SQL syntaxes:

- **CREATE OR REPLACE VIEW**
- Recursive SQL statement **START WITH ... CONNECT BY**
- **LEVEL** keyword
- **COALESCE**
- **INITCAP**
- NVL scalar functions
- An outer join syntax of (+)

It also contains a CASE statement that conditionally calls two functions from a package named project_package.

Example 2-97 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, 'Uknown') ,
  (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;
```

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 tables on demand or by schedule. The statement to create an MQT is semantically similar to that of creating an Oracle Materialized View with just a few syntax differences.

The performance gain is because of 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, see the DB2 Information Center at:

http://pic.dhe.ibm.com/infocenter/db2luw/v10r5/topic/com.ibm.db2.luw.admin.perf.doc/doc/c0005324.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 2-98, the `address_type` consists of attributes such as street, number, city, and state. If methods are created, you can use them 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. For details about what object types are supported in a PL/SQL context, see 2.1.2, "PL/SQL record and collection types" on page 27.

Example 2-98 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 that are supported in both Oracle and DB2. Because of the great similarity in concept and for simplicity, Example 2-99 outlines only the DB2 type declarations and does not provide a step-by-step comparison with Oracle.

Example 2-99 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 2-100 shows the creation of a type that is later used as a table column.

Example 2-100 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;

                      Data type                     Column
Column name           schema    Data type name      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. This section introduces various partitioning features that are provided by DB2 and compares these features to 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 further definition. However, DB2 can be designed to organize data in other ways. These different data organization schemes can be specified at the database or table level.

The following data organization methods are available on DB2:

- Table partitioning
- Database partitioning
- Multidimensional clustering
- Combining methods of data organization
- Rolling in and rolling out of data
- Oracle flashback-archive
- Indexes on partitioned tables

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 that are 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 2-101 shows the creation of 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 2-year period. The `MINVALUE` and `MAXVALUE` catch all values that fall below and above the defined ranges.

Example 2-101 DB2 table partitioning

```
CREATE TABLE orders
(
l_orderkey DECIMAL(10,0) NOT NULL,
l_partkey INTEGER,
l_suppkey INTEGER,
l_linenumber 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 2-102 illustrates table partitioning using a longer syntax, which is required when the partitioning key is composed of a composite column.

Example 2-102 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 range partitioning is conceptually comparable to table partitioning in DB2. The differences between them lay mainly in the syntax that is used to define how the table is partitioned.

The equivalent code of Example 2-102 in Oracle is the range partition statement that is shown in Example 2-103.

Example 2-103 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 );
```

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** keywords.

On Oracle, each partition contains values less than, and not including, the value that defines that partition. On DB2, the values that are 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, you can use table partitioning on a generated column in a similar way as the list partitioning on Oracle.

Example 2-104 shows an Oracle list partitioning.

Example 2-104 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 2-105 shows how the Oracle list partitioning can be written as a DB2 table partitioning based on a generated column.

Example 2-105 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 2-105 on page 129, 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 feature is an optional DB2 feature that is 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. 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 be 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.

Table 2-8 summarizes the differences between the Oracle and DB2 partitioning methods.

Table 2-8 Mapping Oracle data organization schemes to DB2

Oracle partitioning	DB2 data organization	Oracle 10g syntax	DB2 syntax
No equivalent	Round-robin	None	Default: Occurs automatically on a 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, and multi-dimensional clustering	**PARTITION BY RANGE, SUBPPARTITION BY HASH,** and **SUBPARTITION BY LIST**	**DISTRIBUTE BY HASH, PARTITION BY RANGE,** and **ORGANIZE BY DIMENSIONS**
No equivalent	Multidimensional clustering	None	**ORGANIZE BY DIMENSIONS**

Example 2-106 shows a table that is defined when database partitioning is used.

Example 2-106 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 used only 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 2-107 shows an example of how hash partitioning syntax on Oracle compares to DB2.

Example 2-107 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 2-108 is an example of an MDC table definition.

Example 2-108 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 2-108, a table is created with the specification that division and quantity are organized by two dimensions, `STORE` and `SKU`. All data in the table is stored on disk in data blocks that are organized by the `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 a data organization scheme that is similar to the MDC table.

Combining methods of data organization

Just as Oracle has composite partitioning, various data organization schemes can be combined in DB2:

- 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 2-109 shows an example of combining database partitioning, table partitioning, and MDC organization.

Example 2-109 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 2-109, 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 the 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 2-110.

Example 2-110 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.

Rolling in and rolling 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 2-111 shows the newly created table DEC03 that is loaded with data rolled into the partitioned table STOCK.

Example 2-111 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** elements are needed to end the transaction and to make the table available for use.

In a similar way, an existing table can have a partition that is 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, you can modify partitioned tables using the **ADD PARTITION** and **DROP PARTITION** options of the **ALTER TABLE** statement. Use the **ADD PARTITION** clause to add an empty partition with a new range to an existing partitioned table. After it is added, load the partitioned table with data.

Oracle flashback-archive

DB2 10.1 introduces temporal tables or *Time Travel Query*. This feature matches (and exceeds) the Oracle flashback-archive feature. Temporal tables allow the database to manage data change history (also called data versioning).

The processes of setting up the Oracle flashback-archive and DB2 temporal tables features are different. However, if your application uses Oracle flashback-enabled queries, such as the **SELECT ... AS OF... / VERSIONS BETWEEN...** syntax, you can migrate this feature in DB2 with minor changes.

You can transparently enable temporal behavior for regular queries to retrieve data as of a given moment in time. This behavior can be useful when you are working with prepackaged or compiled applications, such as reporting tools that are unaware of temporal features.

Use the following syntax to set or clear the temporal parameter in the current session:

```
SET CURRENT TEMPORAL SYSTEM_TIME = <timestamp> | NULL
SET CURRENT TEMPORAL BUSINESS_TIME = <timestamp> | NULL
```

When used with the **`CONNECT_PROC`** database parameter, you can set the temporal parameters in the current session based on the distinct session parameters, such as user name. These parameters allow transparent shifting of reporting applications back in time and allow you to build reports as they were built in past. Web applications can also benefit from this feature to implement site versioning to timeline features without the need to change the existing code. When those parameters are not null, the client cannot change the temporal data. Thus, the application cannot change the historic data by mistake.

DB2 offers more flexibility in dealing with data versioning. When migrating Oracle flashback-enabled queries to DB2, you can choose between the system-period or application-period temporal tables. In each case Oracle flashback clause enabled queries can be translated, as shown in Example 2-112 through Example 2-119 on page 142.

Example 2-112 Oracle flashback AS OF query type sample

```
SELECT salary FROM employees
   AS OF ‘2011-02-28-09.10.12.64’
WHERE last_name = 'Smith'
```

Example 2-113 DB2 version of the above Oracle query example for System-period case

```
SELECT salary FROM employees
   FOR SYSTEM_TIME AS OF ‘2011-02-28-09.10.12.64’
WHERE last_name = 'Smith'
```

Example 2-114 DB2 version of the above Oracle query for Application-period case

```
SELECT salary FROM employees
   FOR BUSINESS_TIME AS OF ‘2011-02-28-09.10.12.64’
   WHERE last_name = 'Smith'
```

For more information about DB2 Time Travel Queries, go to the following address:

http://pic.dhe.ibm.com/infocenter/db2luw/v10r5/topic/com.ibm.db2.luw.admin.dbobj.doc/doc/c0058476.html

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 you perform roll-in operations with partitioned tables, the partitioned indexes are more efficient (less time and resource consuming) than the non-partitioned indexes.

You can find more details about the indexes on partitioned tables and the related topics in the DB2 Information Center at:

http://publib.boulder.ibm.com/infocenter/db2luw/v10r5/topic/com.ibm.db2.luw.admin.dbobj.doc/doc/c0055328.html

2.2.11 Oracle database links

A table that is in another database can be accessed as a local table through the features that are 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 Informix with DB2 Homogeneous Federation Feature. These features allow applications to access and integrate diverse data (mainframe and distributed) as though they were a DB2 table, regardless of where the information is, while they retain the autonomy and integrity of the data sources. The InfoSphere Federation Server adds to the Homogeneous Federation Feature to expand the choice of data sources to any data, including database management systems on various platforms, flat files, Excel, rich media, emails, XML, and LDAP. This method is an alternative one for moving data from an Oracle database to DB2.

For more information about InfoSphere Federation Server, see:

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

Setting up federated databases

The following example joins the LOCAL_DEPARTMENT table in the DB2_EMP database with the EMPLOYEE table in database SAMPLE. Because the query is executed on DB2_EMP, it is the federated server.

To set up a federated database, complete 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 2-115.

   Example 2-115 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
   ```

   After the **FEDERATED** value in DBM CFG is changed, it is applied after you restart the database server, as shown in Example 2-116.

   Example 2-116 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.WRAPPERS.

SERVER	A data source to a federated database. In our example, the data source is the SAMPLE database (see SYSCAT.SERVERS).
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.

Example 2-117 shows the **fed_config.db2** script, which is used to set up the federated system.

Example 2-117 The fed_config.db2 script

```
----------------------------------------------
-- create wrapper, server and user mapping
----------------------------------------------
--Create wrapper;
CREATE WRAPPER net8;
--
SELECT * FROM syscat.wrappers;
--Create databasse server definition
--You need the node_name from your tnsnames.ora file
CREATE SERVER fedserver type oracle version 8.1.7
WRAPPER net8
OPTIONS (NODE 'node_name');
--
SELECT * FROM syscat.servers
-- Map user
CREATE USER MAPPING FOR USER
SERVER fedserver
OPTIONS( REMOTE_AUTHID 'orausr', REMOTE_PASSWORD 'orausr');
--
SELECT * FROM syscat.usermappings;
```

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.

Accessing the remote table

After you configure the federation server, you can access the remote Oracle tables by using the Oracle database link syntax:

```
SELECT employee_id, last_name FROM hr.employees@fedserver;
```

For your existing PL/SQL code, that means that you do not have to change your calls to tables in remote databases anymore. This mechanism even works when you convert your remote database to DB2: You define a different wrapper, server, and user mapping for your remote DB2 database.

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 that is 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 2-9 lists the DB2 supported Data Dictionary views. The * notation applies to prefix DBA, USER, and ALL in front of each name.

Table 2-9 Oracle dictionary view supported in DB2

Category	View
General	► *_CATALOG ► *_DEPENDENCIES ► *_OBJECTS ► *_SEQUENCES ► DBA/USER_TABLESPACES ► DICTIONARY ► DICT_COLUMNS

Category	View
Tables/View	► *_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	► *_PROCEDURES ► *_SOURCE ► *_TRIGGERS ► *_ERRORS
Security	► 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 2-118. 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 schema. The string can be used later for recompiling, sending an email notification, and other maintenance operations.

Example 2-118 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 2-119 loops through the source code for database objects that are stored in the ALL_SOURCE Data Dictionary view. The application code prepares an SQL statement later for a dynamic execution that is based on object name and owner.

Example 2-119   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 that are 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 various 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 started 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 the same, the information is common, and both return dynamic data.

For example, to obtain information about applications that are connected to the database form 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 useful. One useful view is SYSENV_SYS_RESOURCES, which provides system information, such as memory, processor, 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 2-10 shows some of the V$ views and the equivalent administrative views or table functions.

Table 2-10 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

Oracle	DB2
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 cataloged in the SYSIBMADM schema, and the table functions are cataloged in the SYSPROC schema. The SELECT privilege is required to access these objects.

The Snapshot administrative views or equivalent table functions are 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 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. The DBA can then later query against these tables to generate trend usage, and use that data to maintain complete control over the system.

In DB2, you can use the MON_GET_PKG_CACHE_STMT table function in a script to help monitoring.

For more information about DB2 administrative views and table functions, see *Administrative SQL Routines and Views*, SC10-4293.

2.3 DB2 command-line utilities

DB2 provides the following command-line tools:

- The traditional command line processor (CLP)
- 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, run database utilities, issue SQL statements, run 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 The command line processor plus user interface

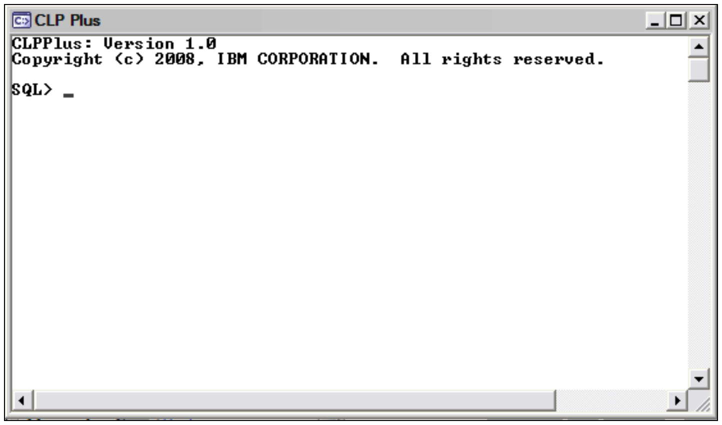
CLPPlus is a command-line user interface that provides a complement to the functionality provided by the CLP. It offers advanced options, including developing and editing database objects using an SQL buffer, calling operating system and database management commands, compiling and running procedures, functions and packages, creating and formatting SQL type reports from the command line, and so on.

CLPPlus is compatible with the Oracle SQL*Plus utility. CLPPlus is designed as an advanced command-line tool that provides compatibility for DBAs and application developers who have grown accustomed to the Oracle SQL*Plus interface. In this case, you might prefer using the DB2 CLPPlus interface when you work with DB2 databases. You find many familiar options, such as output formatting and developing in the SQL buffer. You can run a SQL*Plus script that is taken from Oracle in the DB2 CLPPLus with little or no modification.

To start CLPPlus, enter `clpplus` from the Windows command prompt or UNIX shell prompt.

In a Windows operating systems environment, you can also start CLPPLus by clicking **Start** → **All Programs** → **IBM DB2**, select your DB2 copy, and click **Command Line Tools** → **Command Line Processor Plus**.

Figure 2-6 shows DB2 CLPPlus in a Windows environment.



```
CLPPlus: Version 1.0
Copyright (c) 2008, IBM CORPORATION.  All rights reserved.

SQL> _
```

Figure 2-6 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:

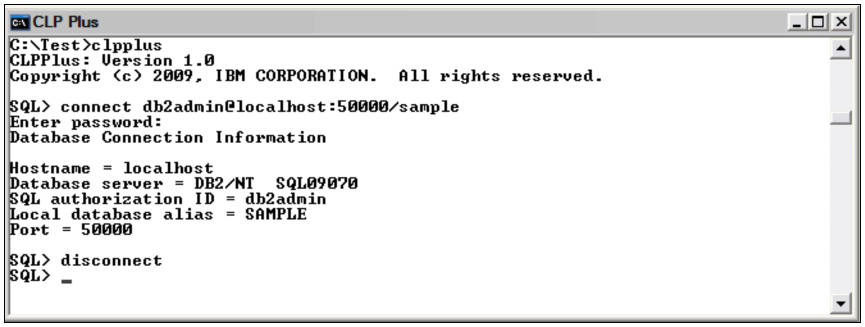
```
SQL> HOST ls | grep plsql.txt
```

As an example on a Windows system, from within CLPPlus, you can run the **`HOST`** command followed by the **`ipconfig`** command to display the Windows IP configuration as follows:

```
SQL> HOST ipconfig
```

CLPPlus can connect to any DB2 database without the need to catalog the database before connecting. Use the data server host name, port number, and database name along with your user credentials.

In Figure 2-7, user DB2ADMIN connects to a local database named SAMPLE that is listening on port 50000. To disconnect from the database, use the **disconnect** command.

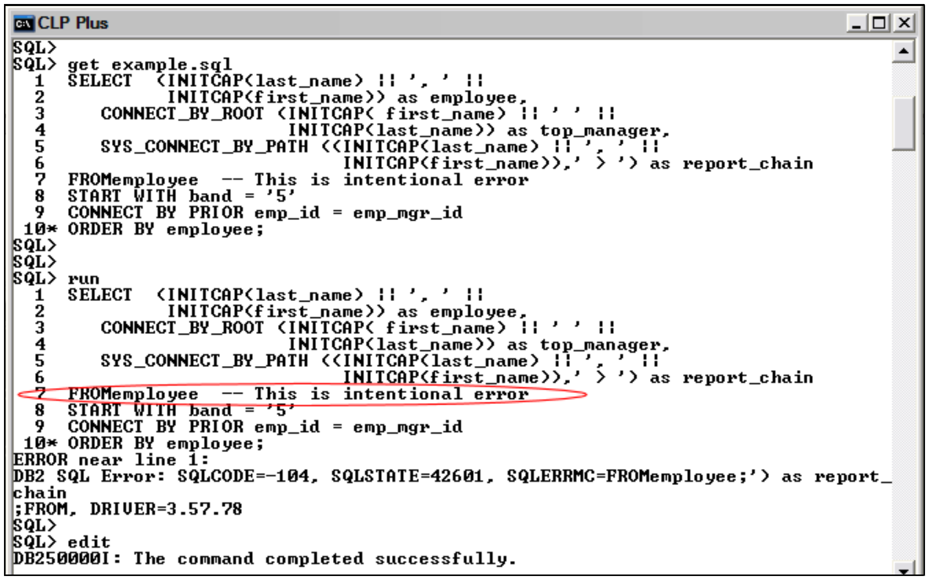


```
C:\Test>clpplus
CLPPlus: Version 1.0
Copyright (c) 2009, IBM CORPORATION.  All rights reserved.

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 2-7 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.

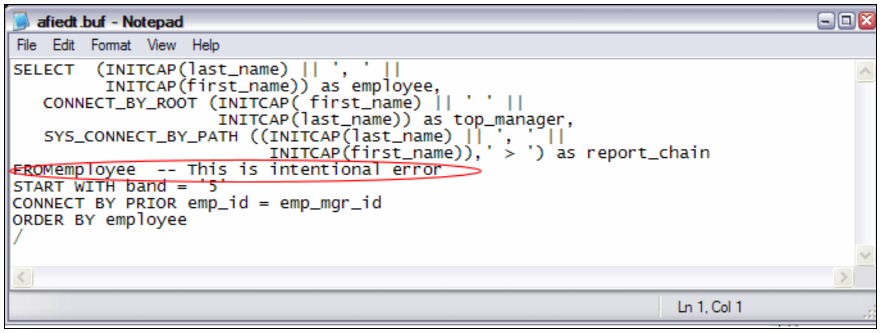
In Figure 2-8 on page 148, first load some PL/SQL code that is stored in a file using the **GET** command. Then, run this code straight from the buffer with the **RUN** command. For testing purposes, this example uses a code sample that has a syntax error in it to show how CLPPlus can help manage debugging and execution of database code. In this case, the **RUN** command fails with a syntax error that is properly displayed in the CLPPlus editor. (There is no space between the **FROM** clause and the table name.)



```
SQL>
SQL> get example.sql
  1  SELECT  (INITCAP(last_name) || ', ' ||
               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 2-8 Working with the SQL buffer - GET, RUN, and EDIT commands

To fix the intentional error, edit the code using the **EDIT** command. With the **EDIT** command, the buffer content is displayed in the preferred text editor (in this case, Notepad). You can correct the error and save the changes, as shown in Figure 2-9.



```
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
FROMemployee  -- This is intentional error
START WITH band = '5'
CONNECT BY PRIOR emp_id = emp_mgr_id
ORDER BY employee
/
```

Figure 2-9 Editing the code

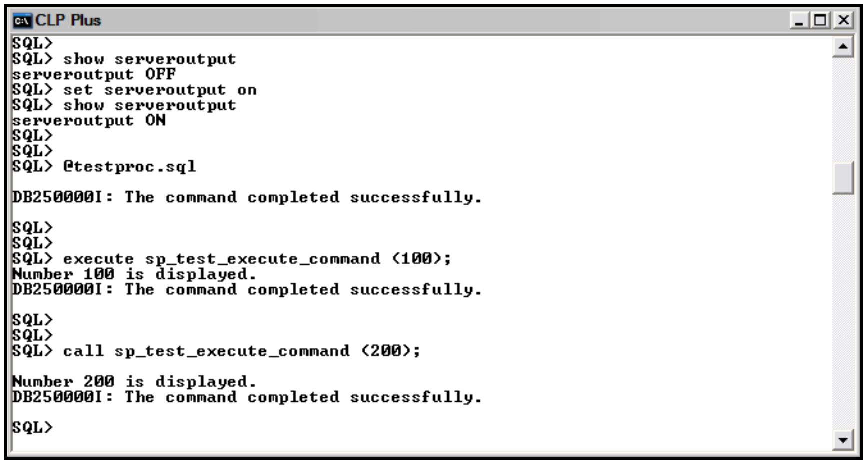
After the error is corrected (by adding space between the **FROM** clause and the table name), the SQL buffer is updated with the new version of the SQL script. The script is then ready to be used after you close the text editor. If you run the script again, the query succeeds and displays the result set in the CLPPlus window.

You can execute the PL/SQL procedures in CLPPlus by using the **EXECUTE** command. Example 2-120 shows a simple PL/SQL procedure that is saved in the `example.sql` file.

Example 2-120 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.');
  END IF;
END;
 /
```

Figure 2-10 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 you run it, you must set **SERVEROUTPUT ON** to see these messages. First, check for the current setting of the **SERVEROUTPUT** parameter using the **SHOW** command. Then, set it to `ON`.

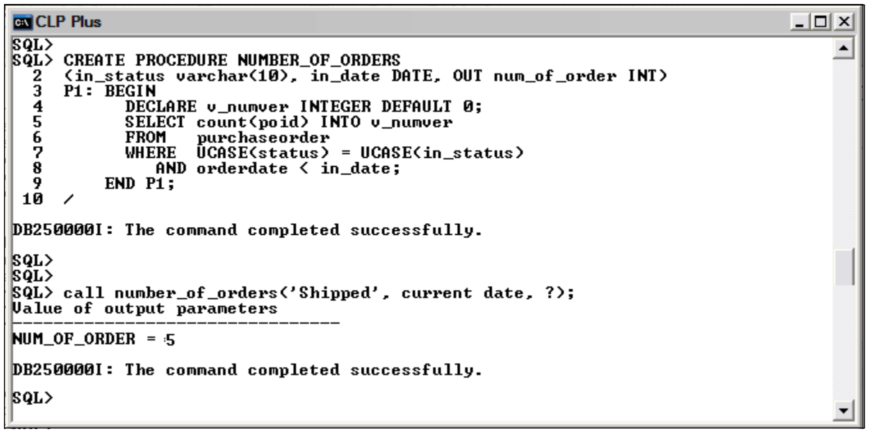


```
SQL>
SQL> show serveroutput
serveroutput OFF
SQL> set serveroutput on
SQL> show serveroutput
serveroutput ON
SQL>
SQL>
SQL> @testproc.sql

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 2-10 Executing PL/SQL procedure in CLPPlus

To execute the procedure, 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**.

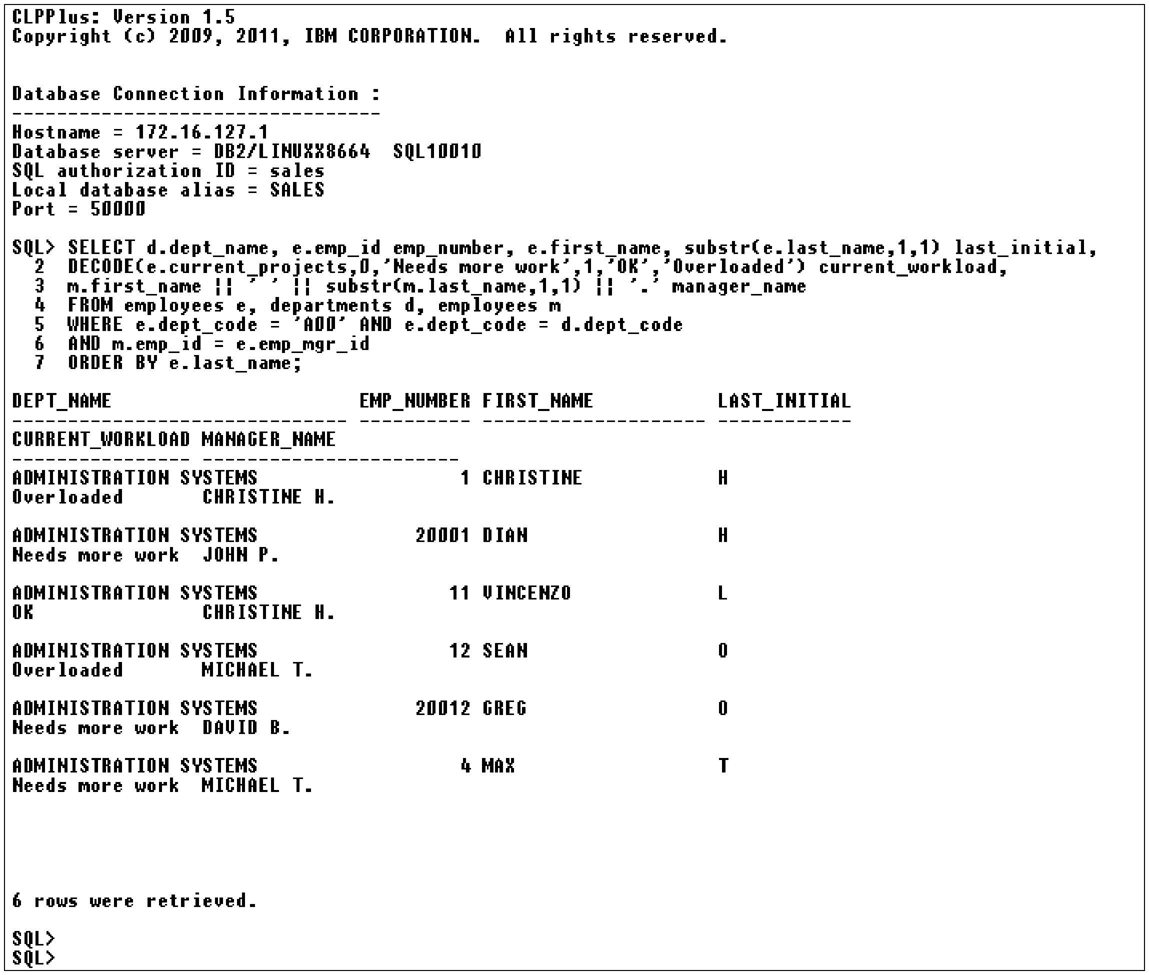
As shown in Figure 2-10 on page 149, you can use the DB2 syntax to call PL/SQL procedures in CLPPlus, and you also can create and run SQL PL procedures in CLPPlus. Support for SQL PL is another significant advantage of CLPPlus. In Figure 2-11, you create a SQL PL procedure using the **OUT** parameter in the SAMPLE database, call it, and display the output parameter in DB2 style. The result is 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  P1: 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 P1;
 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 2-11   Executing the SQL PL procedure with OUT parameters in CLPPlus

CLPPlus also provides numerous options for producing formatted reports dynamically. The query that is shown in Figure 2-12 on page 151 displays information about five employees with selected employee numbers. The output of this query, also shown in Figure 2-12 on page 151, is wrapped, not properly formatted (for example, the money fields do not have a currency sign) and, in general, is hard to read.



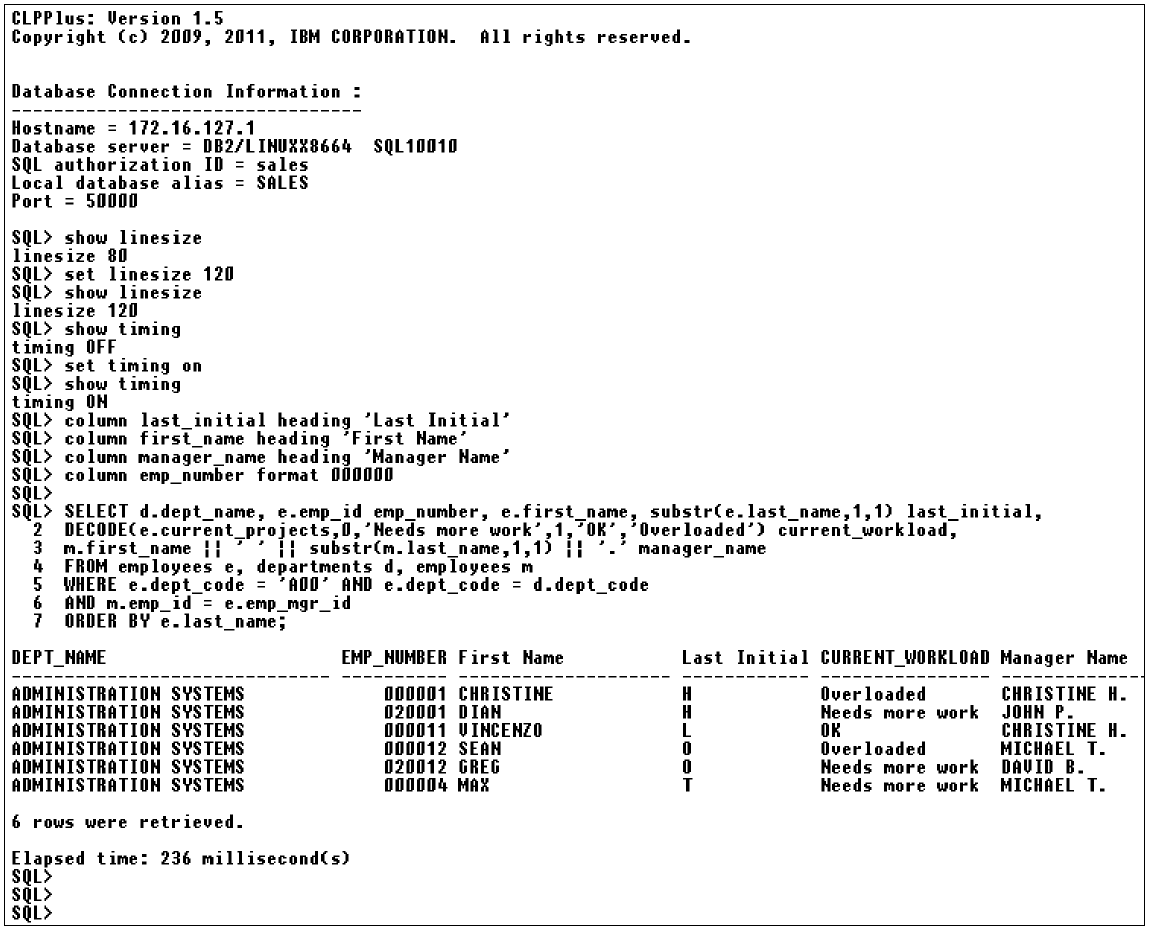
```
CLPPlus: Version 1.5
Copyright (c) 2009, 2011, IBM CORPORATION.  All rights reserved.


Database Connection Information :
---------------------------------
Hostname = 172.16.127.1
Database server = DB2/LINUXX8664  SQL10010
SQL authorization ID = sales
Local database alias = SALES
Port = 50000

SQL> SELECT d.dept_name, e.emp_id emp_number, e.first_name, substr(e.last_name,1,1) last_initial,
  2  DECODE(e.current_projects,0,'Needs more work',1,'OK','Overloaded') current_workload,
  3  m.first_name || ' ' || substr(m.last_name,1,1) || '.' manager_name
  4  FROM employees e, departments d, employees m
  5  WHERE e.dept_code = 'A00' AND e.dept_code = d.dept_code
  6  AND m.emp_id = e.emp_mgr_id
  7  ORDER BY e.last_name;

DEPT_NAME                     EMP_NUMBER FIRST_NAME           LAST_INITIAL
----------------------------- ---------- -------------------- ------------
CURRENT_WORKLOAD MANAGER_NAME
---------------- -----------------------
ADMINISTRATION SYSTEMS                 1 CHRISTINE            H
Overloaded       CHRISTINE H.

ADMINISTRATION SYSTEMS             20001 DIAN                 H
Needs more work  JOHN P.

ADMINISTRATION SYSTEMS                11 VINCENZO             L
OK               CHRISTINE H.

ADMINISTRATION SYSTEMS                12 SEAN                 O
Overloaded       MICHAEL T.

ADMINISTRATION SYSTEMS             20012 GREG                 O
Needs more work  DAVID B.

ADMINISTRATION SYSTEMS                 4 MAX                  T
Needs more work  MICHAEL T.




6 rows were retrieved.

SQL>
SQL>
```

Figure 2-12 Formatting in CLPPLus - part 1

Using the rich pallet of formatting options that are offered by CLPPlus, you can improve the appearance of this result set by changing the output settings and using the column formatting options.

In Figure 2-13, first check for the current settings by running the **SHOW** command. Then, provide new values for the output-related parameters with the **SET** command. The figure also shows how long it took to execute this query by running the **SET TIMING ON** command.

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.



```
CLPPlus: Version 1.5
Copyright (c) 2009, 2011, IBM CORPORATION.  All rights reserved.


Database Connection Information :
---------------------------------
Hostname = 172.16.127.1
Database server = DB2/LINUXX8664  SQL10010
SQL authorization ID = sales
Local database alias = SALES
Port = 50000

SQL> show linesize
linesize 80
SQL> set linesize 120
SQL> show linesize
linesize 120
SQL> show timing
timing OFF
SQL> set timing on
SQL> show timing
timing ON
SQL> column last_initial heading 'Last Initial'
SQL> column first_name heading 'First Name'
SQL> column manager_name heading 'Manager Name'
SQL> column emp_number format 000000
SQL>
SQL> SELECT d.dept_name, e.emp_id emp_number, e.first_name, substr(e.last_name,1,1) last_initial,
  2  DECODE(e.current_projects,0,'Needs more work',1,'OK','Overloaded') current_workload,
  3  m.first_name || ' ' || substr(m.last_name,1,1) || '.' manager_name
  4  FROM employees e, departments d, employees m
  5  WHERE e.dept_code = 'A00' AND e.dept_code = d.dept_code
  6  AND m.emp_id = e.emp_mgr_id
  7  ORDER BY e.last_name;

DEPT_NAME                      EMP_NUMBER First Name           Last Initial CURRENT_WORKLOAD Manager Name
------------------------------ ---------- -------------------- ------------ ---------------- -------------
ADMINISTRATION SYSTEMS             000001 CHRISTINE            H            Overloaded       CHRISTINE H.
ADMINISTRATION SYSTEMS             020001 DIAN                 H            Needs more work  JOHN P.
ADMINISTRATION SYSTEMS             000011 VINCENZO             L            OK               CHRISTINE H.
ADMINISTRATION SYSTEMS             000012 SEAN                 O            Overloaded       MICHAEL T.
ADMINISTRATION SYSTEMS             020012 GREG                 O            Needs more work  DAVID B.
ADMINISTRATION SYSTEMS             000004 MAX                  T            Needs more work  MICHAEL T.

6 rows were retrieved.

Elapsed time: 236 millisecond(s)
SQL>
SQL>
SQL>
```

Figure 2-13 Formatting in CLPPLus - part 2

CLPPlus is a powerful and functionally rich 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 help migration and ongoing maintenance that requires only a minimal investment in skill transfer.

To discover more about the CLPPlus command utility, go to the DB2 Information Center at the following website:

http://pic.dhe.ibm.com/infocenter/db2luw/v10r5/index.jsp?topic=/com.ibm.swg.im.dbclient.clpplus.doc/doc/c0056269.html

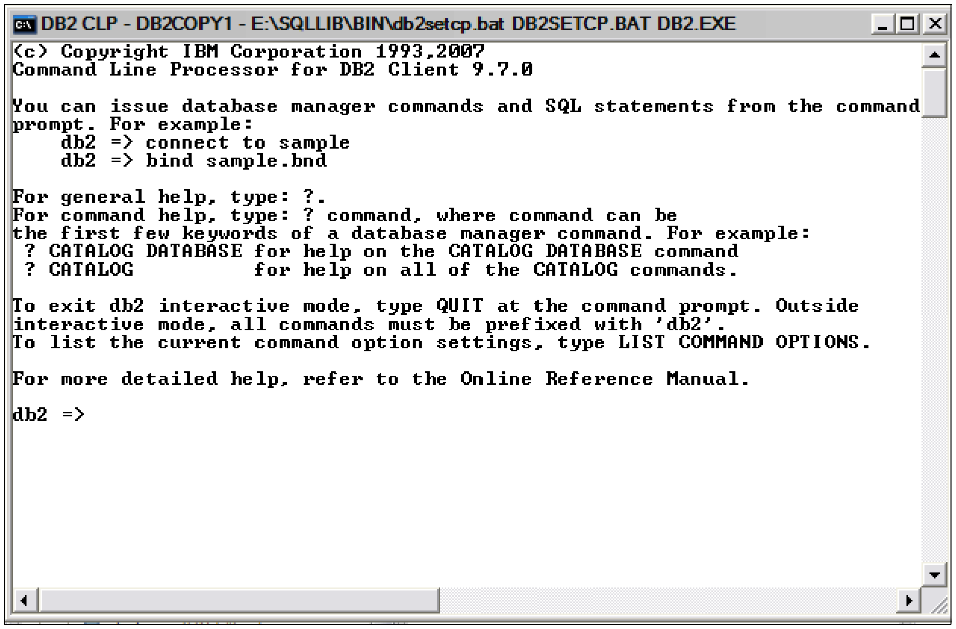
2.3.2 Using the DB2 command line processor

The DB2 command line processor (**CLP**) is a basic DB2 command tool with which most DB2 database administrators and developers are familiar. You use the CLP to run database utilities, SQL statements, and online help. It offers various command options. You can start the tool in interactive input mode, command mode, and batch mode.

From the enablement standpoint, you can use the CLP to run conversion scripts. The scripts can contain the definition of database objects, such as DDL, DML, and PL/SQL You can also include data movement commands, such as **IMPORT** or **LOAD**.

To start the CLP in a Windows operating system environment, click **Start** → **All Programs** → **IBM DB2**, select your DB2 copy, and click **Command Line Tools** → **Command Line Processor**. Another way to start the CLP is to select **Start** → **Run** and enter the **db2cmd** command at the command-line prompt.

Figure 2-14 shows the DB2 CLP in a Windows operating system 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 2-14 DB2 CLP

In a Linux or UNIX operating system environment, you can start the CLP by running **db2**. You also can run a statement directly on the operating system prompt by adding the `db2` prefix to the statement.

Run the DB2 SQL scripts by using the following flags:

```
db2 -tvf <scriptName>
```

Where

- The **-t** flag tells the CLP to use a semicolon (;) as the statement termination character.
- The **-v** flag stands for *verbose* so that the statements display before being executed.
- The **-f** flag tells DB2 that the next parameter is a script file name to execute.

Example 2-121 shows how to run a script to create a table using the CLP.

Example 2-121 Running a procedure with the CLP

```
db2inst1> cat sample_script1.sql

CREATE TABLE simple1
( 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 you execute scripts using the **db2** command with the **-t** flag, the default statement terminator is a semicolon (alternative terminators can be specified). Oracle uses the forward slash ("/") as the default statement terminator. 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 CLP to use a different statement terminator. For example, to use the forward-slash (/) as the statement terminator, run:

```
db2 -td/ -vf <scriptName>
```

Example 2-122 shows how to run a script with a forward slash using the **-td** command-line option.

Example 2-122 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 by setting the **SQLCOMPAT** mode on the command line before you run the script. Setting the mode on the command line before you run the script allows you to leave the script unchanged from its Oracle origins. This function is shown in Example 2-123, which run the script in Example 2-122 on page 155.

Example 2-123 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.
```

3

Conversion process and enablement tools

Database conversion is the process of transferring data between different storage systems, formats, or types. In simple terms, it is a technique for changing one type of data to another type through data extraction and data loading processes. However, depending on the programming language of the database vendors, conversion of the syntax is also required to ensure compatibility between the source and target database objects.

This chapter describes the conversion process according to the leading practices that are used by IBM consultants, and examines the tools and steps that are involved in an Oracle to IBM DB2 migration.

3.1 The conversion process

To provide the context for the description of enablement tools, it is prudent to first give an overview of the data conversion process.

The conversion process has multiple stages that can be divided into three broad phases:

- DDL extraction: The Data Definition Language (DDL) is the syntax for defining and altering data structures of a database. DDL statements are used to create, modify, and drop database objects, such as tables, indexes, users, and routines. DDL extraction is the process of obtaining the required DDL statements from the source database.
- Assessment and conversion: The assessment and conversion phase involves the critical task of examining the DDL of the source database to appropriately plan the conversion project. Traditionally, this phase involved manually reviewing and converting thousands of DDL statements. This phase is typically the most challenging and time consuming aspect of a database migration. To help streamline this process, IBM offers tools such as the DCW Compatibility Report and the DCW Auto-Convert feature, which greatly reduce the work effort required.
- Data movement: This is the last phase of the conversion process and usually occurs after all compatibility issues between the source and target database objects are resolved. Data movement involves replicating the appropriate source database objects on the target database, mapping the data from the source database, and then migrating the data to the target database.

Figure 3-1 shows a visual representation of the database conversion process from Oracle to IBM DB2 using the IBM Database Conversion Workbench.

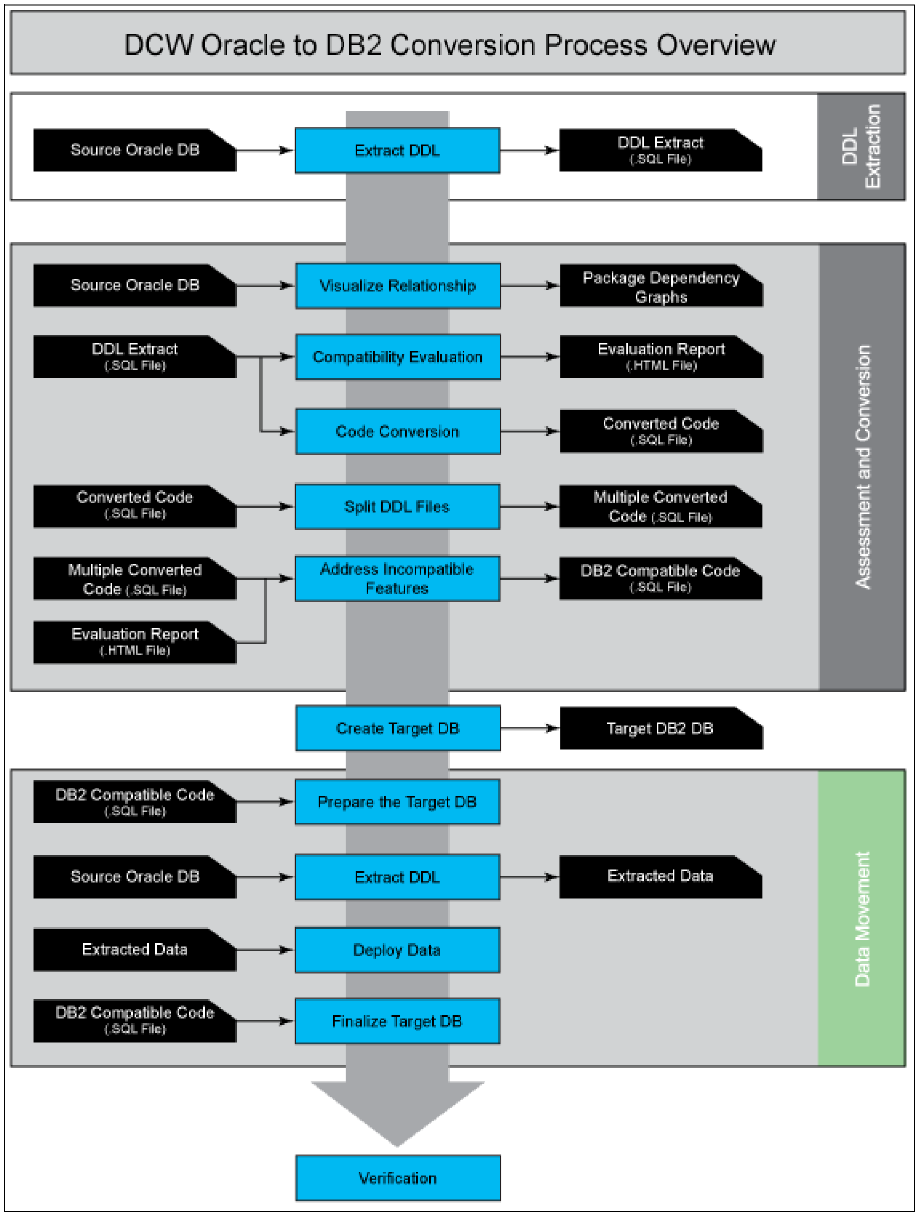

DCW Oracle to DB2 Conversion Process Overview
DDL Extraction
Source Oracle DB
Extract DDL
DDL Extract (.SQL File)
Assessment and Conversion
Source Oracle DB
Visualize Relationship
Package Dependency Graphs
DDL Extract (.SQL File)
Compatibility Evaluation
Evaluation Report (.HTML File)
Code Conversion
Converted Code (.SQL File)
Converted Code (.SQL File)
Split DDL Files
Multiple Converted Code (.SQL File)
Multiple Converted Code (.SQL File)
Evaluation Report (.HTML File)
Address Incompatible Features
DB2 Compatible Code (.SQL File)
Create Target DB
Target DB2 DB
Data Movement
DB2 Compatible Code (.SQL File)
Prepare the Target DB
Source Oracle DB
Extract DDL
Extracted Data
Extracted Data
Deploy Data
DB2 Compatible Code (.SQL File)
Finalize Target DB
Verification

Figure 3-1 Database conversion process - Oracle to IBM DB2

3.2 Enablement tools

Traditionally, converting from one database vendor to another vendor was a manual, difficult, and time-consuming task. But today, IBM provides a set of enablement tools that simplify and partly automate the process. The tools make extraction and deployment of schema definitions and data movement more manageable, even when there are tens of thousands of DDL and procedural objects.

This section covers the two most important enablement tools:

- IBM Data Studio: A unified software solution for database development and administration of IBM DB2 for Linux, UNIX, and Windows
- IBM Database Conversion Workbench (DCW): A plug-in to IBM Data Studio that combines multiple migration tools into a single, integrated environment, providing end-to-end assistance for the entire conversion process

3.2.1 IBM Data Studio

IBM Data Studio is a comprehensive data management tool in the IBM InfoSphere Optim family of data lifecycle tools and solutions. Based on the Eclipse platform, Data Studio provides an integrated, modular environment for database development and administration of DB2 for Linux, UNIX, and Windows. It also aids in application development with support for Java, XML editors, and other technologies. Data Studio enables heterogeneous database environments through its support of collaborative database development tools for IBM DB2 for z/OS, IBM DB2 for i, and IBM Informix.

Data Studio aims to support the entire database development lifecycle from design, through implementation and application development, to change management, administration, and maintenance.

Data Studio helps you perform many typical database development and administration tasks:

- Create, debug, and deploy DB2 SQL PL and PL/SQL procedures, functions, and triggers.
- Build, test, and tune queries, including access plan graphs.
- Manage database objects, analyze their dependencies, compare databases, and generate change scripts for schema synchronization.
- Integrate database development with collaboration tools, such as software version control systems.

Complete details about Data Studio, including an option to download it at no cost, are available at this address:

http://www-01.ibm.com/software/data/studio/

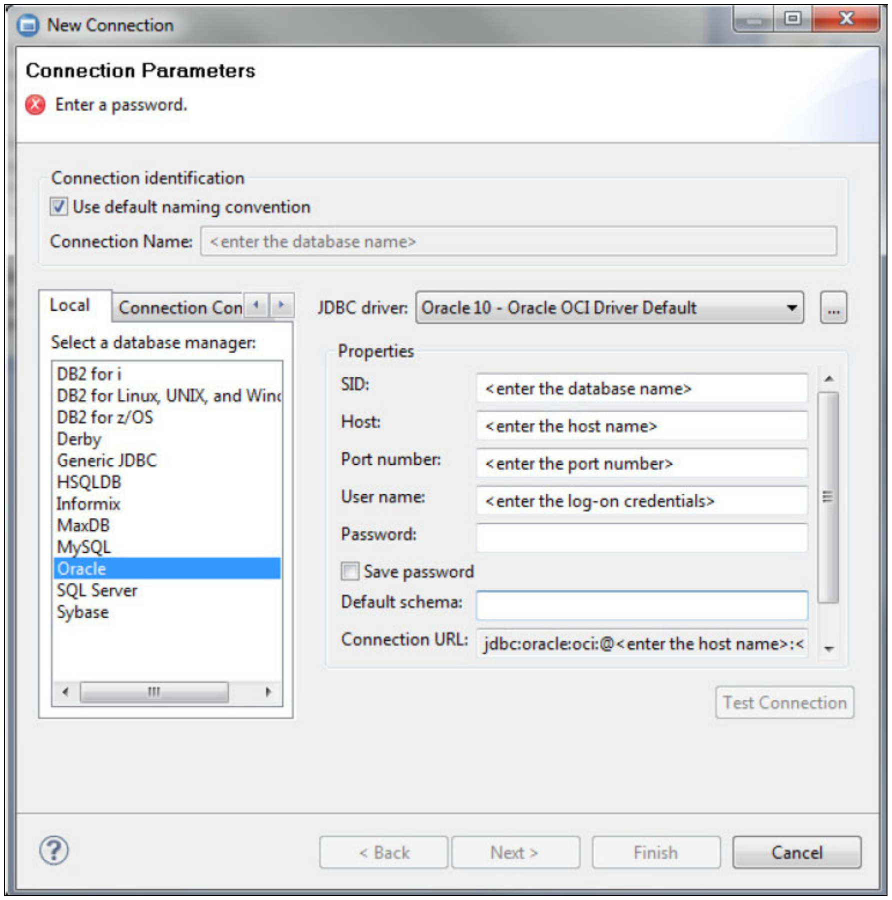
Data Studio has many advanced features that are outside the scope of this book. The following subsection provides information about one notable feature: creating the connection profiles that are needed to connect to the source and target databases.

Creating connection profiles

Data Studio provides a common interface to connect to different databases. You can create and manage multiple connection profiles for each database connection that you subscribe to.

The process involves entering basic server information and user credentials to create the connection profiles. Complete the following steps:

1. Open the Data Source Explorer view. Right-click **Database Connections** and then select **New**. This opens the New Connection wizard, which is shown in Figure 3-2.


New Connection

Connection Parameters

Enter a password.

Connection identification

[✓] Use default naming convention

Connection Name: <enter the database name>

Local | Connection Con

Select a database manager:

- DB2 for i
- DB2 for Linux, UNIX, and Winc
- DB2 for z/OS
- Derby
- Generic JDBC
- HSQLDB
- Informix
- MaxDB
- MySQL
- Oracle
- SQL Server
- Sybase

JDBC driver: Oracle 10 - Oracle OCI Driver Default

Properties

SID:	<enter the database name>
Host:	<enter the host name>
Port number:	<enter the port number>
User name:	<enter the log-on credentials>
Password:	
[ ] Save password	
Default schema:	
Connection URL:	jdbc:oracle:oci:@<enter the host name>:<

Figure 3-2 New Connection wizard

2. Choose the appropriate Database Manager from the list on the left side of the wizard window.
3. Enter the server and user credentials information in the fields on the right side.
4. Select the correct database driver from the **JDBC driver** drop-down list.

Note: If the driver you need is not shown in the list, you must download the required driver and specify the path.

5. Click **Test Connection** to confirm that your server and user credentials are correct.
6. Click **Finish** to complete the process. The connection profile is stored by Data Studio and can be accessed in the Data Source Explorer view.

3.2.2 IBM Database Conversion Workbench

The IBM Database Conversion Workbench (DCW) provides an end-to-end solution through an integrated workbench of tools to facilitate your conversion to DB2. DCW is a no additional charge plug-in that can be easily installed to add database migration capabilities to Data Studio. Whether you are converting to DB2 from another relational database management system (RDBMS) or migrating from one version of DB2 to another version, DCW provides an easy-to-use framework to take you through the conversion process.

DCW combines many of the tools that are used for database conversion into a single graphical user environment and follows processes that are based on several leading practices that are identified by IBM data migration consultants.

DCW provides the following benefits to the user:

- A common graphical interface with a uniform appearance for all phases of conversion
- A consolidated process with defined steps that are based on leading industry practices
- Faster data conversion by using automated tools
- Wizards that provide guidance through all the conversion steps

The conversion features in DCW can be used together to carry a user through all of the steps of a database conversion or, if the user prefers, each of the features can be used separately.

DCW offers several step-by-step functions to facilitate the conversion process:

- DCW Task Launcher: This integrated interface launches the various steps of the conversion process. The Task Launcher contains steps which, when clicked, launch the appropriate wizards and help topics.
- DDL Extraction: This function extracts the DDL of the objects in the source Oracle database. DDL is used by DCW to analyze the source database and provide compatibility assessment, conversion, and object creation on the target DB2 database. Various methods of extraction are available.
- Compatibility Evaluation: This function provides a report of the estimated compatibility (expressed as a percentage) of Oracle SQL and PL/SQL statements with DB2 10.1 or DB2 10.5. It outlines the major issues with the conversion and highlights code that must be fixed manually.
- Code Conversion: This function auto-converts known Oracle syntax to DB2 -compatible syntax. This helps streamline what can otherwise be a long process.
- Split DDL: This function splits a single DDL file into multiple files, which are organized by object types. This provides a more intuitive code organization by breaking down large DDL files into smaller and more manageable components.
- Package Visualizer: This function generates a dependency graph of objects in the source database.
- Data Movement: This function extracts and loads data from the source Oracle database to the target DB2 database. Various methods of extraction and loading are available.

Installation

Installation of DCW requires a working instance of Data Studio. For instructions about obtaining the Data Studio software, see 3.2.1, "IBM Data Studio" on page 162.

You can download the DCW plug-in and view supported versions of Data Studio at this address:

http://www.ibm.com/developerworks/data/ibmdcw

Note: Before preceding with a new installation of DCW, uninstall any previous versions of DCW that are installed in your instance of Data Studio.

To install DCW, complete the following steps:

1. Extract the DCW plug-in to your chosen local directory and then add the extracted directory to Data Studio. To so do, click **Help** → **Install New Software** to open the Install wizard. In the wizard, click **Add** and select **Local**, and then choose the extracted directory of the plug-in and specify a name, as shown in Figure 3-3.

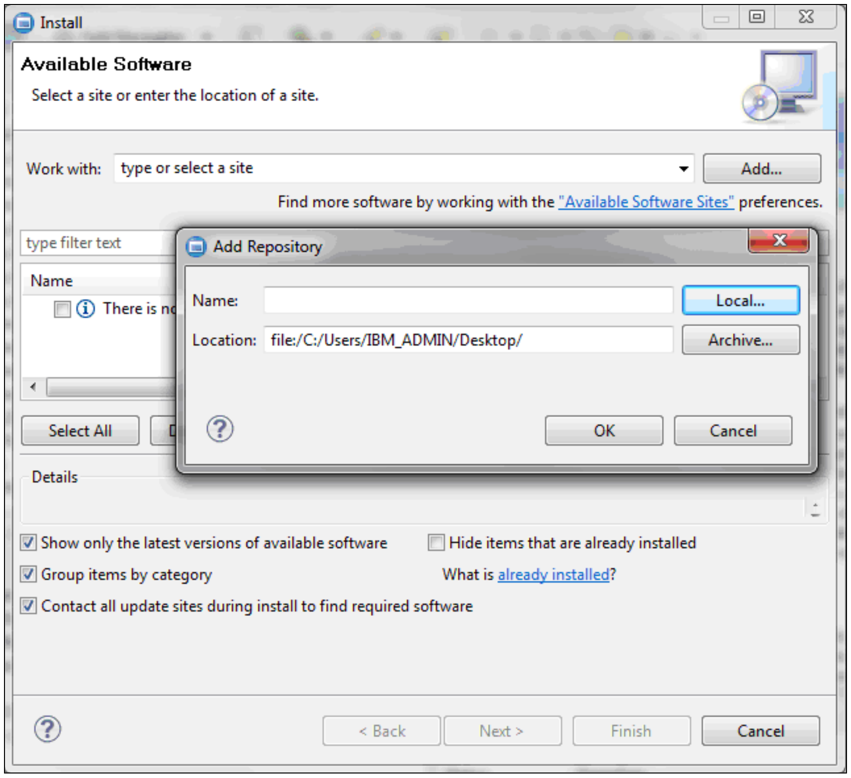
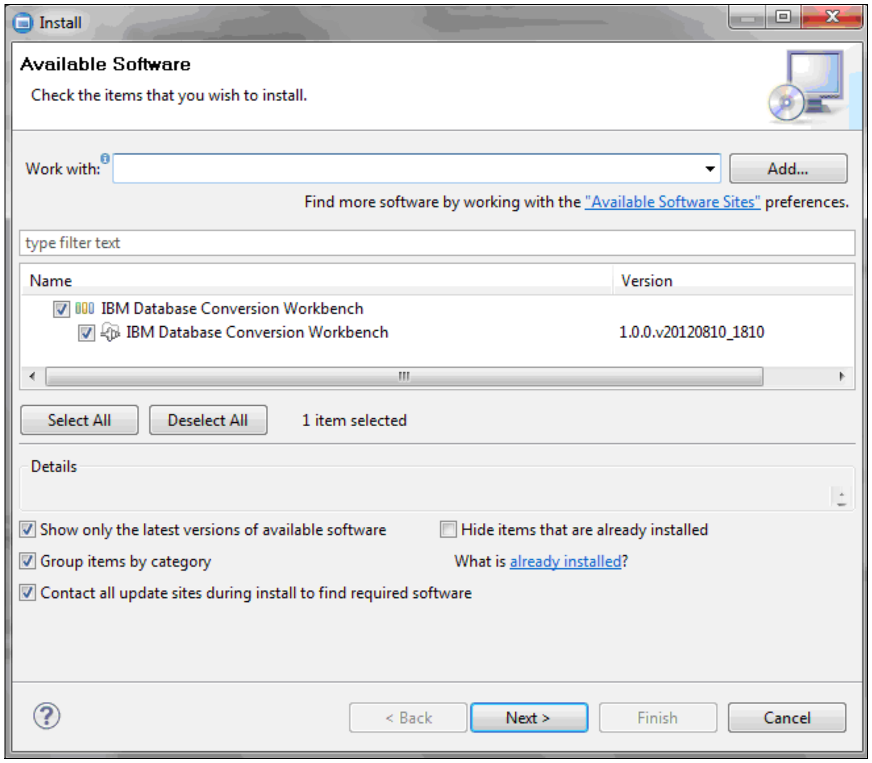

Install
Available Software
Select a site or enter the location of a site.
Work with: type or select a site
Add...
Find more software by working with the "Available Software Sites" preferences.
type filter text
Name
There is no
Select All
Details
Add Repository
Name:
Local...
Location: file:/C:/Users/IBM_ADMIN/Desktop/
Archive...
OK
Cancel
Show only the latest versions of available software
Hide items that are already installed
Group items by category
What is already installed?
Contact all update sites during install to find required software
< Back
Next >
Finish
Cancel

Figure 3-3 DCW installation - adding the download directory to Data Studio

Based on the directory that you selected, Data Studio automatically registers the DCW plug-in as available for installation.

2. Designate the plug-in for installation by selecting the **IBM Database Conversion Workbench** check box, as shown in Figure 3-4.



Install

Available Software

Check the items that you wish to install.

Work with:

Find more software by working with the "Available Software Sites" preferences.

type filter text

Name	Version
☑ IBM Database Conversion Workbench	
☑ IBM Database Conversion Workbench	1.0.0.v20120810_1810

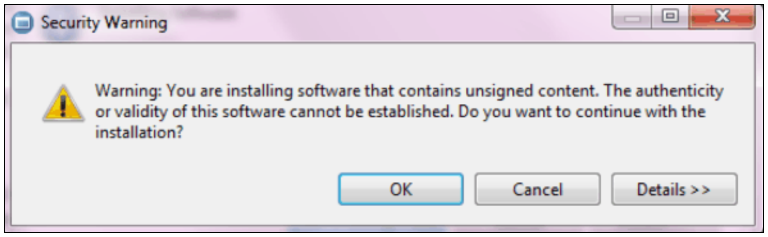
1 item selected

Details

☑ Show only the latest versions of available software
☐ Hide items that are already installed
☑ Group items by category
What is already installed?
☑ Contact all update sites during install to find required software

Figure 3-4 DCW-installation - selecting the plug-in

3. Click **Next** to initiate the installation. During the installation process, you are prompted to verify the terms of the software license and acknowledge a security warning (see Figure 3-5).



Security Warning

Warning: You are installing software that contains unsigned content. The authenticity or validity of this software cannot be established. Do you want to continue with the installation?

Figure 3-5 DCW installation - acknowledging the security warning

4. After the installation process is complete, select **Finish** to close the wizard.
5. Restart Data Studio to activate the newly installed DCW plug-in.

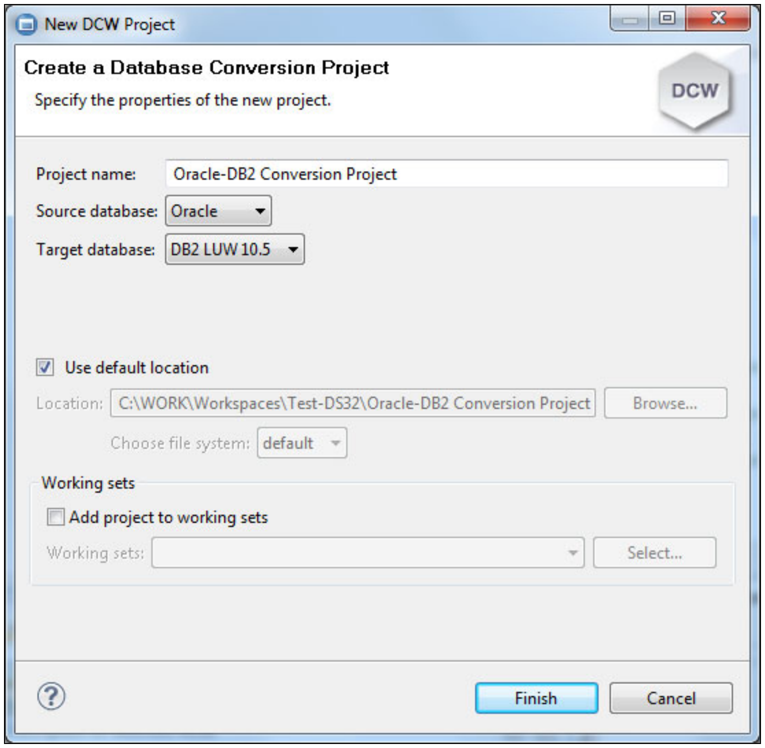
3.3 Getting started with DCW

The Database Conversion Workbench uses *conversion projects* to customize and store the data that is relevant to your migration. When you select your source and target database vendors, DCW automatically customizes your workspace with the applicable tools. DCW manages all of the files that are related to a migration under a single project folder, which allows you to easily manage multiple migration projects in parallel.

3.3.1 Creating a DCW project

To create a DCW project, from Data Studio, click **File** → **New** → **New DCW Project** to open the New DCW Project wizard.

You are prompted to specify a project name and to select the source and target databases from the drop-down lists, as shown in Figure 3-6. For an Oracle to DB2 migration, select **Oracle** as the source and the applicable DB2 database as the target.



Create a Database Conversion Project

Specify the properties of the new project.

Project name: Oracle-DB2 Conversion Project

Source database: Oracle

Target database: DB2 LUW 10.5

☑ Use default location

Location: C:\WORK\Workspaces\Test-DS32\Oracle-DB2 Conversion Project

Choose file system: default

Working sets

☐ Add project to working sets

Working sets:

Figure 3-6 Creating a DCW project in the New DCW Project wizard

Views and perspectives in Data Studio

Much of your work in Data Studio is performed in *views* and *perspectives*.

Upon creation of the DCW project, you mightbe prompted to change to Data Studio's Database Conversion perspective if this is your first time installing DCW. This perspective provides four major views to facilitate the conversion process:

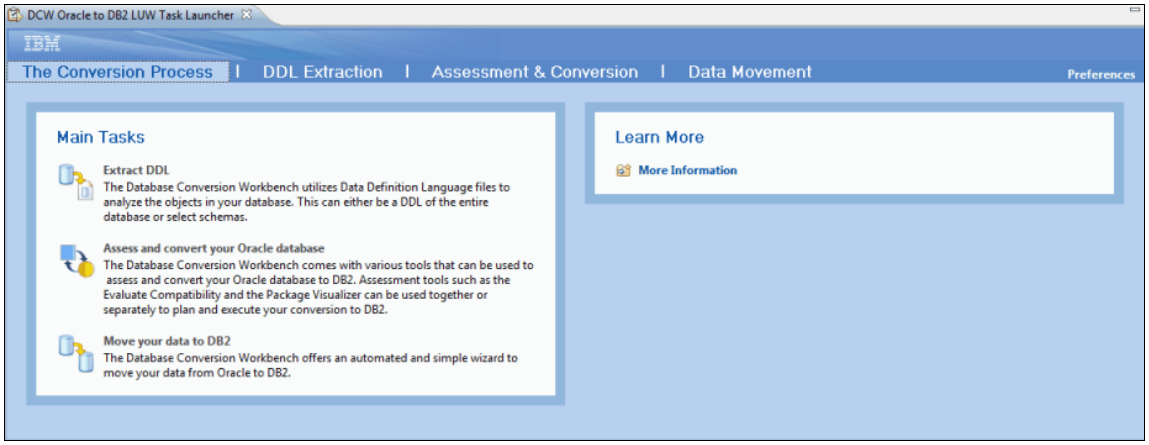
- Project Explorer view: Lists all projects and relevant files.
- Data Source Explorer view: Lists all existing database connection profiles.
- Administration Explorer view: Lists all existing database connection profiles and provides access to DB2 administration functions.
- Oracle to DB2 Task Launcher: Launches various wizards and help topics.

Connecting to the source database

After you create the DCW project, you must create a connection profile for your source Oracle database. To do this task, go to the Data Source Explorer view and complete the steps in“Creating connection profiles” on page 163.

3.3.2 DCW Task Launcher

The centerpiece of DCW is the DCW Task Launcher, which provides an easy-to-use interface to launch the various steps of the conversion process. The task launcher is customized depending on the source and target vendors that you selected when you created your DCW project. The task launcher starts automatically when a new conversion project is created or it can be accessed through the Help menu. Figure 3-7 shows the DCW Oracle to DB2 Task Launcher.


DCW Oracle to DB2 LUW Task Launcher
IBM
The Conversion Process | DDL Extraction | Assessment & Conversion | Data Movement
Preferences
Learn More
More Information

Main Tasks

Extract DDL
The Database Conversion Workbench utilizes Data Definition Language files to analyze the objects in your database. This can either be a DDL of the entire database or select schemas.

Assess and convert your Oracle database
The Database Conversion Workbench comes with various tools that can be used to assess and convert your Oracle database to DB2. Assessment tools such as the Evaluate Compatibility and the Package Visualizer can be used together or separately to plan and execute your conversion to DB2.

Move your data to DB2
The Database Conversion Workbench offers an automated and simple wizard to move your data from Oracle to DB2.

Figure 3-7 DCW Task Launcher

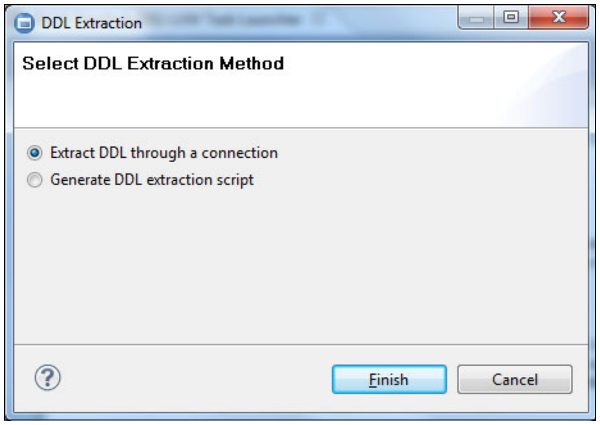
The task launcher provides a set of steps for your Oracle to DB2 migration. Selecting a step launches the appropriate wizard, along with help topics that provide step-by-step guidelines and useful hints to help your migration.

3.4 DDL extraction using DCW

The first step in any migration is DDL extraction. In this phase, the DDL of the source Oracle database is extracted and used by DCW to analyze the existing database objects.

DCW gives you two options for extracting the DDL from your source database. If you are able to connect to your source database, DCW can extract DDL through a connection. Alternatively, if the source database is not directly accessible through the system where DCW is installed, you can extract DDL by running extraction scripts manually on the source Oracle database. In both cases, DCW either generates or runs the commands and queries that are required to create the DDL. Both approaches use the functions of the source database and apply industry-leading practices.

You can choose your extraction method in the DDL Extraction wizard, as shown in Figure 3-8. The wizard can be accessed from the DCW Task Launcher or by right-clicking your DCW project folder in the Project Explorer view and then selecting **Database Conversion** → **DDL Extraction**.



[✓ Extract DDL through a connection]
Generate DDL extraction script

Figure 3-8 Choosing an extraction method in the DDL Extraction wizard

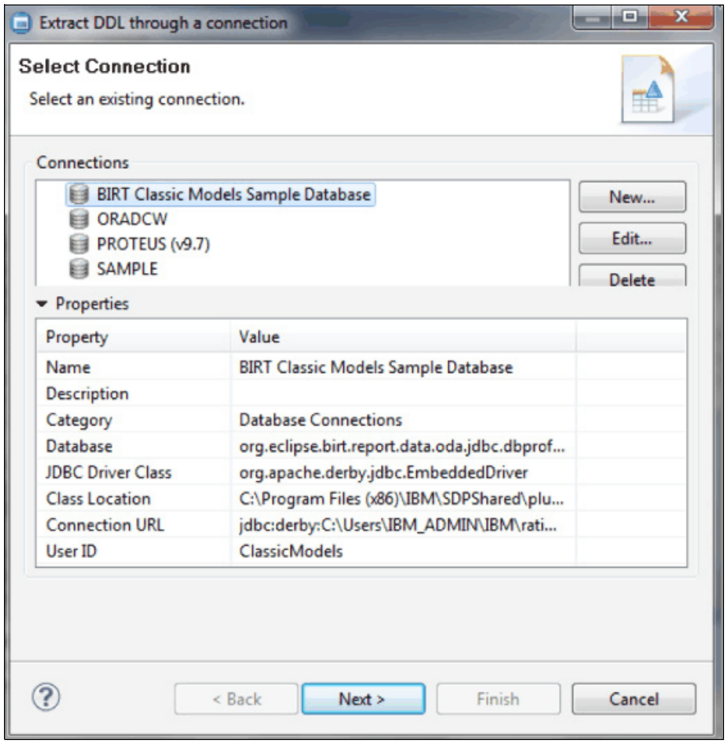
3.4.1 DDL extraction using a connection

Using a connection is the recommended method of DDL extraction, provided it is possible to directly connect to the source Oracle database through Data Studio. This process uses the connection profile for the source Oracle database that is created in “Connecting to the source database” on page 171.

Complete the following steps:

1. In the DDL Extraction wizard (which is accessed from the Oracle to DB2 Task Launcher), select **Extract DDL through a connection**.

2. Choose the connection profile of the source Oracle database, as illustrated in Figure 3-9. Click **Next**.



Property	Value
Name	BIRT Classic Models Sample Database
Description	
Category	Database Connections
Database	org.eclipse.birt.report.data.oda.jdbc.dbprof...
JDBC Driver Class	org.apache.derby.jdbc.EmbeddedDriver
Class Location	C:\Program Files (x86)\IBM\SDPShared\plu...
Connection URL	jdbc:derby:C:\Users\IBM_ADMIN\IBM\rati...
User ID	ClassicModels

Figure 3-9 Selecting the source database connection profile

3. On the next page of the wizard, select the schema that you want to extract. Only the schemas that are authorized for the user credentials that you used in the connection profile are listed, as shown in Figure 3-10. Click **Next**.

Recommendation: To reduce the risk of failure, it is a good idea to perform migrations on one schema at a time.

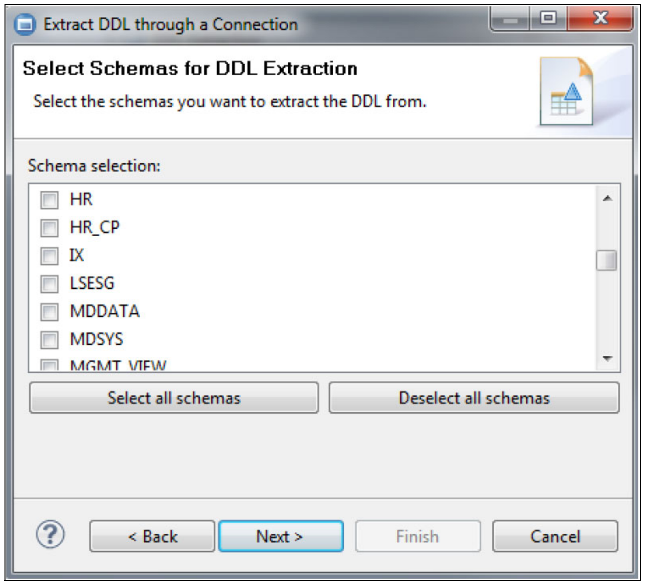
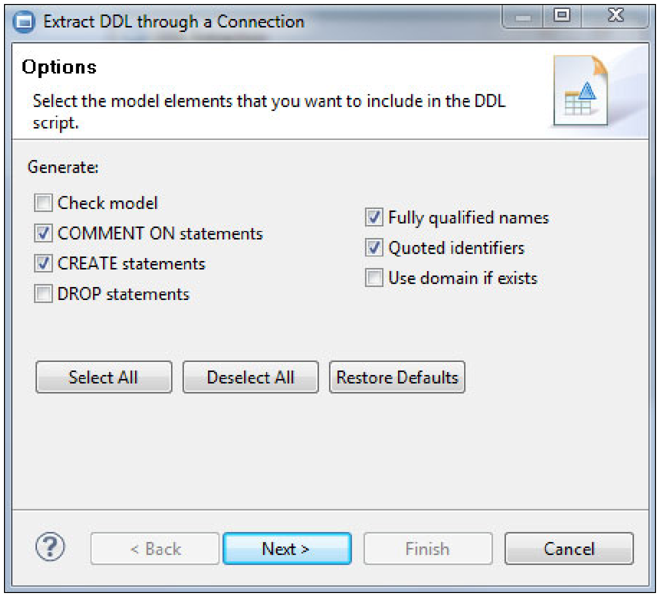

Extract DDL through a Connection
Select Schemas for DDL Extraction
Select the schemas you want to extract the DDL from.
Schema selection:
HR
HR_CP
IX
LSESG
MDDATA
MDSYS
MGMT_VIEW
Select all schemas
Deselect all schemas
< Back
Next >
Finish
Cancel

Figure 3-10 Selecting a schema for extraction

4. On the next window of the wizard, you can specify the model elements that you want to include in your DDL script (see Figure 3-11). After you select the elements, click **Next**.



Extract DDL through a Connection

Options

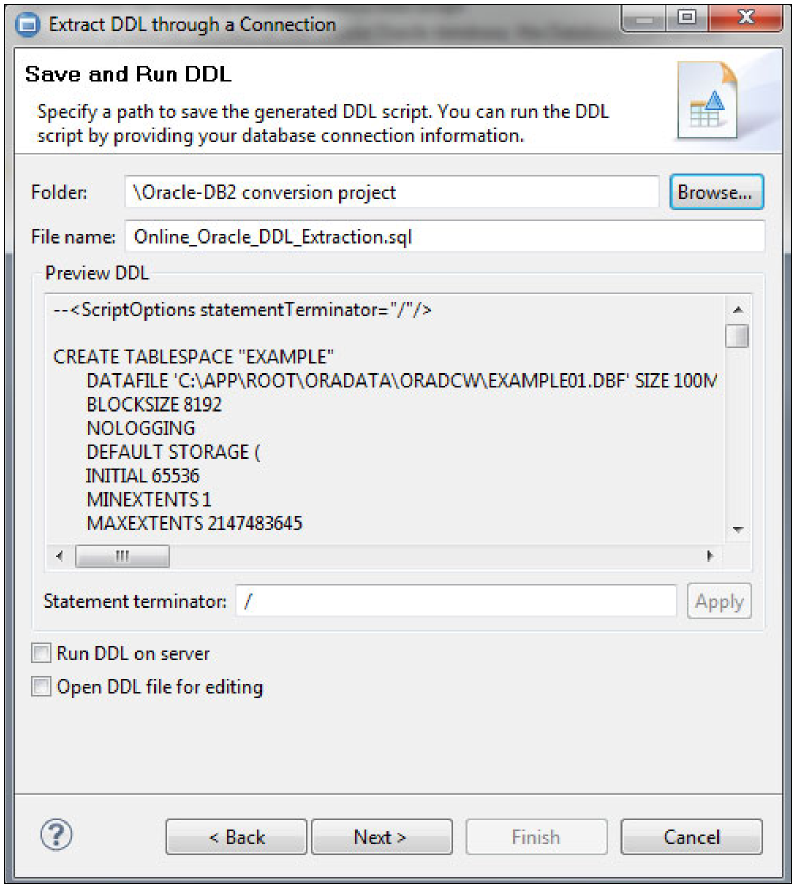
Select the model elements that you want to include in the DDL script.

Generate:

- [ ] Check model
- [✓] COMMENT ON statements
- [✓] CREATE statements
- [ ] DROP statements
- [✓] Fully qualified names
- [✓] Quoted identifiers
- [ ] Use domain if exists

Figure 3-11 Selecting model elements for the DDL script

5. You are now prompted to select a project directory, specify a file name for the extracted DDL, and select a statement terminator, as shown in Figure 3-12. The recommended statement terminator is the forward slash "/" character. This ensures that statements with embedded semicolons are extracted.



Specify a path to save the generated DDL script. You can run the DDL script by providing your database connection information.

Folder: \Oracle-DB2 conversion project

File name: Online_Oracle_DDL_Extraction.sql

Preview DDL

```
--<ScriptOptions statementTerminator="/"/>

CREATE TABLESPACE "EXAMPLE"
    DATAFILE 'C:\APP\ROOT\ORADATA\ORADCW\EXAMPLE01.DBF' SIZE 100M
    BLOCKSIZE 8192
    NOLOGGING
    DEFAULT STORAGE (
    INITIAL 65536
    MINEXTENTS 1
    MAXEXTENTS 2147483645
```

Statement terminator: /

Figure 3-12 Selecting the project directory, and so on

6. On the final window of the wizard, verify your selections and click **Finish**.

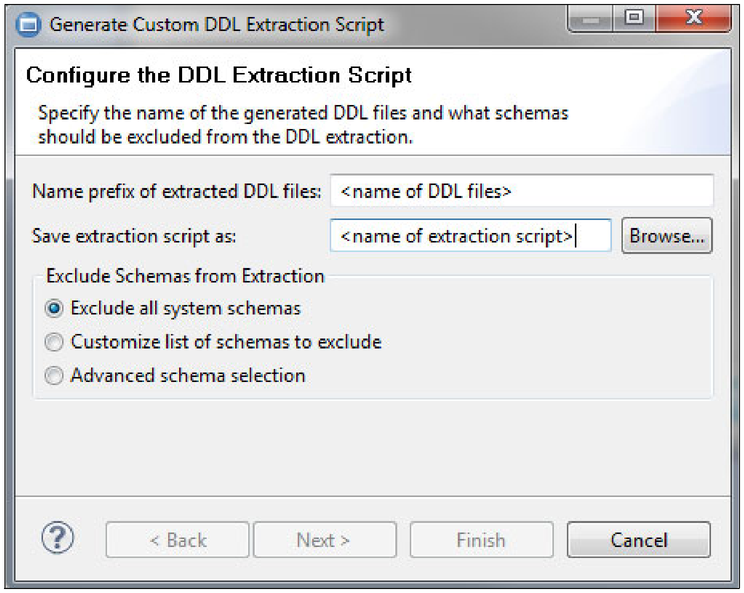
The extraction process generates a single SQL file containing the DDL of the selected schemas. The directory is placed in the project directory that you specified.

3.4.2 DDL extraction using a custom script

If a connection to the source Oracle database is not possible, DCW can generate a custom DDL extraction script that, when manually run against the source Oracle database, generates the required DDL.

To perform DDL extraction using a custom script, complete the following steps:

1. From the Oracle to DB2 Task Launcher, start the DDL Extraction wizard and select **Generate DDL Extraction Script**. This launches another wizard for generating the script (see Figure 3-13).


Generate Custom DDL Extraction Script

Configure the DDL Extraction Script

Specify the name of the generated DDL files and what schemas should be excluded from the DDL extraction.

Name prefix of extracted DDL files: <name of DDL files>

Save extraction script as: <name of extraction script>

Exclude Schemas from Extraction

[✓ Exclude all system schemas]

Customize list of schemas to exclude

Advanced schema selection

Figure 3-13 Generating a custom DDL extraction script

2. Follow the instructions in the wizard to specify the prefix of the DDL files to be generated and the name and location under which to save the custom DDL extraction script. When the wizard completes, the script is placed in the location that you specified.

   Note: As a preferred practice, select the **Exclude all system schemas** option to prevent the extraction of Oracle system schemas.

3. Run the extraction script through Oracle SQL *Plus on a machine that has access to the source Oracle database. This generates the DDL file, which is created in .out format and placed in the same location as the script.

The DDL file is now generated and available for use, but you must import the file into DCW before the rest of the migration process can proceed. This process might require you to move the DDL files manually from your Oracle system to the system on which DCW is installed.

3.4.3 Importing the DDL file

If you used the extraction script process to create the DLL file, or if a DDL file of the source Oracle database already exists, you must import the DDL file into your DCW project.

To import the file, go to the Project Explorer View, right-click the listing for your DCW project, and then navigate to **DCW Conversion** → **Import a DDL File**. This opens the Import wizard, where you select the extracted DDL files and specify the destination DCW project. When the process is initiated, the DDL files are automatically imported with a .sql extension.

This import process can be used for any external DDL file that was not extracted by DCW.

3.5 Assessment and conversion using DCW

Often, the most challenging part of database conversion is determining whether the source syntax is compatible with the target database and making any needed changes. The conventional approach is to manually compare thousands of lines of code from the source syntax against the target database. This can be a time-consuming and complex task, and is prone to many mistakes.

To help simplify things, the IBM Database Conversion Workbench offers two key features to help automate the process of evaluating extracted Oracle DDL code and converting it to the corresponding DB2 equivalent. The following subsections describe these features in detail.

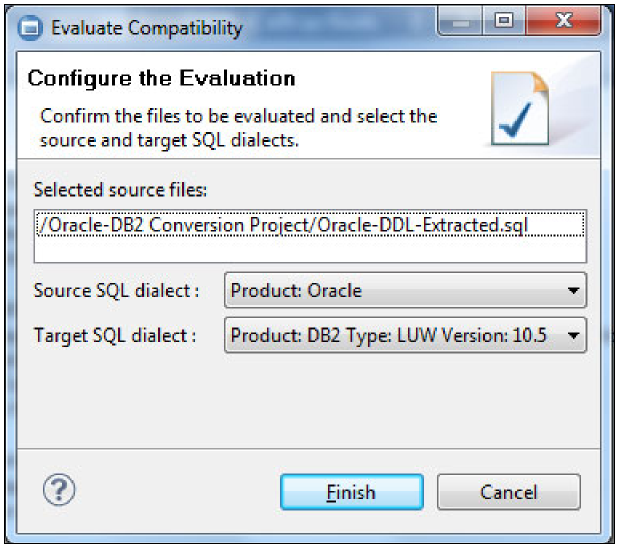
3.5.1 Evaluating an Oracle DDL

To help in assessing your source Oracle database for DB2 compatibility, DCW provides an Evaluate Compatibility feature that analyzes Oracle DDL and PL/SQL statements and generates a report that describes (as a percentage) the compatibility of the source Oracle database with DB2. The Compatibility Report shows the total number of statements that are detected and identifies how many of the statements have potential compatibility problems and thus require attention. These statistics become important for accurately planning the remainder of the conversion project.

In addition to assisting with conversion planning, the Compatibility Report also includes details about the specific lines of code that are likely to experience problems, and provides suggested workarounds for the DDL and PL/SQL incompatibilities.

Generating the Compatibility Report

Like all features in DCW, the user can start the Evaluate Compatibility wizard, which is shown in Figure 3-14, through the task launcher or by right-clicking the listing for the DDL file in the DCW project and then selecting **Database Conversion → Evaluate Compatibility**.



Configure the Evaluation

Confirm the files to be evaluated and select the source and target SQL dialects.

Selected source files:
/Oracle-DB2 Conversion Project/Oracle-DDL-Extracted.sql

Source SQL dialect : Product: Oracle

Target SQL dialect : Product: DB2 Type: LUW Version: 10.5

Figure 3-14 The Evaluate Compatibility wizard

After you open the wizard, select **Oracle** as the Source SQL dialect and your DB2 version as the Target SQL dialect. Click **Finish** to generate the report.

The output of the evaluation process is not just the report but also an `.xmle` file, which is the encrypted version of the report in XML format. This file is placed in your DCW project using the format `<filename>-<timestamp>_report.xmle`. The user must send the .xmle report to `askdcw@ca.ibm.com`, where IBM decrypts the report and returns it to the user in HTML format.

> **Note**: The `.xmle` file does not contain any source code. It contains only the list of identified incompatibilities and the number of those incompatibilities in the original source code.

Understanding the Compatibility Report

The DCW Compatibility Report is a comprehensive listing with multiple sections that describe the identified database compatibility issues with different levels of detail.

Several important terms are used in the report:

- Statement:

  Any part of the SQL script that addresses a specific purpose can be referred to as a statement. A statement can contain one or more statements within it and can vary from a simple variable declaration to a complex Create Table statement.

- DDL Statement:

  DDL Statements include all of the Create, Alter, and Drop statements of the Table Schema elements, such as Table, Index, Tablespace, Schema, Synonym, Type, Sequence, and View.

- PL/SQL Statement:

  Any statement that is not a DDL Statement is a PL/SQL Statement, from a small Loop Statement to a huge Query Block. A PL/SQL statement can be part of a DDL Statement, but a DLL Statement cannot exist within a PL/SQL statement.

- PL/SQL Objects:

  All of the declaration statements for Function, Procedure, Trigger, Package, Package Body, and Anonymous Blocks constitute PL/SQL objects. Any Function Declaration and Procedure Declaration statements that exist within other statements are also considered PL/SQL objects. By definition, every PL/SQL object is also a PL/SQL statement.

- LOC:

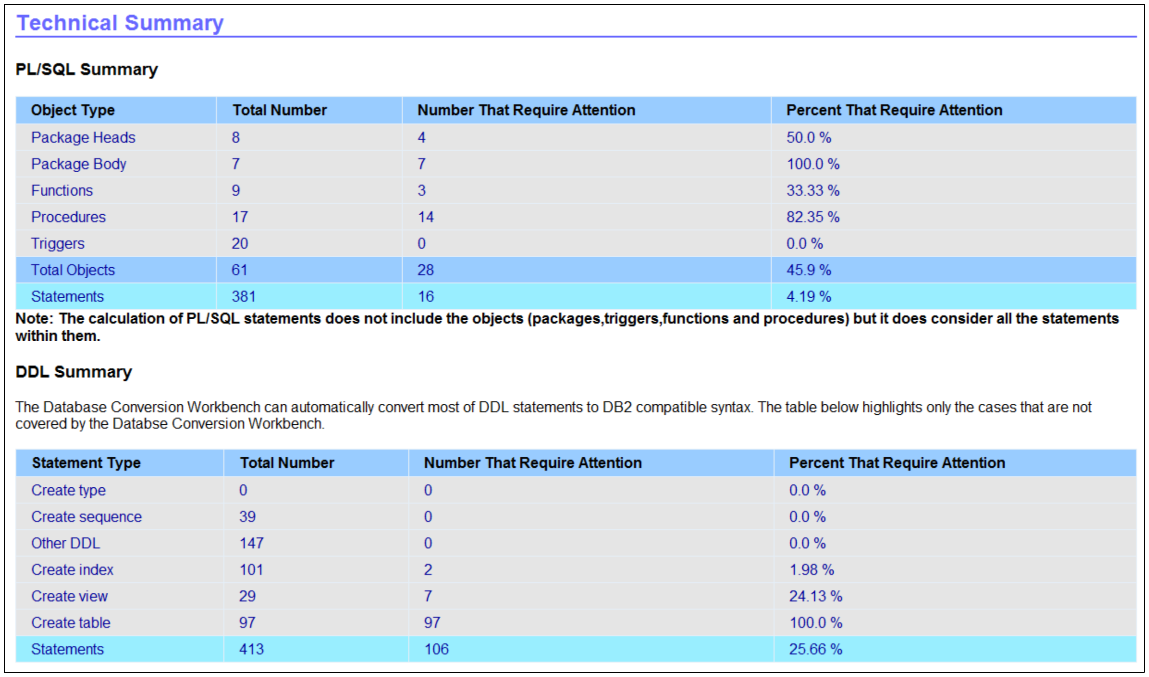
  LOC refers to the lines of code in a statement.

Note:

- Any SQL syntax that is not recognized by DCW is not considered for statistical calculation.
- Statements that can be successfully converted to their corresponding DB2 equivalent (using the DCW auto-conversion feature) are flagged as *compatible* and are not considered in the percentage calculations of incompatibility.

The Compatibility Report is divided into four sections.

- The Executive Summary is a high-level summary that cites the compatibility percentage of all PL/SQL and DDL Statements that can run natively on a DB2 database that were enabled for Oracle compatibility.
- The Technical Summary (see Figure 3-15) breaks up the statistics across different types of PL/SQL Objects and DDL Statements. For each type, the report provides the total number of statements that are analyzed and the percentage of incompatibility that is detected.



Technical Summary

PL/SQL Summary

Object Type	Total Number	Number That Require Attention	Percent That Require Attention
Package Heads	8	4	50.0 %
Package Body	7	7	100.0 %
Functions	9	3	33.33 %
Procedures	17	14	82.35 %
Triggers	20	0	0.0 %
Total Objects	61	28	45.9 %
Statements	381	16	4.19 %

Note: The calculation of PL/SQL statements does not include the objects (packages,triggers,functions and procedures) but it does consider all the statements within them.

DDL Summary

The Database Conversion Workbench can automatically convert most of DDL statements to DB2 compatible syntax. The table below highlights only the cases that are not covered by the Databse Conversion Workbench.

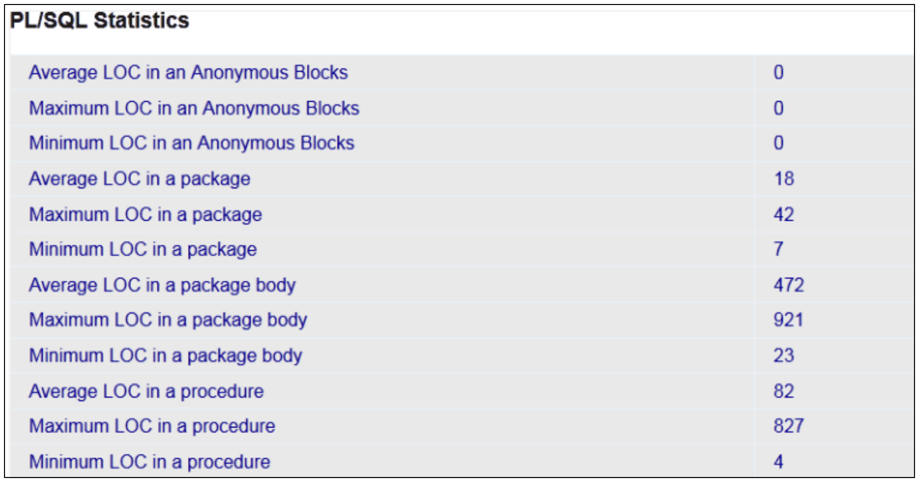
Statement Type	Total Number	Number That Require Attention	Percent That Require Attention
Create type	0	0	0.0 %
Create sequence	39	0	0.0 %
Other DDL	147	0	0.0 %
Create index	101	2	1.98 %
Create view	29	7	24.13 %
Create table	97	97	100.0 %
Statements	413	106	25.66 %

Figure 3-15 Technical Summary in a DCW Compatibility Report

Figure 3-15 on page 181 shows that the Technical Summary is divided into two sections, one for PL/SQL and the other for DDL. The PL/SQL Summary contains statistics for PL/SQL Objects and PL/SQL Statements.

- The Detailed Technical Summary gives an intricately detailed breakdown of the statements that were analyzed. Like the Technical Summary, this summary also has separate sections for PL/SQL and DDL.

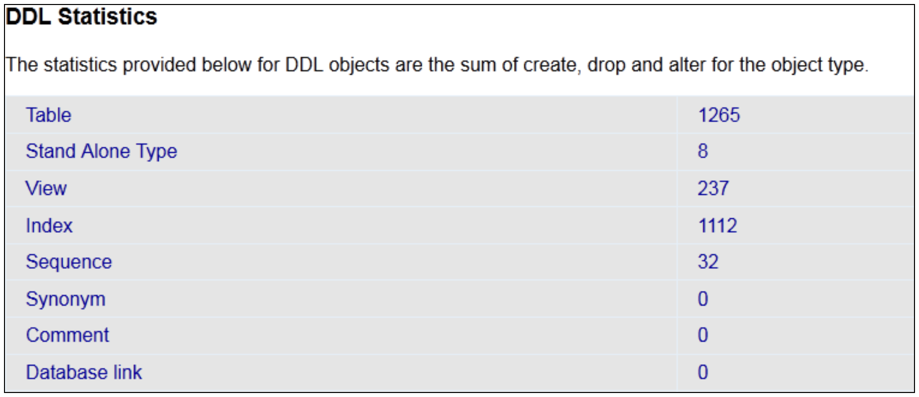
  The PL/SQL Statistics table (Figure 3-16) and DDL Statistics table (Figure 3-17) provide details about the total number of incompatible statements of each type. The PL/SQL table contains counts of the lines of code for each Object Type, whereas the DDL table shows the total count for each type of DDL (the sum of the Create, Alter, and Drop statements).



PL/SQL Statistics

Average LOC in an Anonymous Blocks	0
Maximum LOC in an Anonymous Blocks	0
Minimum LOC in an Anonymous Blocks	0
Average LOC in a package	18
Maximum LOC in a package	42
Minimum LOC in a package	7
Average LOC in a package body	472
Maximum LOC in a package body	921
Minimum LOC in a package body	23
Average LOC in a procedure	82
Maximum LOC in a procedure	827
Minimum LOC in a procedure	4

Figure 3-16 PL/SQL Statistics table in Detailed Technical Summary



DDL Statistics

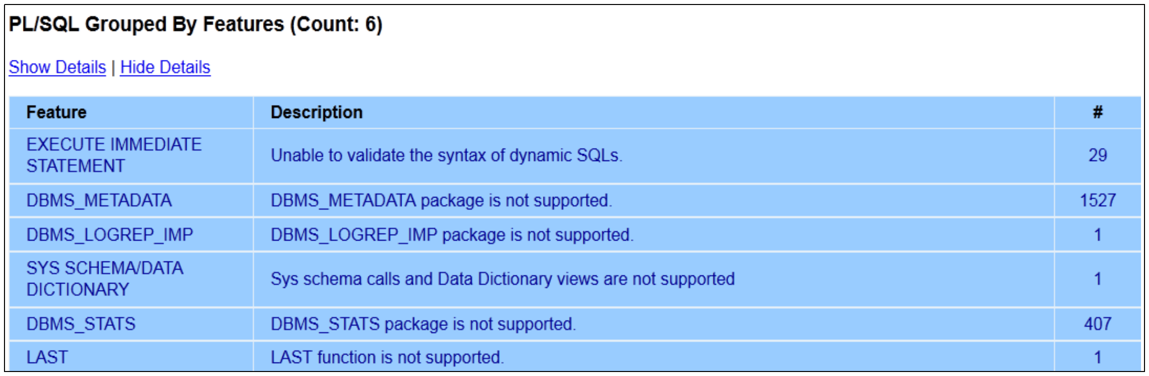
The statistics provided below for DDL objects are the sum of create, drop and alter for the object type.

Table	1265
Stand Alone Type	8
View	237
Index	1112
Sequence	32
Synonym	0
Comment	0
Database link	0

Figure 3-17 DLL Statistics table in Detailed Technical Summary

Although the statistics in the tables do not contain information about incompatibilities, they give a good estimation of the size of the source database.

The Detailed Technical Summary also identifies any issues that require attention, dividing them into two groups: one that lists the features that contributed toward the reported percentage of incompatibility, and another that is informational. The informational section lists those compatibility issues that can be automatically repaired by DCW but are likely to still need confirmation from the user. Both PL/SQL- and DDL-related issues are shown in the same structure, as illustrated in Figure 3-18 and Figure 3-19.

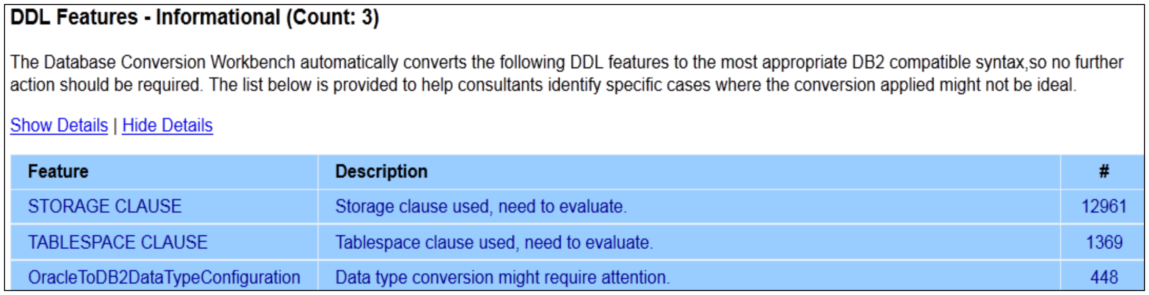


PL/SQL Grouped By Features (Count: 6)

Show Details | Hide Details

Feature	Description	#
EXECUTE IMMEDIATE STATEMENT	Unable to validate the syntax of dynamic SQLs.	29
DBMS_METADATA	DBMS_METADATA package is not supported.	1527
DBMS_LOGREP_IMP	DBMS_LOGREP_IMP package is not supported.	1
SYS SCHEMA/DATA DICTIONARY	Sys schema calls and Data Dictionary views are not supported	1
DBMS_STATS	DBMS_STATS package is not supported.	407
LAST	LAST function is not supported.	1

Figure 3-18   Detailed Technical Summary that is grouped by Features



DDL Features - Informational (Count: 3)

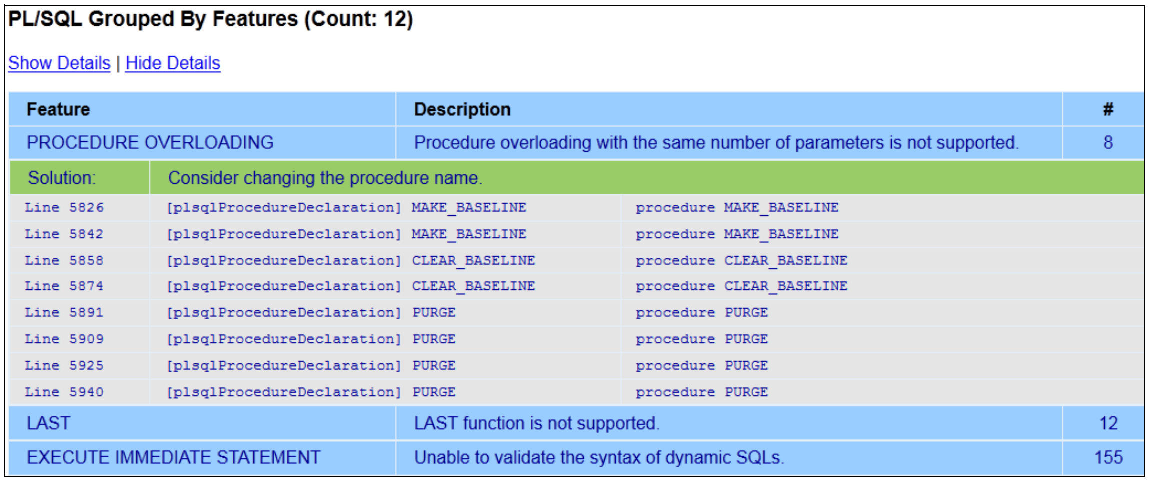
The Database Conversion Workbench automatically converts the following DDL features to the most appropriate DB2 compatible syntax,so no further action should be required. The list below is provided to help consultants identify specific cases where the conversion applied might not be ideal.

Show Details | Hide Details

Feature	Description	#
STORAGE CLAUSE	Storage clause used, need to evaluate.	12961
TABLESPACE CLAUSE	Tablespace clause used, need to evaluate.	1369
OracleToDB2DataTypeConfiguration	Data type conversion might require attention.	448

Figure 3-19   Detailed Technical Summary - DDL Features

You can expand or collapse each incompatible feature by clicking the statement in the list, as shown in Figure 3-20. This expands the entry and provides a list of all occurrences of the feature, including the line number of problem statements in the original DDL code. The expanded entry also shows the part of the syntax that contains the error and lists a recommended solution.



PL/SQL Grouped By Features (Count: 12)

Show Details | Hide Details

Feature	Description	#
PROCEDURE OVERLOADING	Procedure overloading with the same number of parameters is not supported.	8

Solution:	Consider changing the procedure name.	
Line 5826	[plsqlProcedureDeclaration] MAKE_BASELINE	procedure MAKE_BASELINE
Line 5842	[plsqlProcedureDeclaration] MAKE_BASELINE	procedure MAKE_BASELINE
Line 5858	[plsqlProcedureDeclaration] CLEAR_BASELINE	procedure CLEAR_BASELINE
Line 5874	[plsqlProcedureDeclaration] CLEAR_BASELINE	procedure CLEAR_BASELINE
Line 5891	[plsqlProcedureDeclaration] PURGE	procedure PURGE
Line 5909	[plsqlProcedureDeclaration] PURGE	procedure PURGE
Line 5925	[plsqlProcedureDeclaration] PURGE	procedure PURGE
Line 5940	[plsqlProcedureDeclaration] PURGE	procedure PURGE

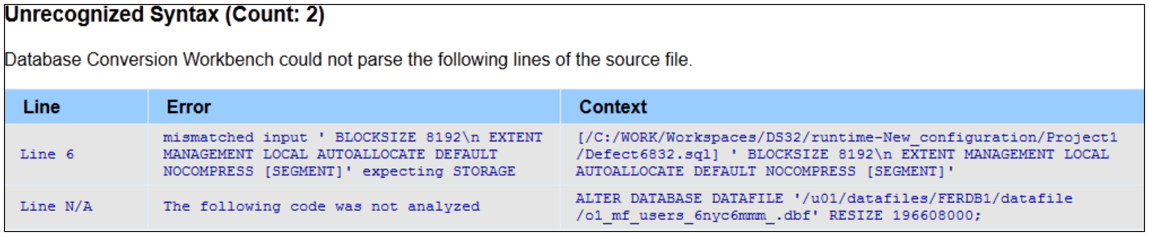
Feature	Description	#
LAST	LAST function is not supported.	12
EXECUTE IMMEDIATE STATEMENT	Unable to validate the syntax of dynamic SQLs.	155

Figure 3-20 Detailed Technical Summary Grouped By Features with Solutions

The three sections of the Compatibility Report are aimed at different audiences and used for different purposes. The Executive Summary is intended for management and for gauging the complexity of a migration project at a high level. The Technical Summary helps users to create an initial sketch of their conversion plan. The Detailed Technical Summary is intended for the users who perform the migration tasks.

Unrecognized lines of code

There are instances when the Database Conversion Workbench is unable to decipher parts of the source DDL. These lines of code do not contribute to the compatibility statistics, so it is up to the user to determine whether the code is DB2 compatible syntax. Figure 3-21 on page 185 shows a simple example of a portion of Unrecognized Syntax.


Unrecognized Syntax (Count: 2)

Database Conversion Workbench could not parse the following lines of the source file.

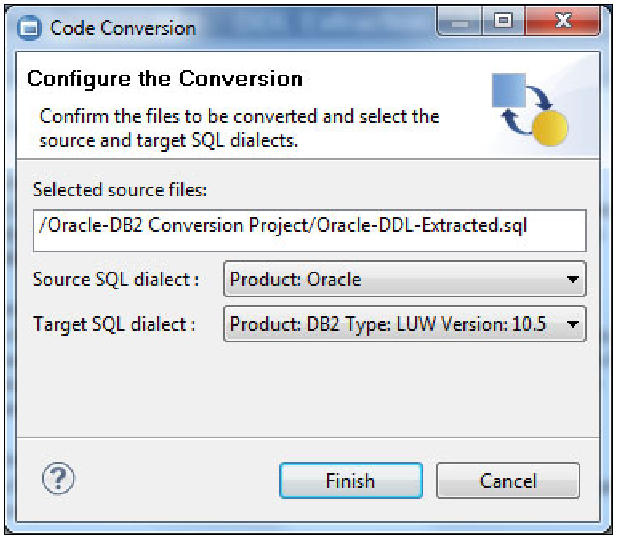
Line	Error	Context
`Line 6`	`mismatched input ' BLOCKSIZE 8192\n EXTENT MANAGEMENT LOCAL AUTOALLOCATE DEFAULT NOCOMPRESS [SEGMENT]' expecting STORAGE`	`[/C:/WORK/Workspaces/DS32/runtime-New_configuration/Project1 /Defect6832.sql] ' BLOCKSIZE 8192\n EXTENT MANAGEMENT LOCAL AUTOALLOCATE DEFAULT NOCOMPRESS [SEGMENT]'`
`Line N/A`	`The following code was not analyzed`	`ALTER DATABASE DATAFILE '/u01/datafiles/FERDB1/datafile /o1_mf_users_6nyc6mmm_.dbf' RESIZE 196608000;`

Figure 3-21 DCW Evaluation - Unrecognized Syntax

3.5.2 Converting Oracle DDL to DB2 compatible syntax

As stated earlier in this chapter, the most time-consuming part of a database migration is the manual conversion of source syntax to compatible target syntax. Fortunately, in addition to the DB2 to Oracle compatibility features that have already been explained, the Database Conversion Workbench provides a powerful tool that can auto-convert well-known Oracle syntax to DB2 -compatible syntax even if the other features cannot. Although it is nearly impossible to perform a complete conversion automatically, the efficiency of DCW in generating the required DB2 -equivalent code is considered remarkable.

The user can start the Code Conversion wizard, which is shown in Figure 3-22, through the Task Launcher or by right-clicking the listing for the source DDL file in their DCW project and then selecting **Database Conversion** → **Convert Code**.



Configure the Conversion

Confirm the files to be converted and select the source and target SQL dialects.

Selected source files: /Oracle-DB2 Conversion Project/Oracle-DDL-Extracted.sql

Source SQL dialect : Product: Oracle

Target SQL dialect : Product: DB2 Type: LUW Version: 10.5

Figure 3-22 Code Conversion wizard

In the wizard, select **Oracle** as the Source SQL dialect and your DB2 version as the Target SQL dialect. Click **Finish** to initiate the conversion and a new converted SQL file is generated in your DCW project folder, where it exists alongside the original SQL file.

The rules and logic of code conversions in DCW are based on field knowledge and the leading practices that are developed by IBM consultants. It is not possible to list all of these rules and configurations in this book. However, several of the most important and complex conversion rules are worth explaining:

- Nested conversions:

  Nested functions are not supported in DB2 but are a common practice in Oracle. DCW unnests these functions, gives them a new name, and then changes all of the calls to the functions to use the new name.

- Data type conversions:

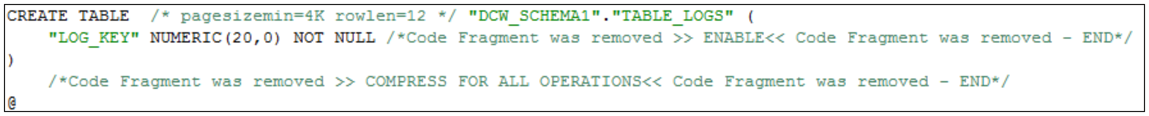
  Many Oracle data types are not compatible with DB2 syntax, either because the data type is represented differently or the offset values for the arguments of the data type are different in DB2. DCW overcomes this issue with a set of Mapper rules for converting from Oracle to DB2 data types. The Mapper rules are listed in Appendix B, “Data types” on page 309.

- Statement separator conversions:

  DCW converts each Oracle statement separator (typically the ; or / characters) to the @ character, which is the default separator in DB2.

Using these rules, the Code Conversion wizard performs the following actions:

- Remove code fragment: DCW comments out the Oracle clauses that are not supported and not required on DB2. Code that is removed is tagged with the phrase `Code Fragment was removed` in the converted file. Figure 3-23 shows an example in which both the Enable Disable Clause and the Table Compression Clause are removed.



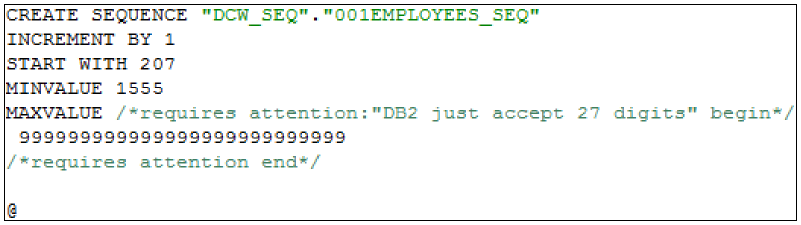
```
CREATE TABLE  /* pagesizemin=4K rowlen=12 */ "DCW_SCHEMA1"."TABLE_LOGS" (
    "LOG_KEY" NUMERIC(20,0) NOT NULL /*Code Fragment was removed >> ENABLE<< Code Fragment was removed - END*/
)
    /*Code Fragment was removed >> COMPRESS FOR ALL OPERATIONS<< Code Fragment was removed - END*/
@
```

Figure 3-23 Indication that a code fragment was removed

- Code conversion: For many Oracle statements, DCW can fully convert the syntax to be compatible with DB2. Statements that are converted are not flagged by DB2.

- Requires Attention: When DCW performs a partial conversion or when it suspects data inconsistency because of the changes it is making, it tags the code with the phrase `Requires Attention`. An example of this tagging is the case of a data type conversion where DCW limits the precision of the NUMBER data type to 31 (which this is the limit for DB2; Oracle may accept higher values). Although this change allows the statement to run on DB2, it can cause problems for any applications requiring greater precision.

  Figure 3-24 illustrates another example where the `MAXVALUE` of a Sequence Statement is restricted to 27 digits for DB2, whereas Oracle can accept 28. In such cases, because of the risk of losing data, DCW tags the statement with `Requires Attention`.

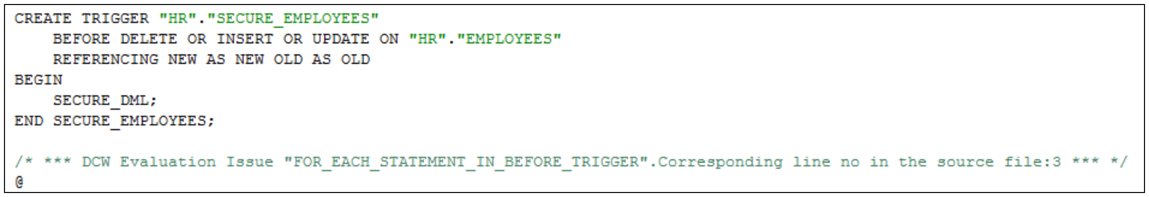


```
CREATE SEQUENCE "DCW_SEQ"."001EMPLOYEES_SEQ"
INCREMENT BY 1
START WITH 207
MINVALUE 1555
MAXVALUE /*requires attention:"DB2 just accept 27 digits" begin*/
 999999999999999999999999999
/*requires attention end*/

@
```

Figure 3-24 Indication that a statement requires attention

- Evaluation Issues: DCW highlights any statements that are identified as potential issues in a Compatibility Report. These statements cannot be converted automatically and so are tagged in the converted code as having a `DCW Evaluation Issue`.

  Figure 3-25 shows an example of the `For Each` statement in `Before Trigger`, which is not supported in DB2 and for which there is no easy conversion.



```
CREATE TRIGGER "HR"."SECURE_EMPLOYEES"
    BEFORE DELETE OR INSERT OR UPDATE ON "HR"."EMPLOYEES"
    REFERENCING NEW AS NEW OLD AS OLD
BEGIN
    SECURE_DML;
END SECURE_EMPLOYEES;

/* *** DCW Evaluation Issue "FOR_EACH_STATEMENT_IN_BEFORE_TRIGGER".Corresponding line no in the source file:3 *** */
@
```

Figure 3-25 Indication that a statement cannot be auto-converted

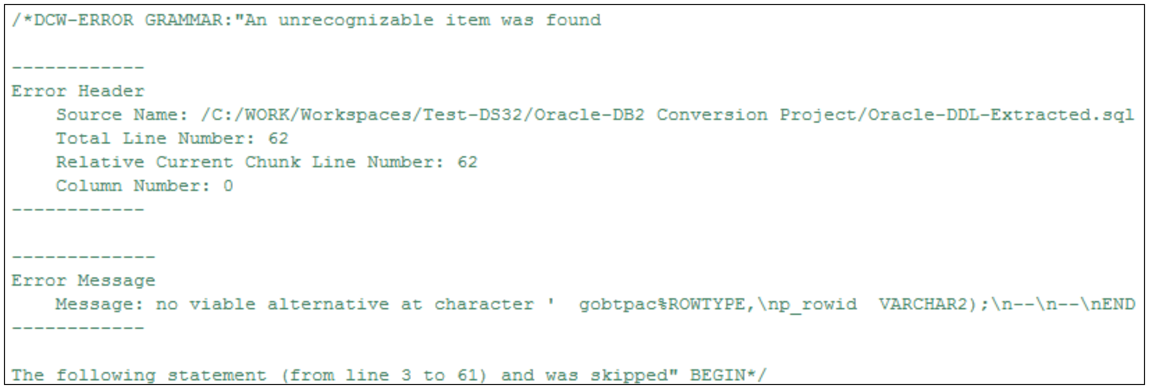
Reviewing the converted output

Although the DCW auto-conversion tools help streamline the code conversion process, user input and review are *still* required.

Typically, a user must perform four actions to complete a database conversion after the DCW auto-conversion activities are finished:

1. Target the portions of code that were not recognized or processed by DCW, which are tagged with the phrase `DCW ERROR GRAMMAR`, as shown in Figure 3-26.

   These problem areas might have resulted from improperly formatted syntax or parsing errors, or the statement might not have been recognized. You want to investigate these parts of the code, fix any problems in your original DDL, and then rerun the DCW conversion process.



```
/*DCW-ERROR GRAMMAR:"An unrecognizable item was found

------------
Error Header
    Source Name: /C:/WORK/Workspaces/Test-DS32/Oracle-DB2 Conversion Project/Oracle-DDL-Extracted.sql
    Total Line Number: 62
    Relative Current Chunk Line Number: 62
    Column Number: 0
------------

-------------
Error Message
    Message: no viable alternative at character '  gobtpac%ROWTYPE,\np_rowid  VARCHAR2);\n--\n--\nEND
------------

The following statement (from line 3 to 61) and was skipped" BEGIN*/
```

Figure 3-26 Indication of unrecognized syntax

2. Review code that was processed but for which DCW was unable to find a viable alternative. Search for “`DCW Evaluation Issue`” to locate these statements in your converted code.

   For help identifying a solution to the problem code, see the Compatibility Report. The report is often where the issue was first listed and usually includes recommended solutions.

3. After the unrecognized syntax and evaluation issues are resolved, search your converted code for `Code Fragment Removed`, which marks statements that were commented out by DCW. Review these sections of code to ensure that their removal does not have negative effects on your application.

4. Finally, check the code for segments that are tagged with the phrase `Requires Attention`. Such code is likely to require verification to ensure that the conversion that was applied is valid for your application. You might need to make adjustments to ensure that it is fully compatible with DB2.

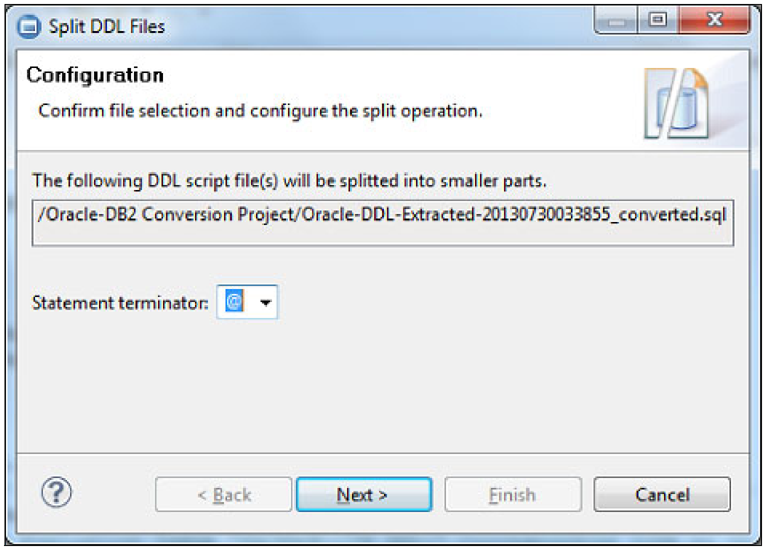
3.5.3 The Split DDL function

Because code conversion is a difficult process to track and organize, DCW offers the Split DLL function, an optional but useful feature that splits a single DDL file into multiple files that are organized by object type. The function helps streamline your conversion by making the code organization more intuitive. Also, by breaking down a large DDL file into smaller and more manageable components, multiple team members can work on different parts of the project concurrently.

Although the Split DLL function supports any type of SQL, it is a preferred practice that it is run only after the DCW code conversion is performed. This prevents the need to run code conversion on multiple separate files.

Complete the following steps:

1. Open the Split DLL Files wizard (see Figure 3-27) by using the task launcher or by right-clicking the listing for the DDL file in your DCW project and then selecting **Database Conversion** → **Split DDL Files**.



Configuration

Confirm file selection and configure the split operation.

The following DDL script file(s) will be splitted into smaller parts.

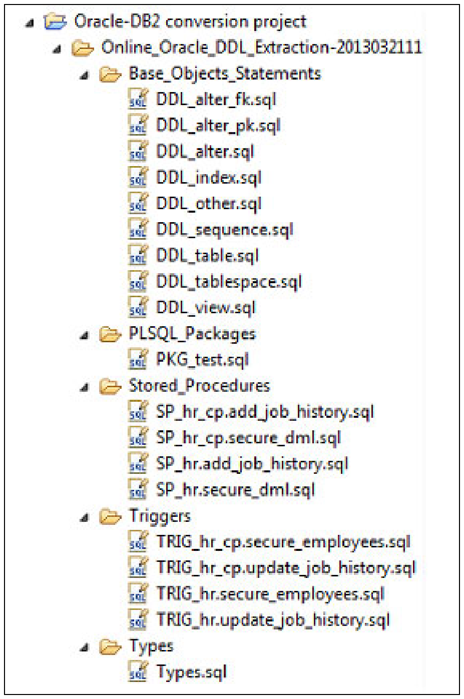
/Oracle-DB2 Conversion Project/Oracle-DDL-Extracted-20130730033855_converted.sql

Statement terminator: @

Figure 3-27 Split DDL Files wizard

2. In the wizard, designate a statement terminator. Choose the @ character for the converted DB2 code and the / character for the source Oracle code. This terminator determines how the file is split. Choosing the wrong terminator results in the file being split incorrectly. Click **Next**.

3. The process creates a directory in your DCW project and puts the split files in specific directories that are named according to the Object types, as shown in Figure 3-28.



```
Oracle-DB2 conversion project
  Online_Oracle_DDL_Extraction-2013032111
    Base_Objects_Statements
      DDL_alter_fk.sql
      DDL_alter_pk.sql
      DDL_alter.sql
      DDL_index.sql
      DDL_other.sql
      DDL_sequence.sql
      DDL_table.sql
      DDL_tablespace.sql
      DDL_view.sql
    PLSQL_Packages
      PKG_test.sql
    Stored_Procedures
      SP_hr_cp.add_job_history.sql
      SP_hr_cp.secure_dml.sql
      SP_hr.add_job_history.sql
      SP_hr.secure_dml.sql
    Triggers
      TRIG_hr_cp.secure_employees.sql
      TRIG_hr_cp.update_job_history.sql
      TRIG_hr.secure_employees.sql
      TRIG_hr.update_job_history.sql
    Types
      Types.sql
```

Figure 3-28 Directory structure organizes split files by Object type

Although splitting the DDL is an optional step, it is a useful utility that can be applied to many processes. The DCW documentation recommends using the Split DLL function as a way to achieve better management and understanding of the SQL code you are converting.

3.6 Preparing your DB2 database for data movement

The final phase of the database conversion process is migrating the data from your source Oracle database to the target DB2 database. However, before you can begin migrating data, you must prepare the target DB2 database to accept data. This typically involves creating the target DB2 database, and then deploying the required DDL for data movement.

3.6.1 Creating the target DB2 database

Creating and setting up the target DB2 database requires manual steps that are not supported by DCW. Details about these steps are out of the scope of this book, but it is still useful to describe certain preferred practices regarding the process:

- Run **`CREATE DATABASE`** on the target DB2 server to create your target database.
- Alternatively, you can use Data Studio to create the database. For tips about this topic, consult the contents of the Data Studio Help menu.
- You can adjust the database configuration parameters to meet your requirements by running **`UPDATE DATABASE CONFIGURATION`**.

For more information about creating your target database, including the DB2 commands that are mentioned here, see the IBM DB2 Information Center at this address:

http://pic.dhe.ibm.com/infocenter/db2luw/v10r5/index.jsp

Oracle Compatibility feature

The SQL compatibility features that are built into DB2 eliminate the need to convert many Oracle database objects and Oracle SQL to DB2 syntax.

To enable the Oracle Compatibility feature in DB2, complete the following steps:

1. Using the DB2 command line, run **`db2set DB2_COMPATIBILITY_VECTOR=ORA`**. This command enables the Oracle Compatibility feature.
2. Stop and restart DB2 by running **`db2stop`** and **`db2start`**.

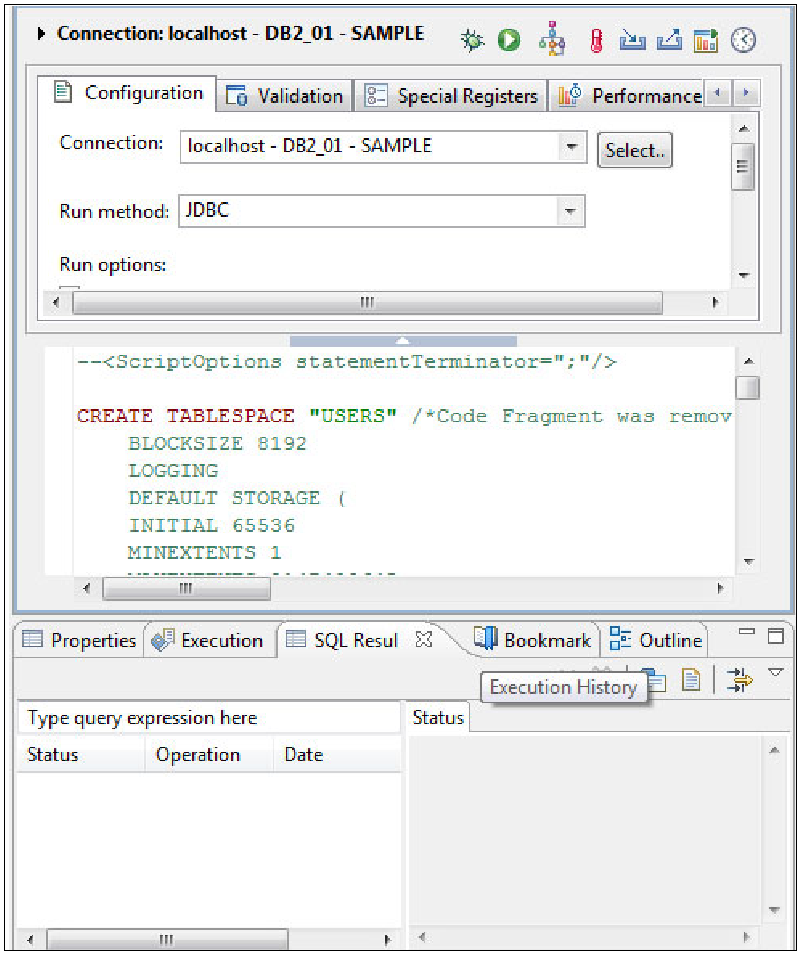
3.6.2 Deploying the DDL objects that are required for data movement

After the source DDL is converted and all issues are resolved, you can create the required objects on the target DB2 database by running SQL statements directly from DCW. This action is a precondition for data movement because the target DB2 database objects must exist before data can be mapped and migrated from the source Oracle database.

In most cases, you must start this step by creating the required buffer pools, table spaces, and tables. As a preferred practice, all other database objects should be created *after* data movement is complete.

Creating objects using SQL Editor

Double-clicking any SQL file in your DCW project opens the file in the SQL Editor, as shown in Figure 3-29.



Connection: localhost - DB2_01 - SAMPLE

Run method: JDBC

```
--<ScriptOptions statementTerminator=";"/>

CREATE TABLESPACE "USERS" /*Code Fragment was remov
    BLOCKSIZE 8192
    LOGGING
    DEFAULT STORAGE (
    INITIAL 65536
    MINEXTENTS 1
```

Figure 3-29 Deploying DDL files using the SQL Editor

To create objects using SQL Editor, select the target DB2 database from the **Connection** drop-down list and then click **Play** (the green button in the upper right line of SQL Editor). Upon initiating the process, the DDL statements are run against the target database that you specified.

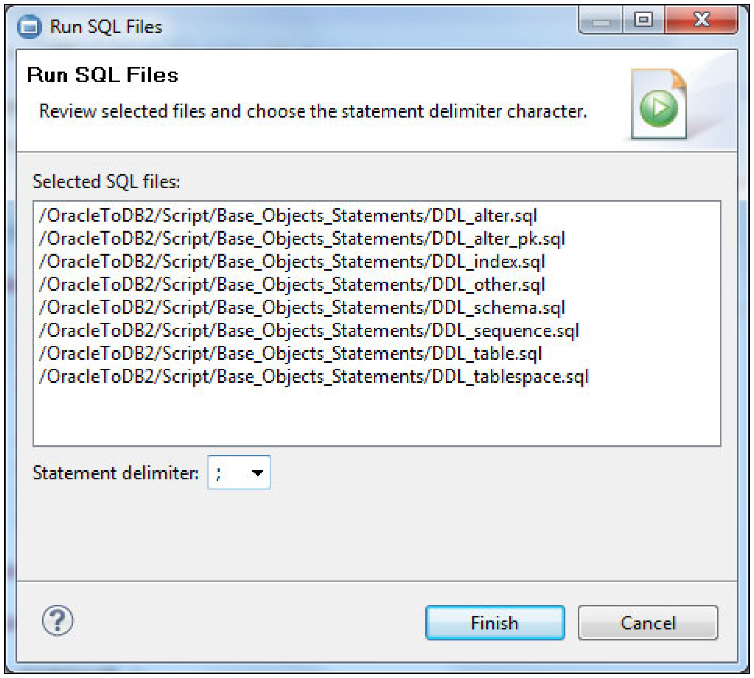
You can monitor the status and results of the statement execution on the editor's SQL Results tab.

Creating objects by using the Run SQL Files batch function

In addition to the option of using SQL Editor, DCW offers a batch execute function (see Figure 3-30) to run multiple SQL files at once.

To use the batch execute function, follow these steps:

1. In your DCW Project, select the SQL files that you want to run. If you are running multiple files, select them all by holding down the Shift key while making your selections. You can also select a folder to run all of the SQL files that are contained within it.
2. Right-click the DDL files and then select **Database Conversion** → **Run SQL Files...**.
3. The Run SQL files wizard opens. Set the **Statement delimiter** to the **@** character. Click **Finish**.
4. You are prompted to select the **Connection profile** of your target DB2 database.
5. Click **Finish** to initiate execution of the scripts.



Run SQL Files

Review selected files and choose the statement delimiter character.

Selected SQL files:

/OracleToDB2/Script/Base_Objects_Statements/DDL_alter.sql
/OracleToDB2/Script/Base_Objects_Statements/DDL_alter_pk.sql
/OracleToDB2/Script/Base_Objects_Statements/DDL_index.sql
/OracleToDB2/Script/Base_Objects_Statements/DDL_other.sql
/OracleToDB2/Script/Base_Objects_Statements/DDL_schema.sql
/OracleToDB2/Script/Base_Objects_Statements/DDL_sequence.sql
/OracleToDB2/Script/Base_Objects_Statements/DDL_table.sql
/OracleToDB2/Script/Base_Objects_Statements/DDL_tablespace.sql

Statement delimiter: ;

Figure 3-30 Deploying DDL files by using the DCW Run SQL Files feature

3.7 Data movement using DCW

After all of the required objects are replicated on your target DB2 database, you can move the data from your source database to your target database. DCW contains various data movement methods to suit your conversion needs. These options range from manual load scripts that can be run against the target DB2 database to advanced features that can be used in environments where databases are progressive and the downtime window is low.

DCW offers several data movement options:

- Extract and load data from flat files.
- Data movement using pipes.
- Data movement using federation.
- Data movement using IBM InfoSphere Change Data Capture.
- Manual deployment of data using scripts.

Note: Depending on what licenses are purchased, some of the advanced data movement features that are described here might require additional licenses from IBM.

Each of these methods can be accessed through the DCW context menu (accessible by right-clicking the listing for your DCW project or from the Task Launcher). The next subsections describe each option. A quick reference guide that summarizes all data movement methods is provided in 3.7.6, "Selecting the appropriate data movement method" on page 210.

3.7.1 Data movement using flat files

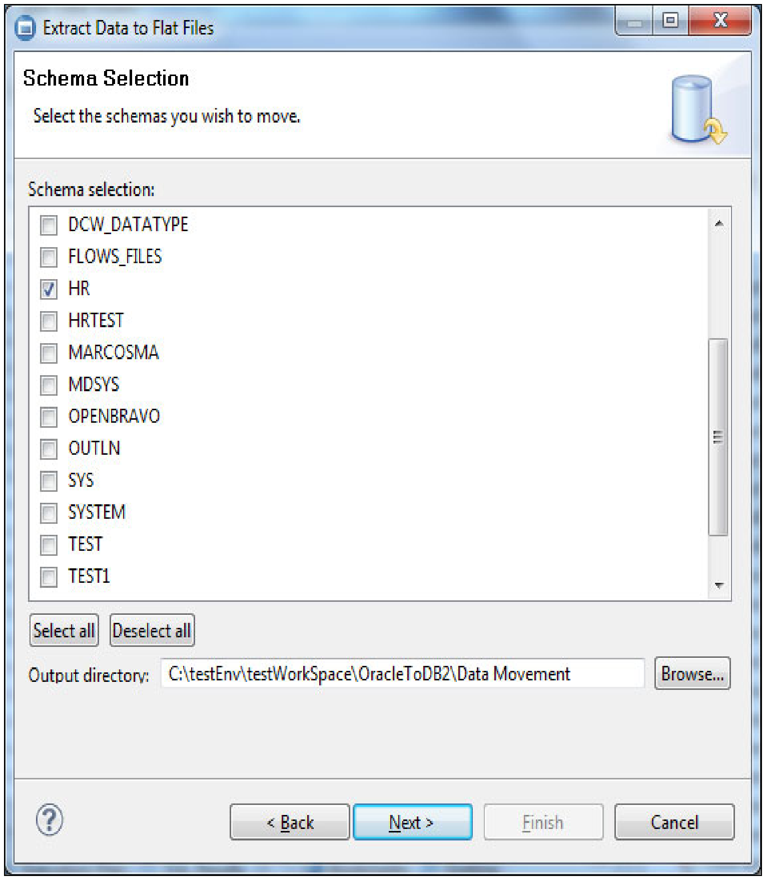
A classic way to move data is by extracting it from the Oracle database to delimited flat files and then loading it from the flat files to your target DB2 database. This is a two-part process and requires additional disk space to store the temporary flat files.

Extracting data to flat files

To extract data, users might require certain admin privileges for the source Oracle database. With these credentials, DCW can establish a JDBC connection with the source database and fetch data through simple SELECT * FROM table SQL queries. During the extraction, the load scripts for the flat files are generated.

Complete the following steps:

1. Right-click the listing for your DCW project and select **Database Conversion** → **Extract Data**. This opens the Extract Data to Flat Files wizard. This wizard is also accessible from the Task Launcher.
2. Select the source Oracle database and then click **Next**.
3. Select the schema that you want to move and enter the output directory for the flat files and logs (see Figure 3-31). Click **Next**.



Extract Data to Flat Files

Schema Selection

Select the schemas you wish to move.

Schema selection:

- [ ] DCW_DATATYPE
- [ ] FLOWS_FILES
- [✓] HR
- [ ] HRTEST
- [ ] MARCOSMA
- [ ] MDSYS
- [ ] OPENBRAVO
- [ ] OUTLN
- [ ] SYS
- [ ] SYSTEM
- [ ] TEST
- [ ] TEST1

Output directory: C:\testEnv\testWorkSpace\OracleToDB2\Data Movement

Figure 3-31 Selecting schemas for movement by using the Extract Data to Flat Files wizard

4. Select the tables that you want to move (see Figure 3-32) and click **Next**.

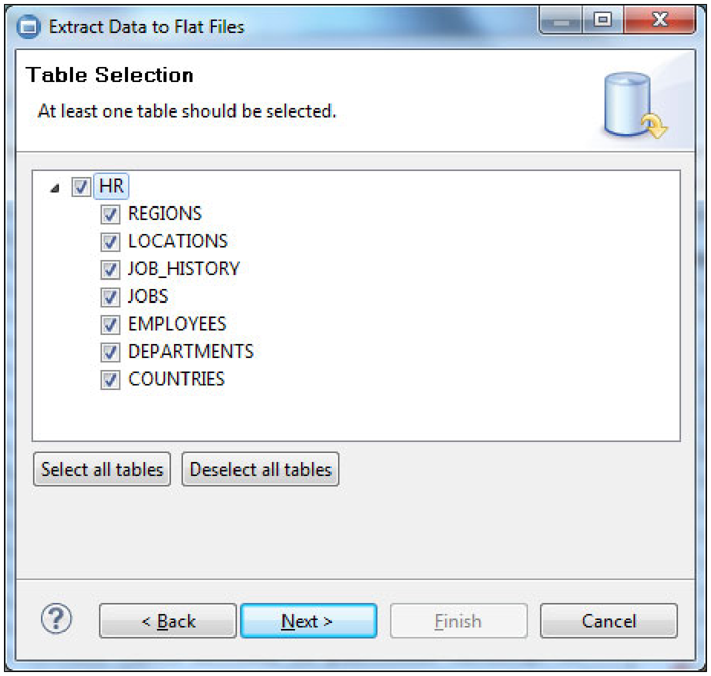


Table Selection

At least one table should be selected.

- [✓] HR
  - [✓] REGIONS
  - [✓] LOCATIONS
  - [✓] JOB_HISTORY
  - [✓] JOBS
  - [✓] EMPLOYEES
  - [✓] DEPARTMENTS
  - [✓] COUNTRIES

Figure 3-32 Selecting tables for movement in the Extract Data to Flat Files wizard

5. Specify the data movement configuration parameters. Novice users can use the default selections. Click **Next**.
6. View the summary to confirm your selections and click **Finish** to begin extracting the data.

The extraction log is automatically displayed in Data Studio. The data is extracted to the directory that was specified in step 3 on page 195 using the file extension `.tables`.

Note: To transfer the extracted data to a different computer, copy the specified directory to an external storage medium.

Loading data from flat files

The process of loading data from flat files is similar to that of data extraction. The user needs LOAD privileges and DCW must be installed on the target DB2 server. The load script that was generated during the extraction process contains the commands that must be used to load the tables that were specified. DCW can process the script and internally run `SYSPROC.ADMIN_CMD()` commands to load the data from the flat files.

Complete the following steps:

1. Right-click the listing for your DCW project and select **Database Conversion** → **Load Data**. This opens the Load Data wizard. This wizard is also accessible from the Task Launcher.
2. Select the target DB2 database. Click **Next**.
3. Select the directory location that was used while extracting the data to the flat files. Click **Next**.
4. View the summary and click **Finish**. This initiates the deployment process. The wizard displays a progress indicator until the process is complete.

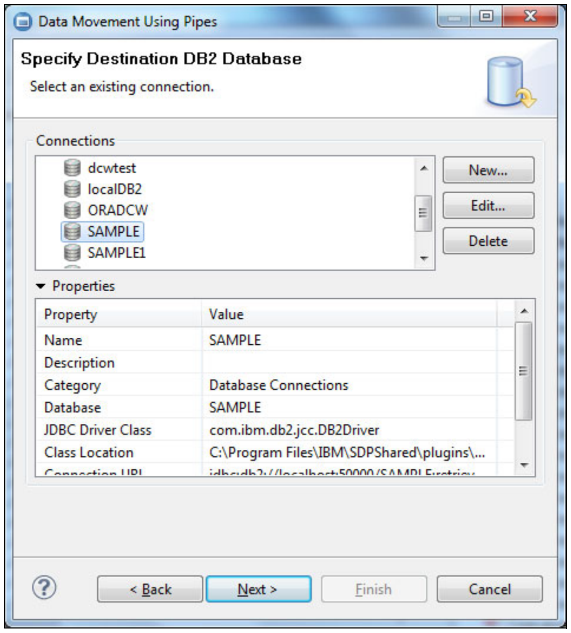
3.7.2 Data movement using pipes

In scenarios where using a secondary staging area for temporary flat files is not viable, DCW provides the option to use pipes instead. In this process, DCW extracts data and loads it into the target database in a single step, without needing to save anything in intermediate files. To use this feature, the user needs both OS and database administrator privileges, and DCW must be installed on the target DB2 server.

Complete the following steps:

1. Right-click the listing for the DCW project and select **Database Conversion** → **Move Data...**. This opens the Move Data wizard. This wizard is also accessible from the Task Launcher.
2. Select **Data Movement Using Pipes** to open the Data Movement Using Pipes wizard.
3. On the first window, you can select an existing Oracle database connection, or create a connection by clicking **New...**.

4. On the second window (Figure 3-33), you can select an existing target DB2 database connection, or create a connection by clicking **New...**.



Data Movement Using Pipes

Specify Destination DB2 Database

Select an existing connection.

Connections

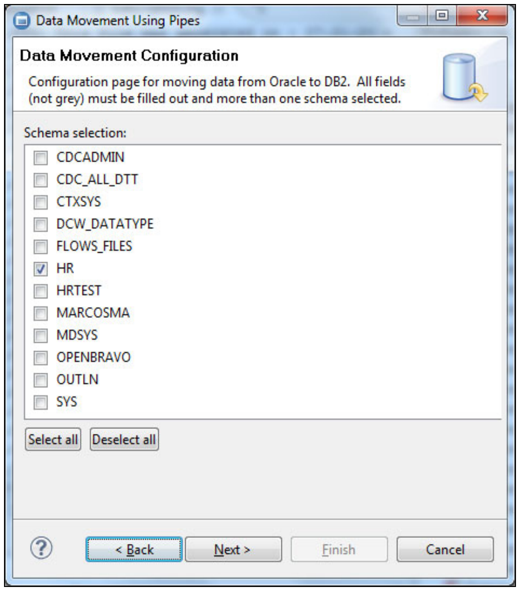
- dcwtest
- localDB2
- ORADCW
- SAMPLE
- SAMPLE1

Properties

Property	Value
Name	SAMPLE
Description	
Category	Database Connections
Database	SAMPLE
JDBC Driver Class	com.ibm.db2.jcc.DB2Driver
Class Location	C:\Program Files\IBM\SDPShared\plugins\...

Figure 3-33 Selecting the target DB2 database

5. On the third window (Figure 3-34), select the schema that you want to extract.


Data Movement Using Pipes

Data Movement Configuration

Configuration page for moving data from Oracle to DB2. All fields (not grey) must be filled out and more than one schema selected.

Schema selection:

- [ ] CDCADMIN
- [ ] CDC_ALL_DTT
- [ ] CTXSYS
- [ ] DCW_DATATYPE
- [ ] FLOWS_FILES
- [x] HR
- [ ] HRTEST
- [ ] MARCOSMA
- [ ] MDSYS
- [ ] OPENBRAVO
- [ ] OUTLN
- [ ] SYS

Figure 3-34 Selecting source schemas

6. On the fourth window (Figure 3-35), select the tables that you want to extract.

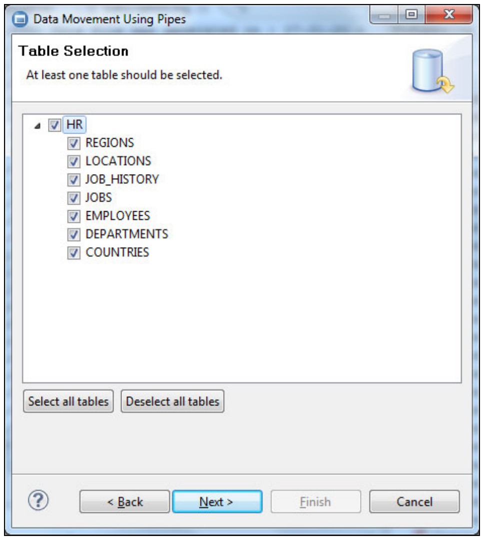

Data Movement Using Pipes

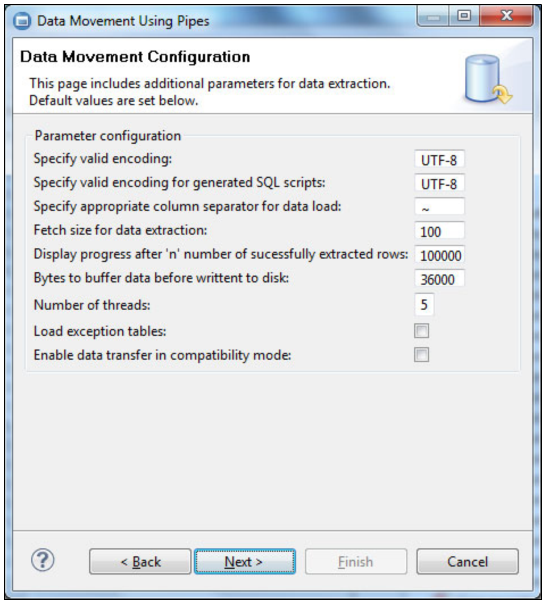
Table Selection

At least one table should be selected.

- [x] HR
  - [x] REGIONS
  - [x] LOCATIONS
  - [x] JOB_HISTORY
  - [x] JOBS
  - [x] EMPLOYEES
  - [x] DEPARTMENTS
  - [x] COUNTRIES

Figure 3-35 Selecting the source tables

7. On the fifth window (Figure 3-36), specify the data movement configuration parameters. Novice users can use the default selections.



Data Movement Configuration

This page includes additional parameters for data extraction.
Default values are set below.

Parameter configuration

Specify valid encoding:	UTF-8
Specify valid encoding for generated SQL scripts:	UTF-8
Specify appropriate column separator for data load:	~
Fetch size for data extraction:	100
Display progress after 'n' number of sucessfully extracted rows:	100000
Bytes to buffer data before writtent to disk:	36000
Number of threads:	5
Load exception tables:	☐
Enable data transfer in compatibility mode:	☐

Figure 3-36 Specifying data movement configuration parameters

8. View the summary on the final window of the wizard and click **Finish**. This starts the extraction process, during which a progress indicator is displayed.

A log of the extraction is automatically displayed in Data Studio after the process is finished. The log is also stored in the Data Movement folder of your DCW project.

3.7.3 Data movement using IBM InfoSphere Federation Server

DCW also supports the movement of data using DB2 federation capabilities. DCW builds on the IBM federated solution for combining information from multiple data sources, and uses this technology to migrate data from Oracle to DB2. This allows for faster, more accurate, and more efficient movement of data.

Internally, DCW creates nicknames for selected tables and fetches data through those nicknames. After the data transfer process is complete, DCW deletes all of the nicknames and other federation objects, such as wrappers and server and user mappings.

Prerequisites

You must ensure that these prerequisites are met before continuing with this approach to data movement:

- IBM InfoSphere Federation Server must be installed on the target DB2 system. Depending on your DB2 edition, this might require additional licensing.
- The Oracle client software must be installed on the system hosting the target DB2 database.

Setting up the target DB2 database for federation

The **`FEDERATED`** parameter must be enabled on the target DB2 instance before you can move objects there. To do this by updating the database manager configuration, complete the following steps:

1. Confirm that the target DB2 instance is started. This can be done from the DB2 command line processor.
2. Run the following command to enable federation:

   ```
   db2 update dbm cfg using federated yes
   ```
3. Restart the DB2 instance.

Federation is sometimes enabled by default. You can confirm if this occurred from the DB2 CLP by running the following command:

```
db2 get database manager configuration
```

Configuring the Oracle client software

In order for DB2 to be able to federate with an Oracle database, Oracle Client software must be installed. In particular, the Oracle wrapper libraries are required.

Complete the following steps:

1. For Linux or UNIX operating systems, set the `ORACLE_HOME` environment variable on your target environment.
2. Insert the `ORACLE_HOME` variable into the `\$DB2PATH$\cfg\db2dj.ini` file.
3. Set up the Oracle client configuration file `tnsnames.ora` (in the `="\$ORACLE_HOME$\NETWORK\ADMIN` directory) by adding the following lines and changing the specified values:

   ```
   alias_name =
   (DESCRIPTION =
     (ADDRESS =
         (PROTOCOL = TCP)
   ```

```
      (HOST = oracle_host)
      (PORT = host_port)
   )
   (CONNECT_DATA =
      (SERVER = DEDICATED)
      (SERVICE_NAME = oracle_sid)
   )
)
```

4. Test the Oracle connection by running **`tnsping alias_name`** after you connect to the Oracle database through SQL Plus by running the following command:

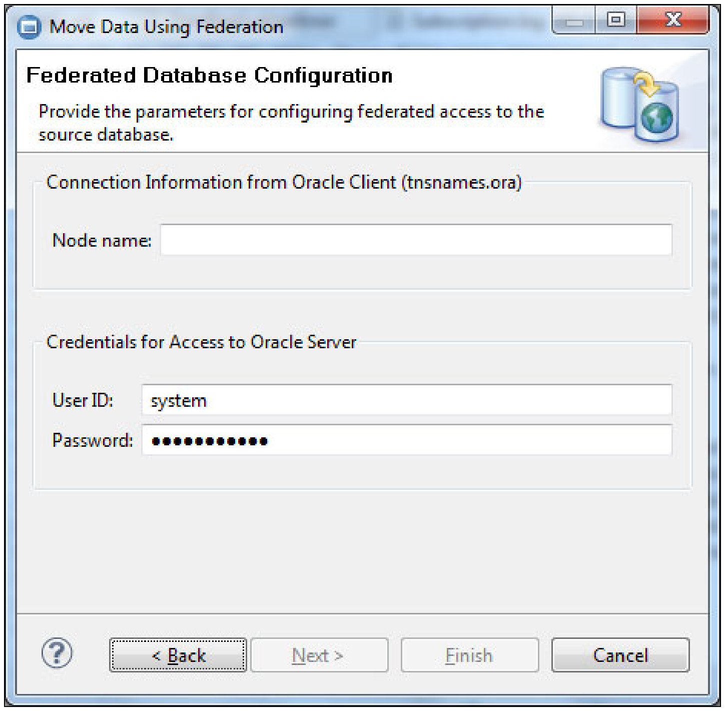
   ```
   sqlplus user/password@alias_name
   ```

Initiating data movement using federation

After your DB2 database is prepared for federation and Oracle client software is installed and configured, you can initiate federation. Complete the following steps:

1. In Project Explorer, right-click the listing for your DCW project and navigate to **Database Conversion → Move Data...**.
2. Select **Data movement using federation**. Click **Finish** to start the wizard.
3. Review the requirements on the first window of the wizard. Click **Next**.
4. Select the source Oracle database. Click **Next**.
5. Select the target DB2 database. Click **Next**.
6. On the fourth window of the wizard (Figure 3-37 on page 203), specify the node name (alias) of the source Oracle database as it is listed in your `tnsnames.ora` file and provide the user ID and password that must be used to run the data load. Click **Next**.

   The wizard automatically populates these fields on this window with the same values from the source database connection profile, but you can change them if needed.


Move Data Using Federation

Federated Database Configuration

Provide the parameters for configuring federated access to the source database.

Connection Information from Oracle Client (tnsnames.ora)

Node name:

Credentials for Access to Oracle Server

User ID: system

Password: ●●●●●●●●●●●

Figure 3-37 Specifying Oracle node details

7. Select the schemas that you want to move. Click **Next**. DCW creates wrappers for your objects, and provides a summary.
8. Click **Finish** to start the data movement using federation.

3.7.4 Data movement using IBM InfoSphere Change Data Capture

For environments where databases are progressivein nature, DCW can use InfoSphere Change Data Capture (CDC). This feature in DCW helps configure data replication, initiate data replication, and end data replication.

The CDC feature of DCW targets the small number of tables that typically hold most of the data. Data replication using CDC can be set up and started weeks before the cut-over date, while your Oracle database is still in use, and the changes are still tracked in the target DB2 database. The small amount of data in the remaining tables can be moved at the cut-over date by using the DCW extract and load technology.

Prerequisites

CDC-based data replication requires these key components:

- InfoSphere CDC license: A license is required to use the CDC software.
- Data store replication engine: This is an InfoSphere CDC process that sends or receives replicated data. The required data store replication engine processes on both the source and target servers are created when you install InfoSphere CDC.
- DBMS: This is the source or target database. You can work only with databases that are supported by InfoSphere CDC as a source for target of replicated data.
- Data store: This is an InfoSphere CDC process on a source or target server that accepts requests from an instance of Access Server and communicates with the data store replication engine to initiate and manage replication activity. Data store processes on source and target servers are created when you install InfoSphere CDC.
- Access Server: Access Server is the server application that controls access to your replication environment. You can have multiple instances of Access Server in your working environment, but you can connect only to one server at a time. For more information about installing Access Server, see *Management Console and Access Server - Installation Guide* at this address:

  http://pic.dhe.ibm.com/infocenter/soliddb/v6r5/index.jsp?topic=%2Fcom.ibm.swg.im.soliddb.universalcacheuserguide.doc%2Fdoc%2Fs0009042_install_configure_AS.html

In addition, the user must ensure that the following actions occurred before initiating data replication by using CDC:

- CDC must be installed on the source and target systems.
- Access Server and the replication servers must be up and running.
- The data store must be defined.
- The source database must have logging enabled in Archive mode, and supplemental logs must be enabled at the table and column levels.

Note: You need a separate license to install CDC. A license for CDC does not come with DCW.

Configuring data replication

To replicate data, you must configure the data replication process. The source and target tables must exist in their respective databases. In configuring data replication, you must create mappings between selected source and target tables. These mappings are identified by unique subscription names that are used to initiate and end the process.

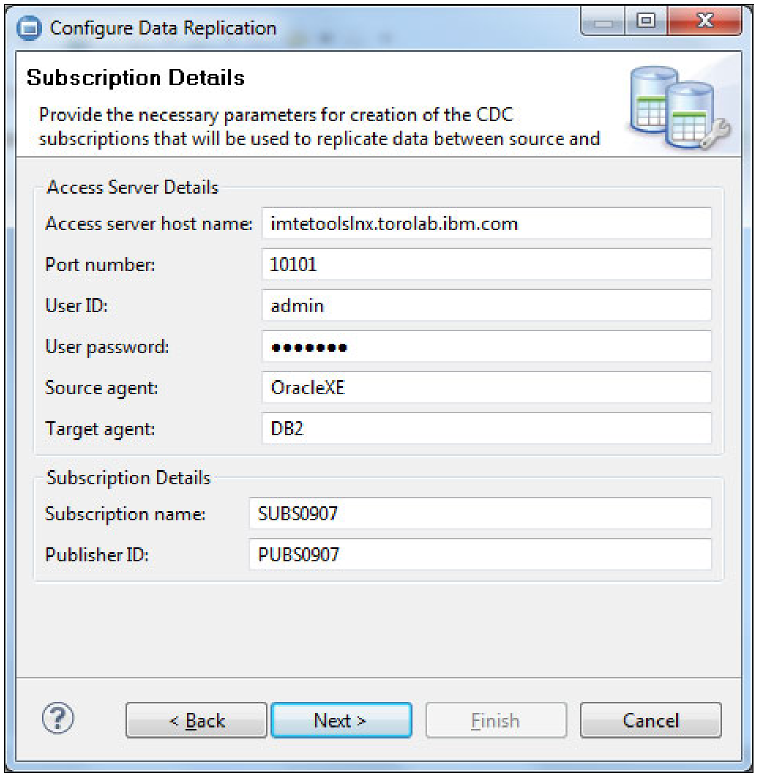
A subscription is a connection that is required to replicate data between a source data store and a target data store. A subscription contains details about the data that is being replicated and how the source data is to be applied to the target.

DCW creates subscriptions only with unique names. Editing an existing subscription is not allowed in DCW.

To configure your data replication, complete the following steps:

1. Right-click the listing for your DCW project and select **Database Conversion** → **Replicate Data through CDC...** to open the Task Selection wizard. This wizard is also accessible from the Task Launcher.
2. Select **Configure data replication** and click **Finish**. This opens the Configure data replication wizard.
3. On the first window of the wizard, make sure that you meet all the requirements that are displayed on the first window of the wizard and click **Next**.
4. In the Source field, select the connection to the source Oracle database, or create a connection by clicking **New...**. Click **Next**.
5. In the Target field, select the connection to the target DB2 database, or create a connection by clicking **New...**. Click **Next**.
6. In the Tables field, select the tables that will be replicated. Click **Next**.

7. On the Subscription Details window (Figure 3-38), specify the **Access Server Details** (host name, port, user, and password) and replication agent names. Provide a unique subscription name and publisher ID. The subscription name and publisher ID are used for replicating the data. Click **Next**.



Subscription Details

Provide the necessary parameters for creation of the CDC subscriptions that will be used to replicate data between source and

Access Server Details

Access server host name:	imtetoolslnx.torolab.ibm.com
Port number:	10101
User ID:	admin
User password:	•••••••
Source agent:	OracleXE
Target agent:	DB2

Subscription Details

Subscription name:	SUBS0907
Publisher ID:	PUBS0907

Figure 3-38 Subscription Details window in the Configure Data Replication wizard

8. On the summary window, review your settings and click **Finish** to initiate configuration of the data replication process and follow the progress.

Initiating data replication

With InfoSphere Change Data Capture, you can mirror data from source tables (using CDC replication) that are mapped to target tables.

InfoSphere CDC provides two types of mirroring:

- Continuous mirroring:

  This mirroring replicates changes to the target database on a continual basis. Use this type of mirroring when business requirements dictate that you need replication to be running continuously and you do not have a clearly defined reason to end replication at the present time.

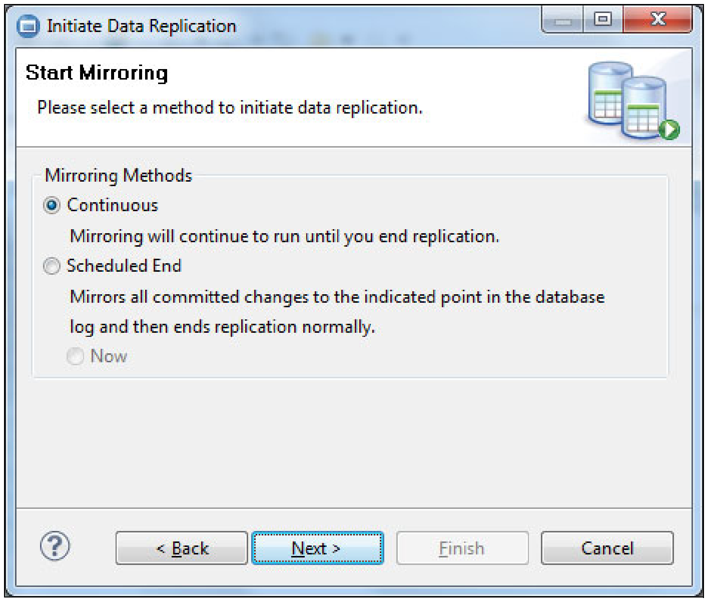
- Scheduled End (Net Change) mirroring:

  This mirroring replicates changes to the target database until a user-specified point in the source database log is reached, at which time replication is stopped. Use this type of mirroring when business requirements dictate that you replicate your data only periodically and you have a clearly defined endpoint for the eventual state of your target database.

To begin CDC data replication, complete the following steps:

1. Right-click the listing for your DCW project and select **Database Conversion** → **Replicate Data through CDC...**. This opens the Task Selection wizard.
2. Select **Initiate data replication** and click **Finish** to open the **Initiate Data Replication** wizard.
3. Specify the Access Server and replication agent details and provide a valid subscription name and publisher ID details. Click **Next**.

4. Select a mirroring method (Figure 3-39). Click **Next**.



Start Mirroring

Please select a method to initiate data replication.

Mirroring Methods

(●) Continuous

Mirroring will continue to run until you end replication.

( ) Scheduled End

Mirrors all committed changes to the indicated point in the database log and then ends replication normally.

( ) Now

Figure 3-39 Choosing a mirroring method in the Initiate Data Replication wizard

5. View the summary and then click **Finish**. This initiates the data replication process and shows the progress that occurs until replication is complete.

Ending data replication

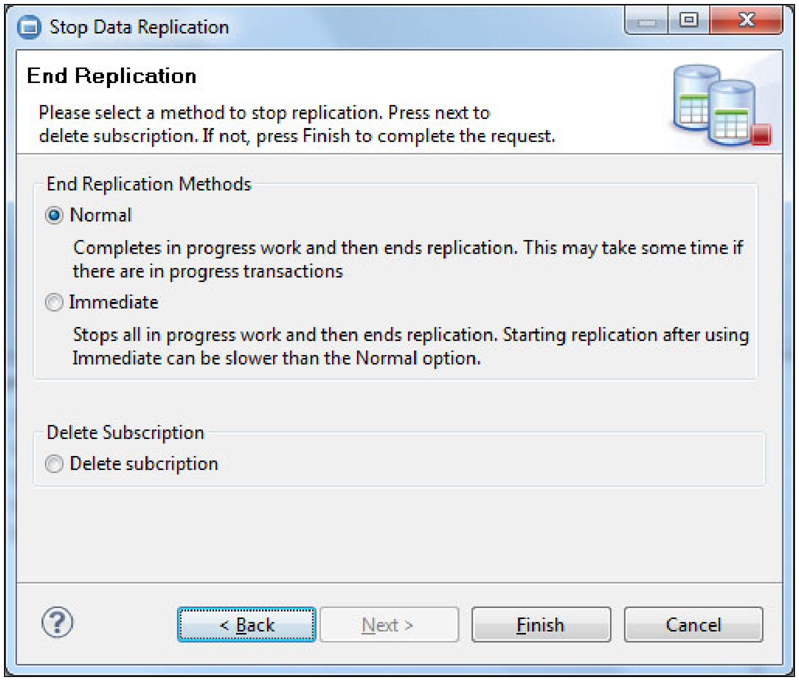
Active replication of data can be stopped in Normal mode or Immediate mode:

- Normal: In this mode, the work in progress is completed before replication is ended. Some time might elapse if there are in-progress transactions.
- Immediate: In this mode, the work in progress stops immediately and replication is ended. Starting replication using Immediate mode can be slower than using the Normal mode.

To stop data replication, complete the following steps:

1. Right-click the listing for your DCW project and select **Database Conversion** → **Replicate Data through CDC...**. This opens the Task Selection wizard.
2. Select **Stop data replication** and click **Finish**. This opens the **Stop data replication** wizard.
3. Specify the access server that you want to stop and replication agent details and provide a valid subscription name and publisher ID details. Click **Next**.

4. Select your preferred method for stopping active replication for the subscription. If you also want to delete the subscription and remove any related configuration, select **Delete subscription**, as shown in Figure 3-40. When you click **Finish**, the wizard requests that the CDC server stop data replication and deletes the subscription if that option was selected.



End Replication

Please select a method to stop replication. Press next to delete subscription. If not, press Finish to complete the request.

End Replication Methods

[✓ Normal]

Completes in progress work and then ends replication. This may take some time if there are in progress transactions

Immediate

Stops all in progress work and then ends replication. Starting replication after using Immediate can be slower than the Normal option.

Delete Subscription

Delete subcription

Figure 3-40 Selecting a method for ending replication

3.7.5 Manual data deployment

If you cannot run DCW in the environment where their target database exists, you can load data manually by using scripts.

In this scenario, you follow the process for extracting data from flat files (see "Extracting data to flat files" on page 194) and then transfer the load script and the extracted data to your target DB2 environment. Then, you use the DB2 command line processor to run the script manually.

Complete the following steps:

1. Copy the directory with the extracted data (see "Extracting data to flat files" on page 194) to the machine with the target DB2 database using your preferred storage medium.

2. Open the load script (`db2load.sql`) and change the path of the data, message, and dump folders.
3. If the OS of the machine where the extraction occurred is different from the OS of the machine where the data will be deployed, replace the directory separator to either the \ character (for Windows) or the / character (for Linux).
4. Run the `db2load.sql` script on the DB2 command line processor, as shown here:

   ```
   db2 => @db2load.sql
   ```

3.7.6 Selecting the appropriate data movement method

The data movement techniques explained in this chapter have their own advantages and disadvantages.

Here is a list that summarizes the advantages of each technique and explains when each method is likely to be most applicable:

- Extract and load data using flat files

  This is the most common method for data movement. Users can extract data from any environment, but the extracted data must be stored in intermediate flat files that are later accessed for the data load. DCW must be installed on the environment where the target DB2 server is running.

- Data movement using pipes

  If you are working with large databases and have a low storage staging area where extracted data can be stored, the data movement through pipes option is preferred. In this method, data is directly transferred to the target DB2 database without an intermediary storage medium, thus making data movement a single-step process. DCW must be installed on the environment where the target DB2 server is running.

- Data movement using federation

  When working with multiple source databases, use the federation option if it is available. This method is one of the cleanest ways to perform data movement and can be run from a local or a remote environment. Configuration involves a one-time setup process, and data movement through federation does not require a storage area to stage extracted data.

- Data movement using InfoSphere Change Data Capture

  For environments where databases are progressive and the downtime window is low, replicate using the CDC feature. This feature replicates data through mirroring without stopping the source database. For production databases, this is most appropriate way to migrate data, but this option requires a CDC license that is not provided with DCW.

- Manual deployment of data

  If you want to customize load scripts or cannot run DCW in the environment where the target DB2 server is installed, run the load script directly from the DB2 command line processor in the target environment.

3.7.7 Verifying data movement

Check the log and message files in the Data Movement folder in your DCW project. If the data transfer was successful, these files provide a row count of each table, which can be used to ensure that all of the data was moved correctly. If the transfer failed, the log files contain detailed information about the errors that occurred.

3.8 Deploying remaining objects on the target DB2 database

After moving data from the source Oracle database to the target DB2 database, run the remaining DDL scripts against the target DB2 database to complete the creation of database objects. You must ensure that you have completed the data transfer from all tables in the scope of your project.

To deploy your remaining database objects, in Project Explorer, select your DCW project and navigate to your folder with the split DDL files. You must run every file within this folder to create all your remaining objects. You should have already run the files in the Base Objects folder previously. You can run the files by completing the following steps:

1. You can open each individual file by using the SQL Editor and clicking **Run SQL** to run the DDL statements.
2. Alternatively, right-click the `.sql` file and select **Database Conversion** → **Run SQL files...**.

Depending on the order in which the DDL statements are run and the dependencies between the database objects, it is possible that you must repeat the execution of the DDL statements until all of the dependencies are satisfied.

3.9 Conclusion

The Database Conversion Workbench and Data Studio form a smart and efficient toolset that offers a holistic approach toward the database conversion process, with an automated solution for each stage plus hints and guidelines to assist you in completing your project. Together with the improved Oracle compatibility features in DB2 10.5 for Linux, UNIX, and Windows, IBM has created a base for simplifying Oracle migrations and has added a new dimension to database development in DB2.

If necessary, contact your IBM marketing representative to inquire about additional assistance and services to help you enable your applications for a DB2 database.

The latest information about the Database Conversion Workbench can be found by visiting the Database Conversion Workbench community in IBM developerWorks at this address:

`http://www.ibm.com/developerworks/data/ibmdcw`

4

Enablement scenario

Using the Oracle compatibility features of IBM DB2 10.5, you can move your Oracle database application to DB2 with few or no changes.

The previous chapters highlighted the available compatibility features of DB2 10.5, including natively supported syntax and enablement tools. This chapter presents a complete migration scenario to illustrate how the compatibility features can make conversions faster and easier. The scenario includes installing DB2 software and creating a DB2 database in the Oracle compatibility mode. It also demonstrates moving an Oracle database with a PL/SQL application to DB2 using the IBM Database Conversion Workbench (DCW).

The chapter includes these sections:

- Installing DB2 and creating an instance
- Enabling SQL compatibility
- Creating and configuring the target DB2 database
- Defining a new database user
- Using IBM Database Conversion Workbench
- Verifying enablement
- Summary

4.1 Installing DB2 and creating an instance

The enablement scenario that is presented here assumes a DB2 database server that is set up on a machine that is running a Linux operating system.

Regardless of the platform on which DB2 is installed, you must consider and satisfy all of the necessary hardware and software requirements. Review the installation requirements and options in the DB2 Information Center at this website:

http://pic.dhe.ibm.com/infocenter/db2luw/v10r5/topic/com.ibm.db2.luw.qb.server.doc/doc/r0025127.html

The following DB2 manuals are also good sources of information:

- *Getting Started with DB2 installation and administration on Linux and Windows*, GI13-2047
- *Installing DB2 Servers*, GC27-3884
- *Installing IBM Data Server Clients*, GC27-3883

Important: For the most recent software requirements, see this website:

http://www.ibm.com/software/data/db2/linux-unix-windows/sysreqs.html

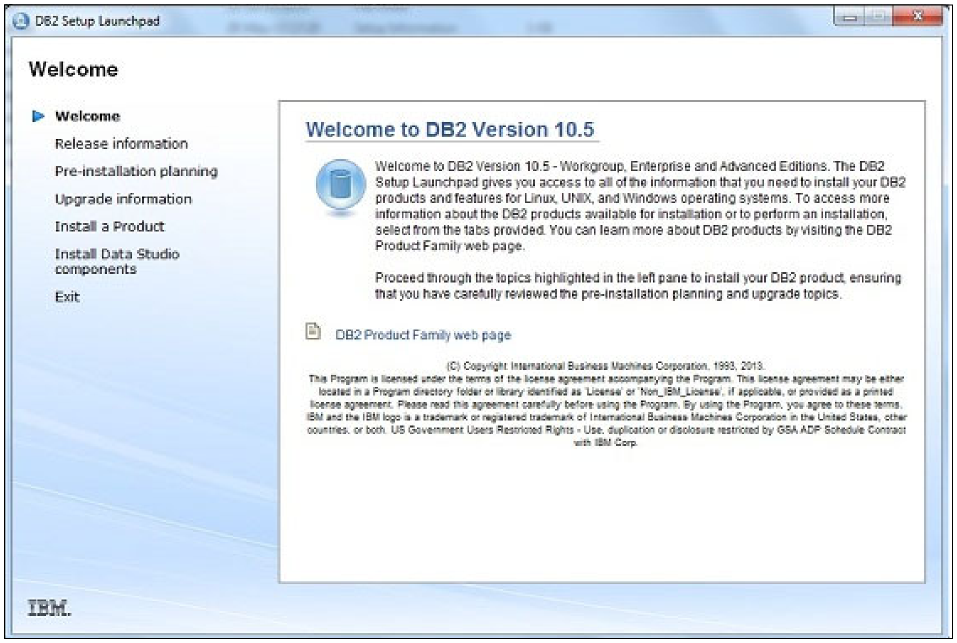
Start by obtaining the DB2 10.5 software. If you do not have an IBM Passport Advantage® account, you can download a DB2 trial package that you can use to use the software for 90 days after installation. The package is available at:

http://www.ibm.com/software/data/db2/linux-unix-windows/download.html

For purposes of this scenario, install DB2 using the **db2setup** graphical setup wizard. It can also be installed using an unattended response file. For more information about installation methods, see:

http://pic.dhe.ibm.com/infocenter/db2luw/v10r5/index.jsp?topic=%2Fcom.ibm.db2.luw.qb.server.doc%2Fdoc%2Fc0008711.html

After you download and unpack DB2 10.5 for Linux, UNIX, and Windows operating systems, run the **db2setup** command as a user with administrative privileges, such as root on Linux. This command starts the DB2 Setup Launchpad that is shown in Figure 4-1.



Welcome to DB2 Version 10.5

Welcome to DB2 Version 10.5 - Workgroup, Enterprise and Advanced Editions. The DB2 Setup Launchpad gives you access to all of the information that you need to install your DB2 products and features for Linux, UNIX, and Windows operating systems. To access more information about the DB2 products available for installation or to perform an installation, select from the tabs provided. You can learn more about DB2 products by visiting the DB2 Product Family web page.

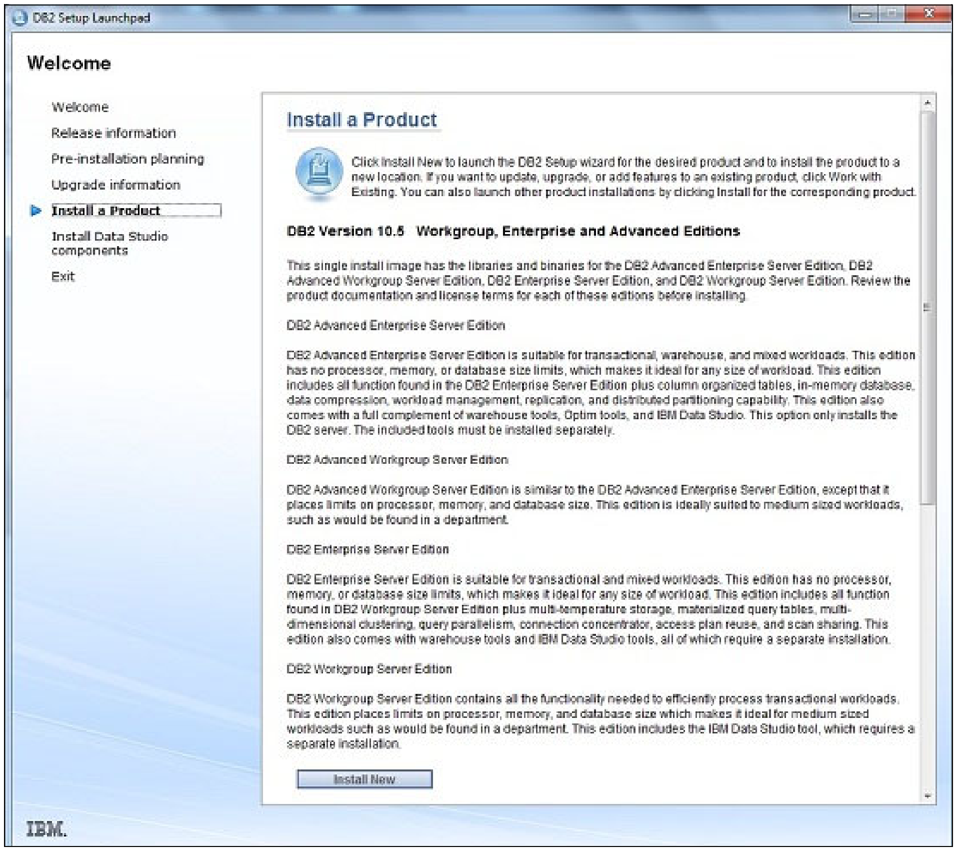
Proceed through the topics highlighted in the left pane to install your DB2 product, ensuring that you have carefully reviewed the pre-installation planning and upgrade topics.

DB2 Product Family web page

(C) Copyright International Business Machines Corporation, 1993, 2013.
This Program is licensed under the terms of the license agreement accompanying the Program. This license agreement may be either located in a Program directory folder or library identified as 'License' or 'Non_IBM_License', if applicable, or provided as a printed license agreement. Please read this agreement carefully before using the Program. By using the Program, you agree to these terms. IBM and the IBM logo is a trademark or registered trademark of International Business Machines Corporation in the United States, other countries, or both. US Government Users Restricted Rights - Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Figure 4-1 DB2 Setup Launchpad

Select the **Install a Product** option and then click **Install New** (Figure 4-2). Follow the setup wizard prompts, accepting the default values as you proceed. For the DB2 administrator, provide a new user ID (this scenario uses `db2inst1`), which is created as part of the installation.



Install a Product

Click Install New to launch the DB2 Setup wizard for the desired product and to install the product to a new location. If you want to update, upgrade, or add features to an existing product, click Work with Existing. You can also launch other product installations by clicking Install for the corresponding product.

DB2 Version 10.5 Workgroup, Enterprise and Advanced Editions

This single install image has the libraries and binaries for the DB2 Advanced Enterprise Server Edition, DB2 Advanced Workgroup Server Edition, DB2 Enterprise Server Edition, and DB2 Workgroup Server Edition. Review the product documentation and license terms for each of these editions before installing.

DB2 Advanced Enterprise Server Edition

DB2 Advanced Enterprise Server Edition is suitable for transactional, warehouse, and mixed workloads. This edition has no processor, memory, or database size limits, which makes it ideal for any size of workload. This edition includes all function found in the DB2 Enterprise Server Edition plus column organized tables, in-memory database, data compression, workload management, replication, and distributed partitioning capability. This edition also comes with a full complement of warehouse tools, Optim tools, and IBM Data Studio. This option only installs the DB2 server. The included tools must be installed separately.

DB2 Advanced Workgroup Server Edition

DB2 Advanced Workgroup Server Edition is similar to the DB2 Advanced Enterprise Server Edition, except that it places limits on processor, memory, and database size. This edition is ideally suited to medium sized workloads, such as would be found in a department.

DB2 Enterprise Server Edition

DB2 Enterprise Server Edition is suitable for transactional and mixed workloads. This edition has no processor, memory, or database size limits, which makes it ideal for any size of workload. This edition includes all function found in DB2 Workgroup Server Edition plus multi-temperature storage, materialized query tables, multi-dimensional clustering, query parallelism, connection concentrator, access plan reuse, and scan sharing. This edition also comes with warehouse tools and IBM Data Studio tools, all of which require a separate installation.

DB2 Workgroup Server Edition

DB2 Workgroup Server Edition contains all the functionality needed to efficiently process transactional workloads. This edition places limits on processor, memory, and database size which makes it ideal for medium sized workloads such as would be found in a department. This edition includes the IBM Data Studio tool, which requires a separate installation.

Figure 4-2 Installing DB2

When the process concludes, the setup wizard opens a window where you confirm that the DB2 installation was successful.

4.2 Enabling SQL compatibility

As described in Chapter 2, “Language compatibility features” on page 21, you must enable the Oracle compatibility mode in the DB2 instance before you create the DB2 database. You enable SQL compatibility by setting the `DB2_COMPATIBILITY_VECTOR` and `DB2_DEFERRED_PREPARE_SEMANTICS` registry variables. Then, you must stop and restart the instance using the **`db2stop`** and **`db2start`** commands, as shown in Example 4-1.

Example 4-1 Enabling Oracle compatibility mode in DB2

```
/home/db2inst1>db2set DB2_COMPATIBILITY_VECTOR=ORA
/home/db2inst1>db2set DB2_DEFERRED_PREPARE_SEMANTICS=YES
/home/db2inst1>db2set -all
[i] DB2_DEFERRED_PREPARE_SEMANTICS=YES
[i] DB2_COMPATIBILITY_VECTOR=ORA
[i] DB2COMM=TCPIP
[i] DB2AUTOSTART=YES
[g] DB2SYSTEM=oc2522778176.ibm.com
[g] DB2INSTDEF=db2inst1
[g] DB2ADMINSERVER=dasusr1
/home/db2inst1>db2stop
2012-03-06 18.16.46     0   0   SQL1064N  DB2STOP processing was
successful.
SQL1064N  DB2STOP processing was successful.
/home/db2inst1>db2start
03/06/2012 18:16:54     0   0   SQL1063N  DB2START processing was
successful.
SQL1063N  DB2START processing was successful.
```

4.3 Creating and configuring the target DB2 database

Example 4-2 shows a script that uses the **`CREATE DATABASE`** command and other, post-creation commands. Run this script while logged in to the database server as the instance owner (in this scenario, as the **`db2inst1`** user). When logged in locally, the **`CONNECT TO`** command does not require a user ID and password because these values are taken implicitly from the operating system.

Example 4-2 Script to create the DB2 database

```
CREATE DATABASE sales AUTOMATIC STORAGE YES ON /home/db2inst1 PAGESIZE
32 K;
```

```
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 the script, save it in a text file and run the following command:

```
db2 -tvf <scriptname>
```

Important: To ensure that the DB2 Oracle compatibility features function correctly, create the database in Unicode, which is the default code page for all new DB2 databases.

The command parameters for this enablement example scenario activate certain DB2 features, which are explained below, that can help simplify database design, future management, and initial tuning. The scenario also uses automation features that are enabled by default. You do not need to specify parameters to benefit from these automation features.

- Automatic storage

  When you use this feature, which is enabled by default, you do not have to configure the table space sizes or placement or be concerned about 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`) or on a path that is specified by the user.

- Page size 32 KB

  The default page size that is used by the **`CREATE DATABASE`** command is 4 KB, but you can override this value by selecting 32 KB, which is the largest page size available. This page size ensures a seamless migration of tables with both small and large row sizes.

Example 4-2 on page 217 also defines the following parameters:

- User temporary table space

  A user temporary table space is used to allow the future creation of a global temporary table.

- Automatic object revalidation

  You change the database **`AUTO_REVAL`** configuration parameter to **`DEFERRED_FORCE`**, as explained in 2.1.1, "SQL compatibility setup" on page 22.

- Decimal floating point number rounding

  You set the **`DECFLT_ROUNDING`** database configuration parameter to more closely mimic the Oracle database behavior.

In addition, you also use the following related features of DB2 when you create the database:

- Autonomic memory management

  The scenario makes full use of the DB2 self-tuning memory manager and autonomic features. By default, many DB2 memory heaps are automatically tuned by the self-tuning memory manager, including buffer pools, sort memory, hash join memory, and lock memory.

  The self-tuning memory manager dynamically adjusts both the total memory consumption of the database and the distribution of that memory to various purposes and needs. Thus, the self-tuning memory manager can automatically adjust pool sizes as needed to balance between different requirements to achieve optimal performance. These adjustments happen every few minutes in response to changing workload demands while the database is active.

- Physical database design and storage management

  Physical database design practices for DB2 are similar to the ones in an Oracle database, although there are differences. For example, DB2 provides multidimensional clustering (MDC) that is not offered by Oracle, and Oracle provides bitmap indexes and list partitioning. Similarly, DB2 provides easy automatic storage to simplify the management of data placement across several devices and file systems.

  Although similar in principle to the Oracle database storage model, the DB2 features use some different terminology and semantics. For an overview of preferred practices for both physical database design and database storage management, visit the IBM developerWorks website for DB2 at:

  `http://www.ibm.com/developerworks/data/bestpractices/`

  The section about physical database design offers recommendations for designing indexes and materialized query tables (materialized views), table clustering and multi-dimensional clustering, table partitioning (range partitioning), database partitioning, and using the EXPLAIN facility to improve design choices. The section about database storage includes guidelines and recommendations for physical and logical volume design, RAID array use and striping, placement of transaction logs and data, usage of file systems versus raw devices, registry variable and configuration parameter settings, and automatic storage.

4.4 Defining a new database user

Although there are various authentication methods available in DB2, such as LDAP, Kerberos, or client authentication, the most common method is server authentication. With server authentication, the user ID and password that is needed to access the database are verified using the database server's operating system mechanisms.

Connections to the DB2 database for this scenario are made with the **`sales`** user, which has database administrator (DBADM) authority. To create this user, run the following Linux command (with root authority):

```
#useradd -m -d /home/sales sales
```

After the user is created, grant it the DBADM authority through a **`db2inst1`** shell session:

```
/home/db2inst1>db2 "CONNECT TO sales"
/home/db2inst1>db2 "GRANT DBADM ON DATABASE TO USER sales"
```

Important: Although uppercase DB2 commands and SQL statements are frequently used in this book, the commands, statements, and object names in DB2 are not case-sensitive. Unless stated otherwise, you can use any case when you run these commands.

4.5 Using IBM Database Conversion Workbench

You can use the DCW to move the database objects, table data, and PL/SQL routines from the Oracle source database to the DB2 database in the enablement scenario.

The DDL scripts and data that are used for this scenario are available for download at the IBM Redbooks website. For the download details, see Appendix F, "Additional material" on page 425. Appendix E, "Code samples" on page 351 includes the DDL scripts.

4.5.1 Getting started

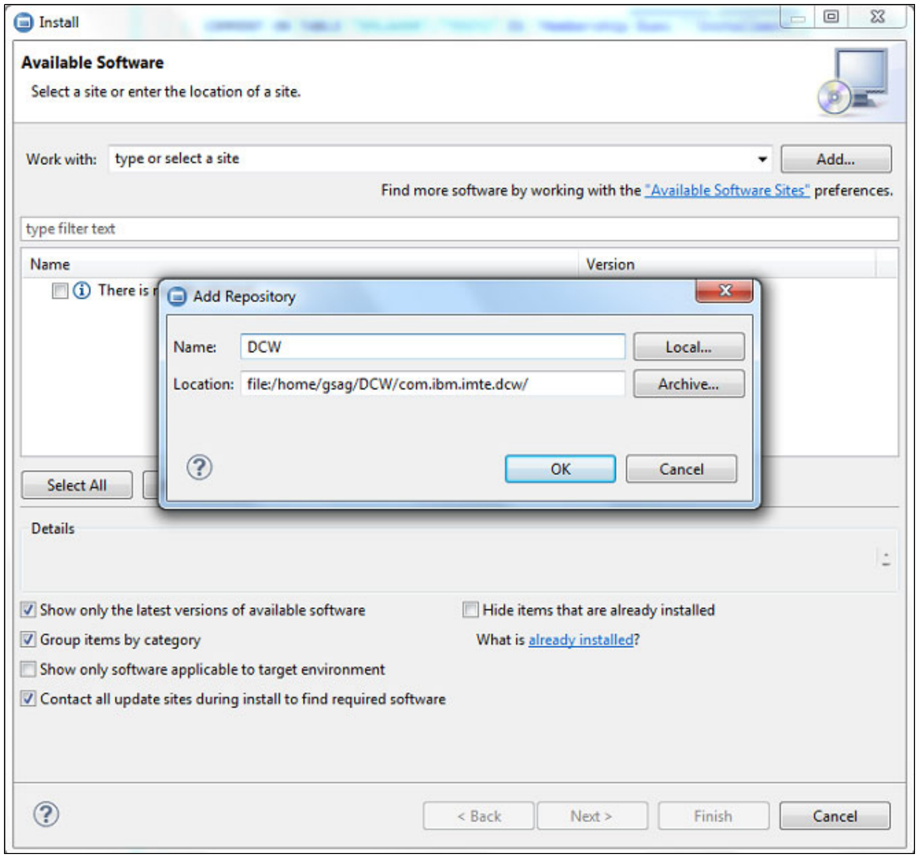
Begin by downloading the latest Database Conversion Workbench from the following address:

https://www14.software.ibm.com/webapp/iwm/web/preLogin.do?source=swg-idcw

Important: The IBM Database Conversion Workbench is a plug-in for IBM Data Studio, so you should install Data Studio on your machine before installing the DCW. At the time of writing, the current version of IBM Data Studio is Version 4.1. You can download IBM Data Studio at no additional cost from `http://www.ibm.com/developerworks/downloads/im/data/`.

Complete the following steps:

1. Extract the contents of the plug-in compressed archive to your chosen plug-in directory.
2. Add the plug-in directory in Data Studio (Figure 4-3) by clicking **Help** → **Install New Software**. After the Install wizard window opens, click **Add** and click **Local** to select the directory where the plug-in was extracted. Provide a name for this repository and click **OK**.


Install
Available Software
Select a site or enter the location of a site.
Work with:
type or select a site
Add...
Find more software by working with the "Available Software Sites" preferences.
type filter text
Name
Version
There is
Add Repository
Local...
Archive...
OK
Cancel
Select All
Details
Hide items that are already installed
What is already installed?
< Back
Next >
Finish
Cancel

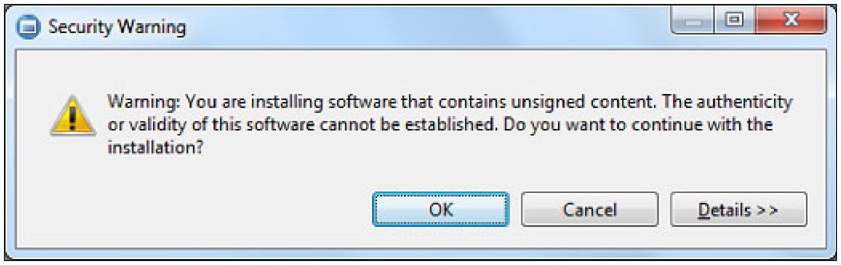
Name: DCW

Location: file:/home/gsag/DCW/com.ibm.imte.dcw/

- [✓] Show only the latest versions of available software
- [✓] Group items by category
- [ ] Show only software applicable to target environment
- [✓] Contact all update sites during install to find required software
- [ ] Hide items that are already installed

Figure 4-3 Adding a DCW plug-in location

3. Install the IBM Database Conversion Workbench plug-in. Following the wizard prompts, review and accept the terms of the license and select **Finish**. You might see a warning indicating that you are installing an unsigned plug-in; click **OK** (see Figure 4-4) to proceed with the installation.

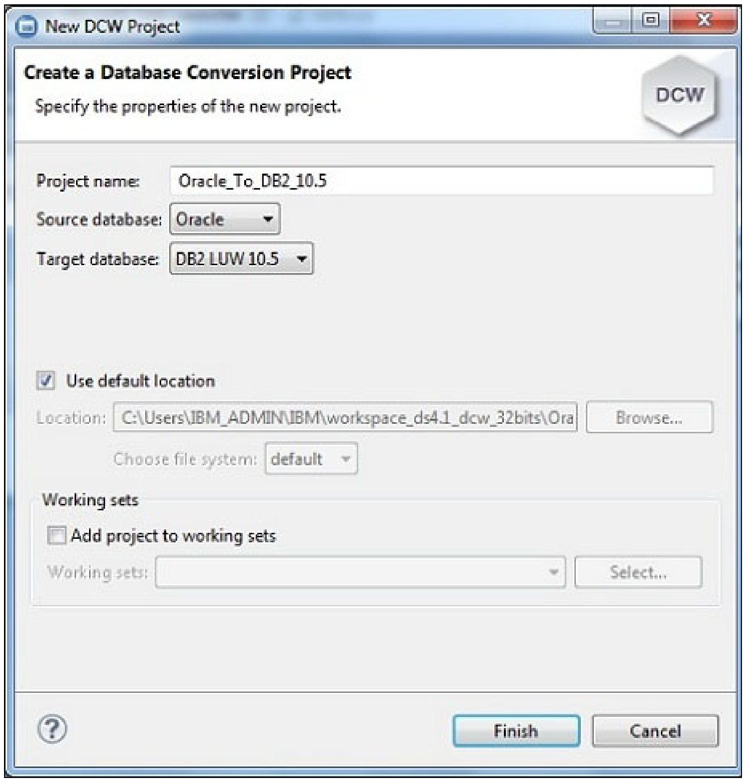


Warning: You are installing software that contains unsigned content. The authenticity or validity of this software cannot be established. Do you want to continue with the installation?

Figure 4-4   Unsigned plug-in warning during DCW installation

4. After the installation is completed, restart Data Studio.

To work with the Database Conversion Workbench in Data Studio, you use the Database Conversion perspective. Switch to this perspective if it is not active when the Data Studio opens. The first step is to create a DCW project by clicking **File** → **New** → **DCW Project**, and then you can specify the project name and select the source and target databases (Figure 4-5).


New DCW Project

Create a Database Conversion Project
Specify the properties of the new project.

Project name: Oracle_To_DB2_10.5
Source database: Oracle
Target database: DB2 LUW 10.5

[✓] Use default location
Location: C:\Users\IBM_ADMIN\IBM\workspace_ds4.1_dcw_32bits\Ora
Choose file system: default

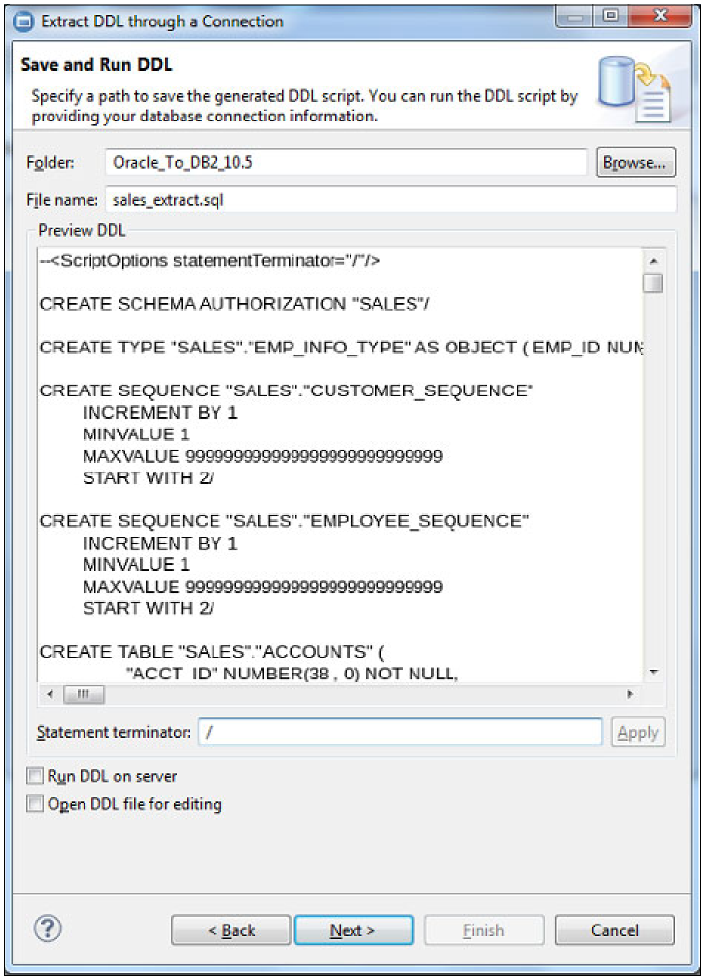
Working sets
[ ] Add project to working sets
Working sets:

Figure 4-5 Creating a DCW project

4.5.2 Extracting DDL and PL/SQL objects

The DCW offers two methods of extracting the database object and routine DDL statements, sometimes referred to as the database schema. Both methods are accessible in the DCW project menu under the Extract DDL option.

One method, often called *online extraction*, is useful when you can establish a direct connection from the DCW workstation to the Oracle database server. It uses an existing Data Studio connection to read the object definitions directly from the source database. When using this method, you can choose schemas and object types to be extracted, define the statement terminator, and preview the generated script before saving it, as shown in Figure 4-6.


Extract DDL through a Connection
Save and Run DDL
Specify a path to save the generated DDL script. You can run the DDL script by providing your database connection information.
Folder:
Oracle_To_DB2_10.5
Browse...
File name:
sales_extract.sql
Preview DDL
Statement terminator:
/
Apply
Run DDL on server
Open DDL file for editing
< Back
Next >
Finish
Cancel

```
--<ScriptOptions statementTerminator="/"/>

CREATE SCHEMA AUTHORIZATION "SALES"/

CREATE TYPE "SALES"."EMP_INFO_TYPE" AS OBJECT ( EMP_ID NUM

CREATE SEQUENCE "SALES"."CUSTOMER_SEQUENCE"
      INCREMENT BY 1
      MINVALUE 1
      MAXVALUE 9999999999999999999999999999
      START WITH 2/

CREATE SEQUENCE "SALES"."EMPLOYEE_SEQUENCE"
      INCREMENT BY 1
      MINVALUE 1
      MAXVALUE 9999999999999999999999999999
      START WITH 2/

CREATE TABLE "SALES"."ACCOUNTS" (
            "ACCT_ID" NUMBER(38 , 0) NOT NULL,
```

Figure 4-6 Extracting the source DDL using a direct database connection

Another method, called *offline extraction*, is useful when it is not possible to establish a direct connection to the source database. In that case, the DCW generates a SQL*Plus script that you can run on the Oracle database server to create the complete database DDL script. That script is later imported into the DCW project.

Before you proceed with the extraction of database objects and data, you must create a connection to the Oracle database in Data Studio. For more information, see the Data Studio documentation at the following address:

`http://pic.dhe.ibm.com/infocenter/dstudio/v4r1/topic/com.ibm.datatools.qrytune.configothers.doc/topics/configuretuning_ds.html`

For our sample project, we import an existing Oracle database script. You can find the source code in Appendix E, "Code samples" on page 351 or download it from the IBM Redbooks website.

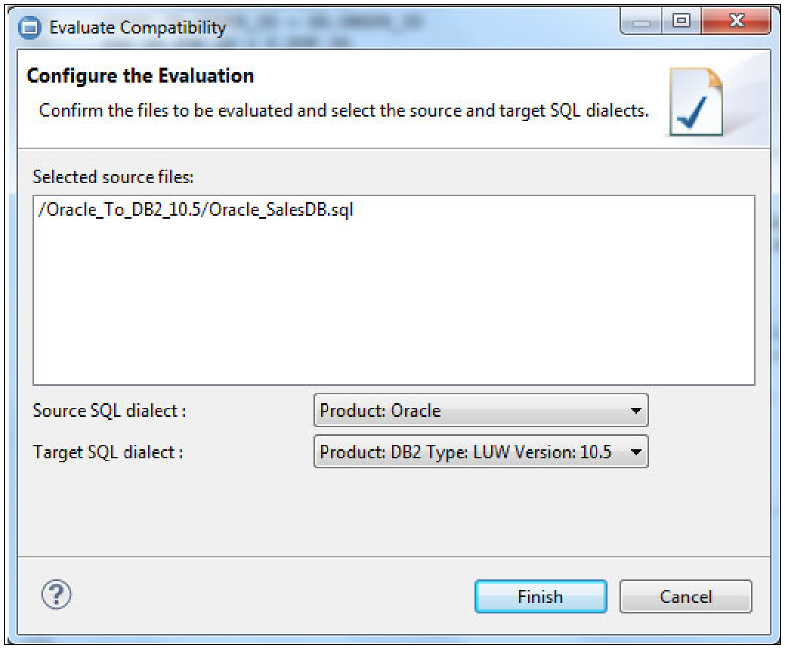
Right-click your project, select **Database Conversion** → **Import a DDL file...**, and then select the file containing the DDL statements. The imported script displays in the DCW project in Data Studio.

> **Important:** Whether extracting the source DDL using a direct database connection or importing an existing script, make sure that the statement terminator is set to the *forward slash* symbol ("/") to avoid conflicts with the default terminator, the semicolon ("**;**") found inside the PL/SQL code.

4.5.3 Compatibility evaluation

With a simple click of a button, the IBM Database Conversion Workbench can analyze Oracle DDL and PL/SQL statements and create an assessment report that outlines the compatibility of the source Oracle database with DB2. The compatibility evaluation report includes an executive summary and a detailed list of DDL and PL/SQL incompatibilities, and some suggested workarounds to fix the incompatible code.

Begin by right-clicking the `.sql` file containing the source DDL and select **Database Conversion** → **Evaluate Compatibility**. The Compatibility Evaluation wizard opens with the option to select the source and target database (Figure 4-7).



Configure the Evaluation

Confirm the files to be evaluated and select the source and target SQL dialects.

Selected source files:

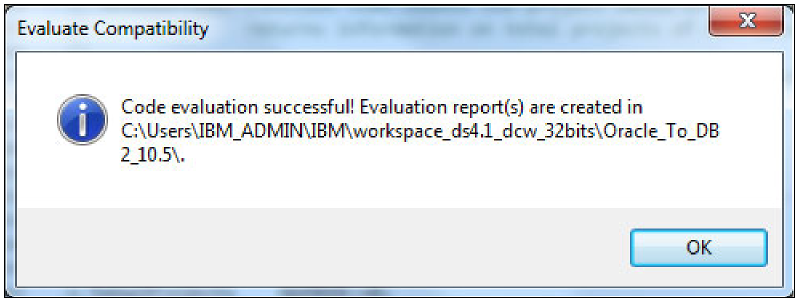
/Oracle_To_DB2_10.5/Oracle_SalesDB.sql

Source SQL dialect : Product: Oracle

Target SQL dialect : Product: DB2 Type: LUW Version: 10.5

Figure 4-7 Compatibility Evaluation wizard window

When you click **Finish**, the DCW generates an encrypted XML file with the extension `.xmle`, containing the summary of the source script. You see the message shown in Figure 4-8.



Code evaluation successful! Evaluation report(s) are created in C:\Users\IBM_ADMIN\IBM\workspace_ds4.1_dcw_32bits\Oracle_To_DB2_10.5\.

Figure 4-8 Evaluation success

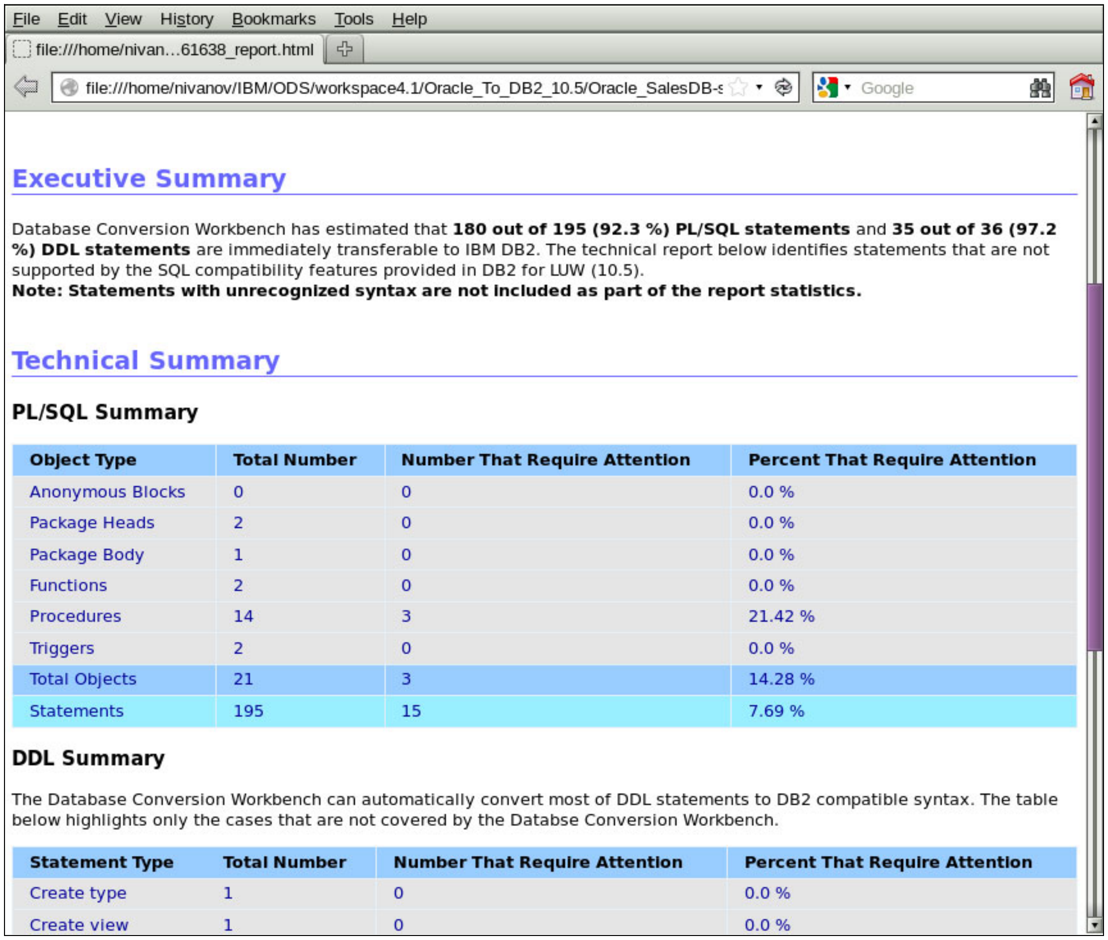
The name of the generated file follows this syntax:

`<source_script_name>-<timestamp>_report.xmle`.

Attach this file to an email message and send it to `askdcw@ca.ibm.com`. You receive an automated response shortly, which contains the formatted HTML compatibility report that you can open using any web browser.

Important: The `.xmle` file does not contain any source code. It contains only the list of detected incompatibilities.

The evaluation report for our sample application (Figure 4-9) indicates that most of its SQL and PL/SQL statements run in DB2 without changes. Among the detected incompatibilities is the use of Oracle XMLType and the DBMS_LOB package, which we address later. Some issues, such as the NUMBER data type precision exceeding the DB2 limit, are listed in the Informational section. They are resolved automatically during the DCW conversion.


File Edit View History Bookmarks Tools Help
file:///home/nivan…61638_report.html
file:///home/nivanov/IBM/ODS/workspace4.1/Oracle_To_DB2_10.5/Oracle_SalesDB-s
Google

Executive Summary

Database Conversion Workbench has estimated that **180 out of 195 (92.3 %) PL/SQL statements** and **35 out of 36 (97.2 %) DDL statements** are immediately transferable to IBM DB2. The technical report below identifies statements that are not supported by the SQL compatibility features provided in DB2 for LUW (10.5).
Note: Statements with unrecognized syntax are not included as part of the report statistics.

Technical Summary

PL/SQL Summary

Object Type	Total Number	Number That Require Attention	Percent That Require Attention
Anonymous Blocks	0	0	0.0 %
Package Heads	2	0	0.0 %
Package Body	1	0	0.0 %
Functions	2	0	0.0 %
Procedures	14	3	21.42 %
Triggers	2	0	0.0 %
Total Objects	21	3	14.28 %
Statements	195	15	7.69 %

DDL Summary

The Database Conversion Workbench can automatically convert most of DDL statements to DB2 compatible syntax. The table below highlights only the cases that are not covered by the Databse Conversion Workbench.

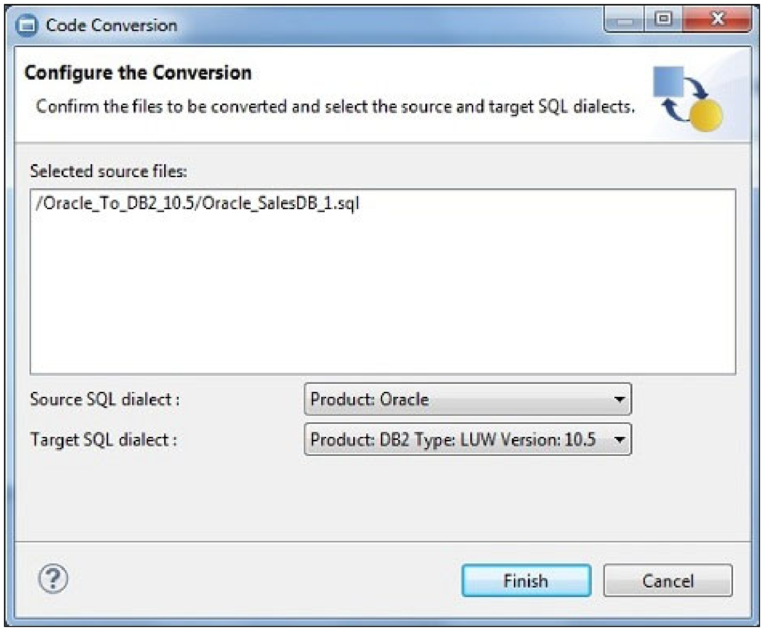
Statement Type	Total Number	Number That Require Attention	Percent That Require Attention
Create type	1	0	0.0 %
Create view	1	0	0.0 %

Figure 4-9 The sample application evaluation report

4.5.4 Conversion

The Database Conversion Workbench offers a powerful tool that can automatically convert certain known Oracle syntax incompatibilities to the DB2 compatible syntax with a simple click.

Begin by right-clicking the `.sql` file containing the Oracle DDL statements and select **Database Conversion** → **Convert Code...**. The Code Conversion wizard opens with the option to select the source and target databases (Figure 4-10).


Code Conversion
Configure the Conversion
Confirm the files to be converted and select the source and target SQL dialects.
Finish
Cancel

Selected source files:
/Oracle_To_DB2_10.5/Oracle_SalesDB_1.sql

Source SQL dialect : Product: Oracle
Target SQL dialect : Product: DB2 Type: LUW Version: 10.5

Figure 4-10 Selecting the source and target databases in the Code Conversion wizard

Important: The Code Conversion wizard expects the statement terminators in the source file to be *forward slash* symbols ("/"), which is the standard terminator for Oracle SQL*Plus utility, and automatically changes them to the *@ symbol*, which is a commonly used terminator in DB2 CLP scripts.

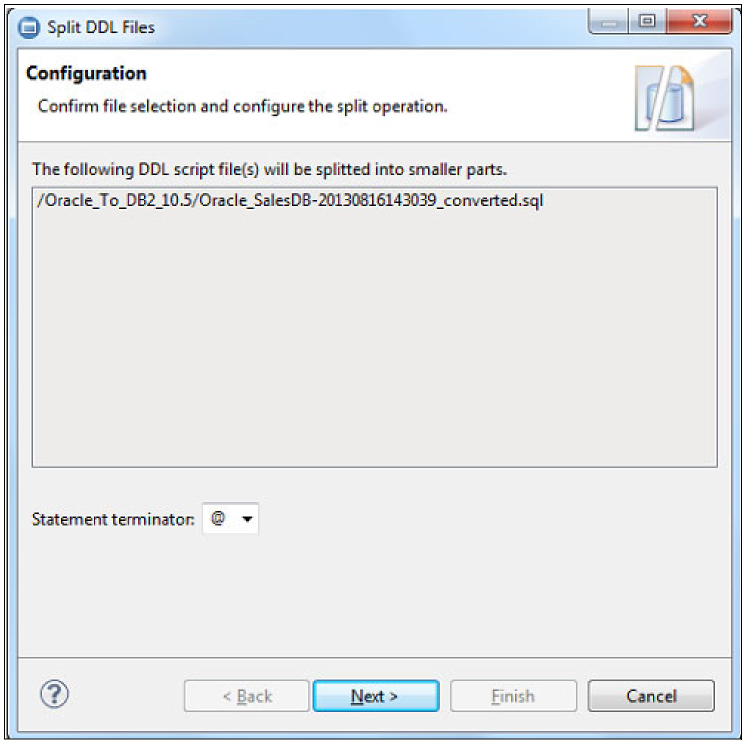
The converted DDL file is in your DCW Project folder. Its name follows the pattern `<original_file_name>-<timestamp>_converted.sql`.

4.5.5 Splitting DDL

After the code conversion is complete, you can begin deployment of the objects in the target DB2 database. To simplify this task, it is often useful to split the monolithic DDL script, containing all of the database objects, into smaller files to simplify deployment and troubleshooting. The DCW has a useful function to convert a single DDL file into multiple files that are organized by object type. This function helps streamline your Oracle to DB2 conversion process by providing a much more intuitive code organization that helps you run the files in a logical sequence of dependencies. Furthermore, it breaks down a large DDL file into smaller and more manageable components, which in a larger project can allow several developers to work in parallel.

Right-click the converted DDL file in the DCW Project and select **Database Conversion** → **Split DDL Files**.

The Split SQL Files wizard opens. Select the appropriate **Statement terminator**, as shown in Figure 4-11. We recommend using the @ symbol, which follows the convention of the DB2 CLP scripts.


Split DDL Files

Configuration

Confirm file selection and configure the split operation.

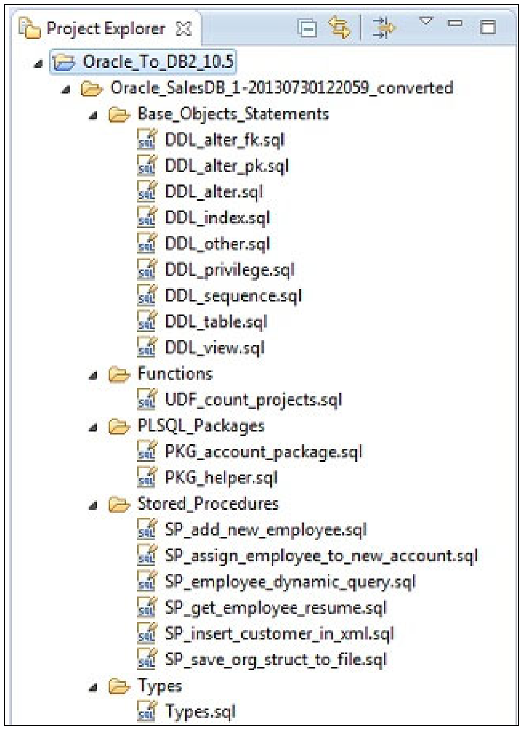
The following DDL script file(s) will be splitted into smaller parts.

/Oracle_To_DB2_10.5/Oracle_SalesDB-20130816143039_converted.sql

Statement terminator: @

Figure 4-11 Split DDL wizard - select the statement terminator

After clicking **Next**, you see the list of objects that are identified by the wizard and a confirmation window, showing what folders and files will be created. After the wizard completes, you find the new files in the DCW project, as you can see in Figure 4-12.



```
Oracle_To_DB2_10.5
  Oracle_SalesDB_1-20130730122059_converted
    Base_Objects_Statements
      DDL_alter_fk.sql
      DDL_alter_pk.sql
      DDL_alter.sql
      DDL_index.sql
      DDL_other.sql
      DDL_privilege.sql
      DDL_sequence.sql
      DDL_table.sql
      DDL_view.sql
    Functions
      UDF_count_projects.sql
    PLSQL_Packages
      PKG_account_package.sql
      PKG_helper.sql
    Stored_Procedures
      SP_add_new_employee.sql
      SP_assign_employee_to_new_account.sql
      SP_employee_dynamic_query.sql
      SP_get_employee_resume.sql
      SP_insert_customer_in_xml.sql
      SP_save_org_struct_to_file.sql
    Types
      Types.sql
```

Figure 4-12 Split DDL wizard creates a directory tree with new scripts

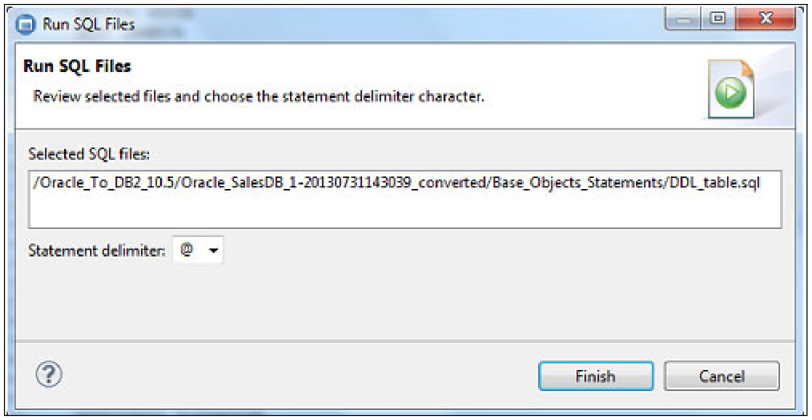
4.5.6 Deploying objects

With the extraction, conversion, and split process completed, you can now create tables and other necessary objects in the DB2 database and load the extracted data. Do not deploy any PL/SQL objects now because they might need modification.

This step involves running the extracted DDL statements against the target DB2 database to re-create the database objects.

First, use the DDL files generated by the Split DDL wizard to create tables (`DDL_table.sql`) in the DB2 target database. This can be done by using the SQL Editor in Data Studio. Double-click any SQL file to open it in the editor and select **Run** → **Run SQL**. When prompted, choose an existing connection to the target DB2 database or create a connection. The results and status are displayed in the SQL Results tab.

For this enablement scenario, we use the functionality that provided by the DCW that allows us to run multiple files at once. You can select multiple files or a folder containing SQL scripts, then right-click the selection and select **Database Conversion** → **Run SQL Files...**. In the Run SQL Files window, set the statement delimiter to "**@**", as shown in Figure 4-13, and choose the connection profile of your target database.


Run SQL Files

Review selected files and choose the statement delimiter character.

Selected SQL files:
/Oracle_To_DB2_10.5/Oracle_SalesDB_1-20130731143039_converted/Base_Objects_Statements/DDL_table.sql

Statement delimiter: @

Figure 4-13 Running SQL scripts

When the deployment is complete, you see a message similar to Figure 4-14. Each statement result displays in the SQL Results tab.

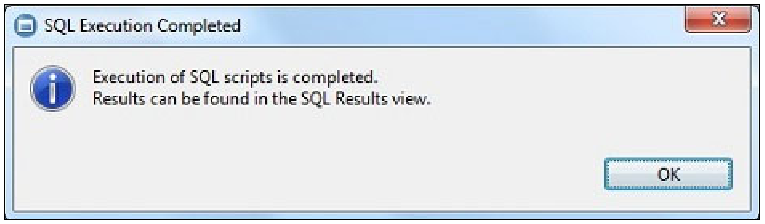

Execution of SQL scripts is completed.
Results can be found in the SQL Results view.

Figure 4-14 Successful completion of SQL scripts

Important: When deploying sequences, you might want to reset their current values to avoid generating duplicate values or gaps. Use an **`ALTER SEQUENCE`** statement similar to the following one:

```
ALTER SEQUENCE myseq RESTART WITH <new_value>
```

In our sample application, one of the tables, CUSTOMERS, has a function-based index on an XML expression. However, DB2 10.5 does not yet directly support function-based indexes and the Oracle XMLTYPE data type. Although the XMLType column is automatically converted to the DB2 XML type by the DCW, the creation of the index SALES.CUSTOMER_CITY_IND fails, as you can confirm by viewing the SQL Results tab for the script `DDL_index.sql`.

The usual workaround for an Oracle function-based index in DB2 is to create a generated column that is derived from the index expression and then build an index on that column. However, in this case, the index expression involves XMLTYPE, as shown in Example 4-3.

Example 4-3 Index that is based on an Oracle XMLTYPE function

```
CREATE INDEX customer_city_ind ON customers a
(XMLType.GETSTRINGVAL(

XMLType.EXTRACT(a.cust_details_xml,'//customer-details/addr/city/text()
'))
)
```

DB2 does not support XMLTYPE and the generated column cannot be created. However, the IBM DB2 pureXML feature offers a solution. You can create an index on an XML expression, as shown in Example 4-4.

Example 4-4 DB2 XML index

```
CREATE INDEX sales.customer_city_ind ON sales.customers
(cust_details_xml)
GENERATE KEYS USING XMLPATTERN '//customer-details/addr/city/text()'
AS SQL VARCHAR(50)
```

You can even enable case-insensitive searches, such as by the customer city name, by creating a case-insensitive XML index, as shown in Example 4-5.

Example 4-5 Creating a case-insensitive XML index

```
CREATE INDEX sales.customer_city_ind ON sales.customers
(cust_details_xml)
```

```
GENERATE KEYS USING XMLPATTERN
'//customer-details/addr/city/fn:upper-case(.)'
AS SQL VARCHAR(60)
```

Open the `DDL_index.sql` file and replace the DDL statement for `SALES.CUSTOMER_CITY_IND` with one using DB2 pureXML.

Now you can repeat the deployment of `DDL_index.sql`. Check the index deployment logon SQL Results tab to verify that the XML index is created.

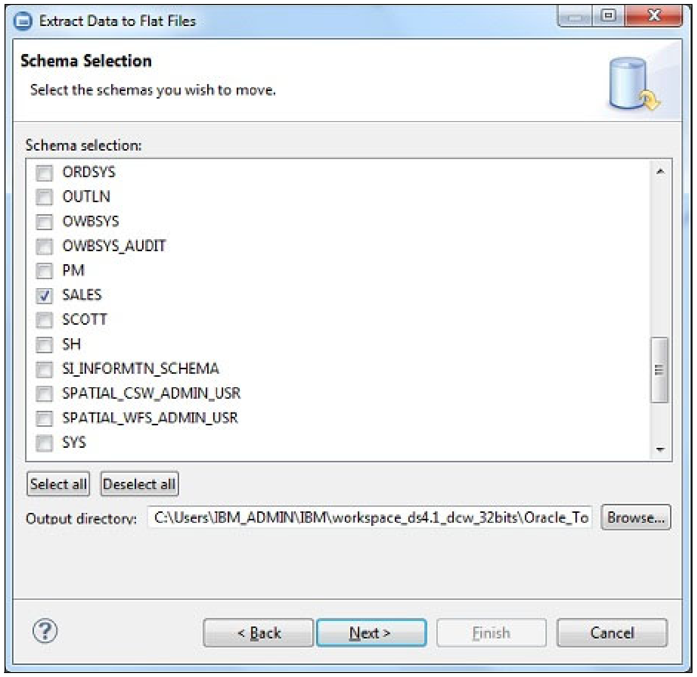
4.5.7 Extracting and loading data from files

After the target DB2 database is created and set up, you can move your data from the source Oracle database to the target DB2 database. The Database Conversion Workbench provides multiple methods to transfer data from one database to another: flat files, pipes, and federation. For this enablement scenario, we use flat files. For a description of other data movement methods, see Chapter 3, “Conversion process and enablement tools” on page 159 and the DCW documentation at the following address:

http://www.ibm.com/developerworks/data/ibmdcw

The Data Movement wizard provides an easy to use method to transfer the data for entire schemas. Right-click the conversion project and select **Database Conversion** → **Extract Data...**. The Extract Data to Flat Files wizard opens. It guides you through the required steps:

1. Select the source Oracle database and choose the schemas to extract, as shown in Figure 4-15. Click **Next**.



Schema selection:

- [ ] ORDSYS
- [ ] OUTLN
- [ ] OWBSYS
- [ ] OWBSYS_AUDIT
- [ ] PM
- [✓] SALES
- [ ] SCOTT
- [ ] SH
- [ ] SI_INFORMTN_SCHEMA
- [ ] SPATIAL_CSW_ADMIN_USR
- [ ] SPATIAL_WFS_ADMIN_USR
- [ ] SYS

Output directory: C:\Users\IBM_ADMIN\IBM\workspace_ds4.1_dcw_32bits\Oracle_To

Figure 4-15 Data Movement wizard - selecting schemas

2. After you select the schemas, pick the tables you want to move, as shown in Figure 4-16. Click **Next**.

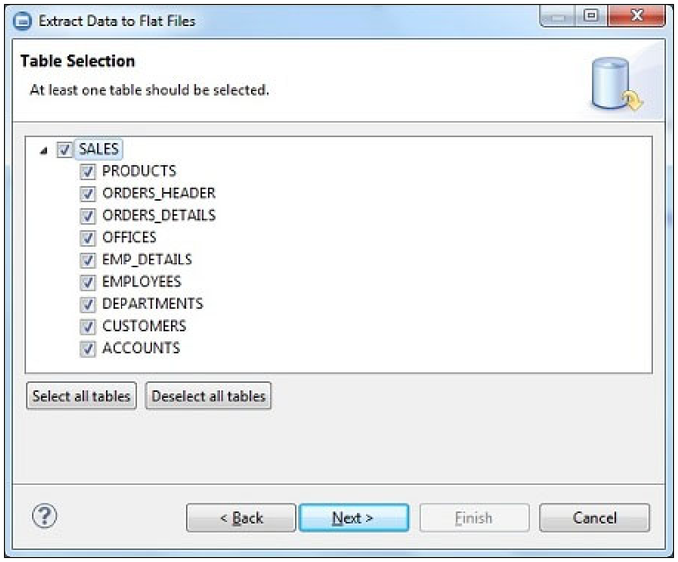


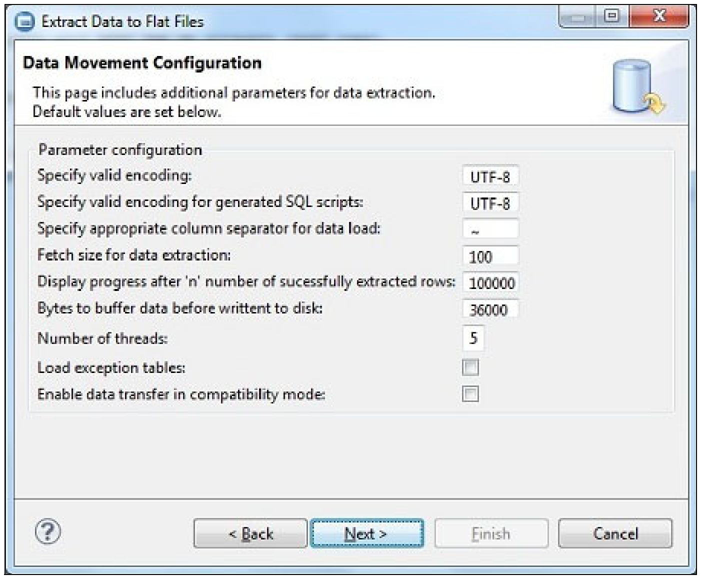
Table Selection

At least one table should be selected.

- [✓] SALES
  - [✓] PRODUCTS
  - [✓] ORDERS_HEADER
  - [✓] ORDERS_DETAILS
  - [✓] OFFICES
  - [✓] EMP_DETAILS
  - [✓] EMPLOYEES
  - [✓] DEPARTMENTS
  - [✓] CUSTOMERS
  - [✓] ACCOUNTS

Figure 4-16   Data Movement wizard - selecting tables

a. Specify the data movement configuration parameters (Figure 4-17). Click **Next**.


Extract Data to Flat Files

Data Movement Configuration

This page includes additional parameters for data extraction.
Default values are set below.

Parameter configuration

Specify valid encoding:	UTF-8
Specify valid encoding for generated SQL scripts:	UTF-8
Specify appropriate column separator for data load:	~
Fetch size for data extraction:	100
Display progress after 'n' number of sucessfully extracted rows:	100000
Bytes to buffer data before writtent to disk:	36000
Number of threads:	5
Load exception tables:	☐
Enable data transfer in compatibility mode:	☐

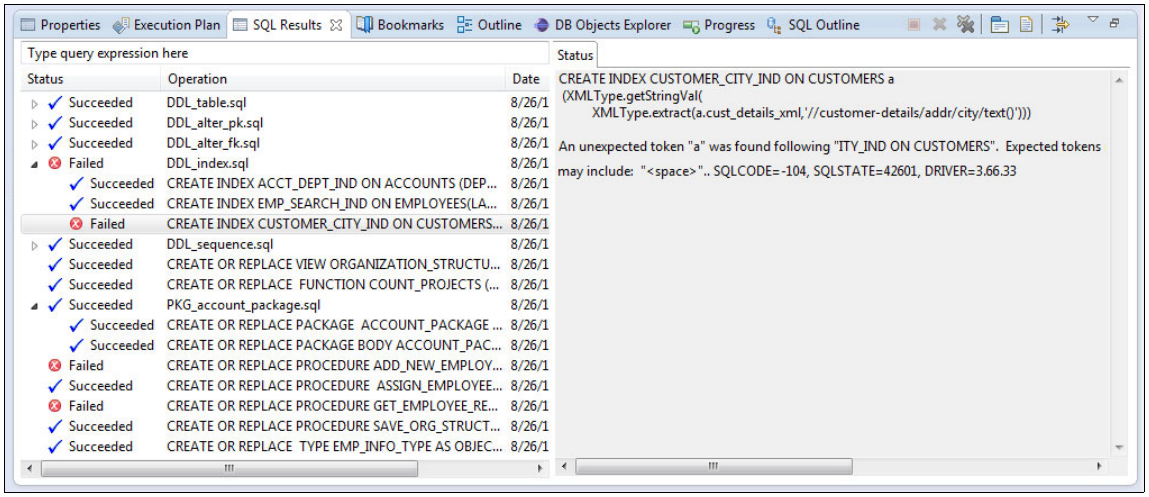
Figure 4-17 Data movement configuration parameters

After configuring the Data Movement parameters, you can view the summary of settings for data extraction and click **Finish**. The extraction starts and the progress is displayed in a separate window. The data movement log is automatically shown in Data Studio after the process is finished. These logs are also created in your DCW project under the Data Movement folder.

4.5.8 Deploying PL/SQL objects

Recall that the PL/SQL objects were extracted in 4.5.2, “Extracting DDL and PL/SQL objects” on page 223. After you load your data, you can use the interactive deployment feature of Data Studio to edit and deploy the PL/SQL routines to the target database.

You can use the same functionality that is used to deploy tables and other DDL scripts. Here you can select all the SQL scripts that you did not run yet, and the entire folders for PLSQL objects. Right-click the project and navigate to **Database Conversion** → **Run SQL Files...**, set the statement delimiter to "**@**", and select the connection profile of your target database. When the process completes, the results display in the SQL Results tab, as shown in Figure 4-18.



Status	Operation	Date
✓ Succeeded	DDL_table.sql	8/26/1
✓ Succeeded	DDL_alter_pk.sql	8/26/1
✓ Succeeded	DDL_alter_fk.sql	8/26/1
● Failed	DDL_index.sql	8/26/1
✓ Succeeded	CREATE INDEX ACCT_DEPT_IND ON ACCOUNTS (DEP...	8/26/1
✓ Succeeded	CREATE INDEX EMP_SEARCH_IND ON EMPLOYEES(LA...	8/26/1
● Failed	CREATE INDEX CUSTOMER_CITY_IND ON CUSTOMERS...	8/26/1
✓ Succeeded	DDL_sequence.sql	8/26/1
✓ Succeeded	CREATE OR REPLACE VIEW ORGANIZATION_STRUCTU...	8/26/1
✓ Succeeded	CREATE OR REPLACE FUNCTION COUNT_PROJECTS (...	8/26/1
✓ Succeeded	PKG_account_package.sql	8/26/1
✓ Succeeded	CREATE OR REPLACE PACKAGE ACCOUNT_PACKAGE ...	8/26/1
✓ Succeeded	CREATE OR REPLACE PACKAGE BODY ACCOUNT_PAC...	8/26/1
● Failed	CREATE OR REPLACE PROCEDURE ADD_NEW_EMPLOY...	8/26/1
✓ Succeeded	CREATE OR REPLACE PROCEDURE ASSIGN_EMPLOYEE...	8/26/1
● Failed	CREATE OR REPLACE PROCEDURE GET_EMPLOYEE_RE...	8/26/1
✓ Succeeded	CREATE OR REPLACE PROCEDURE SAVE_ORG_STRUCT...	8/26/1
✓ Succeeded	CREATE OR REPLACE TYPE EMP_INFO_TYPE AS OBJEC...	8/26/1

Status

```
CREATE INDEX CUSTOMER_CITY_IND ON CUSTOMERS a
 (XMLType.getStringVal(
       XMLType.extract(a.cust_details_xml,'//customer-details/addr/city/text()')))

An unexpected token "a" was found following "ITY_IND ON CUSTOMERS". Expected tokens
may include: "<space>".. SQLCODE=-104, SQLSTATE=42601, DRIVER=3.66.33
```

Figure 4-18 Viewing deployment results

Important: Some of the views and routines that you migrate might contain unqualified references to tables in the SALES schema. To avoid potential problems during deployment and at run time, connect to the target database as the *sales* user for deployment. This connection ensures that unqualified references are resolved to the correct schema.

4.5.9 Resolving incompatibilities with Interactive Deploy

This scenario intentionally introduces incompatibilities to illustrate methods for solving them. For example, the deployment results list that is shown in Figure 4-18 shows the statements that failed.

Deploy PL/SQL routines one by one, modifying statements when necessary. You must make the following changes:

- In the ADD_NEW_EMPLOYEE procedure, replace the reference to the `EMP_INFO_TYPE` constructor with type field assignments, as shown in Example 4-6. You can also define a PL/SQL function with the same signature as the constructor, which performs the field assignment.

Example 4-6 Correcting an unsupported object data type constructor

```
-- constructors not supported >> o_Employee:=EMP_INFO_TYPE(
--        x.emp_id, x.first_name, x.last_name, x.band);
       o_Employee.emp_id     := x.emp_id;
       o_Employee.first_name := x.first_name;
       o_Employee.last_name  := x.last_name;
       o_Employee.band       := x.band;
```

- In the EMPLOYEE_DYNAMIC_QUERY procedure, calls to the BIND_VARIABLE, COLUMN_VALUE, and DEFINE_COLUMN procedures of the built-in DBMS_SQL module must be modified to include the variable or column data type, as illustrated in Example 4-7.

Example 4-7 Modifying DBMS_SQL procedure signatures

```
-- modify signature >> DBMS_SQL.BIND_VARIABLE(
--       v_CursorID, ':d1', p_department1);
DBMS_SQL.BIND_VARIABLE_CHAR(v_CursorID, ':d1', p_department1);
```

- In the GET_EMPLOYEE_RESUME procedure, replace the call to **`DBMS_LOB.CREATETEMPORARY()`** with a call to the built-in function **`EMPTY_CLOB()`**. Also, replace the calls to **`DBMS_LOB.WRITE()`** and **`DBMS_LOB.APPEND()`** with **`DBMS_LOB.WRITE_CLOB()`** and **`DBMS_LOB.APPEND_CLOB()`**.
- In the INSERT_CUSTOMER_IN_XML procedure, replace references to the Oracle XMLTYPE and the related functions **`EXTRACT()`**, **`GETSTRINGVAL()`**, and **`EXISTSNODE()`** with the appropriate DB2 pureXML functions. Example 4-8 shows one of the original queries that retrieve data from the XML column, and Example 4-9 on page 240 presents the same query rewritten using the DB2 pureXML features.

Example 4-8 Querying Oracle XMLType column

```
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
```

Example 4-9 Querying DB2 XML column using pureXML

```
SELECT x.cust_name
FROM sales.CUSTOMERS c, XMLTABLE(
      '$CUST_DETAILS_XML//customer-details[addr/city=$city]'
      PASSING v_city as "city"
      COLUMNS
         cust_name VARCHAR(128) PATH 'name'
   ) x
Where cust_id<>v_cust_id
```

- Replace the Oracle **CREATE DIRECTORY** statements in the `DDL_other.sql` file with calls to the DB2 built-in procedure **UTL_DIR.CREATE_OR_REPLACE_DIRECTORY()**.

Important: The **CREATE DIRECTORY** statement references a directory name. Verify that the directory paths exist on the target server and that the DB2 instance owner (db2inst1) can write to these paths.

In addition to the manual changes described here, the converted PL/SQL code might include more modifications that the DCW performs automatically during conversion. For example, the nested function **AVERAGE_BAND()** in the procedure ACCOUNT_PACKAGE.DISPLAY_ACCOUNT_LIST is automatically converted into a package-level function, as DB2 10.5 does not yet support nested functions.

4.6 Verifying enablement

To verify that the database objects and PL/SQL code were converted successfully, run the PL/SQL anonymous block that simulates the application by calling its various functions. You can find the sample code for this block in Appendix E, “Code samples” on page 351.

Use the DB2 CLPPlus command-line interface (CLI), as shown in Example 4-10. You can use CLPPlus to run Oracle SQL*Plus scripts with minimal or no modifications.

*Example 4-10 Running an Oracle SQL*Plus script using DB2 CLPPlus*

```
db2inst1$ clpplus -nw sales/sales@localhost:50000/sales /
@Oracle_SalesDB_app_simulation.sql
CLPPlus: Version 1.5
Copyright (c) 2009, 2011, IBM CORPORATION.  All rights reserved.


Database Connection Information :
---------------------------------
Hostname = localhost
Database server = DB2/LINUXX8664  SQL10010
SQL authorization ID = sales
Local database alias = SALES
Port = 50000
```

The last command in the sample script `Oracle_SalesDB_app_simulation.sql` is **`SHOW ERRORS`**. It lists exceptions that might have occurred during the script execution. In this scenario, the script completes without errors and its output, which is printed by the calls to the built-in procedure **`DBMS_OUTPUT.PUT_LINE()`**, matches the output from the original application in the Oracle database, which confirms that the application enablement is successful.

4.7 Summary

This chapter demonstrated how to move a sample Oracle database application to DB2, complete with different data types and PL/SQL procedures and packages, with minimal effort. Almost all of the statements in the source database were supported by DB2 without modifications, and the few incompatible statements were quickly identified, modified, and deployed using IBM Database Conversion Workbench. The result of this short enablement process is a fully functional DB2 database.

5 Application conversion

This chapter provides examples of converting client-side applications from an Oracle environment to a DB2 environment. The applications use various languages, and each language can have its unique way of using a DB2 application interface (API). This chapter explains the necessary steps for converting client-side applications from the Oracle database to DB2.

This chapter covers the following topics:

- DB2 application development introduction
- Application enablement planning
- Converting XML features
- Converting Oracle Pro*C applications to DB2
- Converting Oracle Java applications to DB2
- Converting Oracle Call Interface applications
- Converting Open Database Connectivity applications
- Converting Perl applications
- Converting PHP applications
- Converting .NET applications

Examples in this chapter: The examples in this chapter are *excerpts* from actual programs and cannot be compiled and run by themselves.

5.1 DB2 application development introduction

When you develop applications that access databases, you embed the data access methods that are available in the programming language of choice into the application. IBM DB2 provides various programming interfaces for data access and manipulation.

There are various methods for performing database interaction from your application, including embedded static and dynamic SQL, and standard and DB2 native API calls.

5.1.1 Driver support

DB2 currently supports many standard database APIs and provides drivers for them, 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, 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 wants to use. DBD::DB2 is a Perl module, which, when used with DBI, allows Perl to communicate with the DB2 database.

Figure 5-1 illustrates the Perl/DB2 environment.

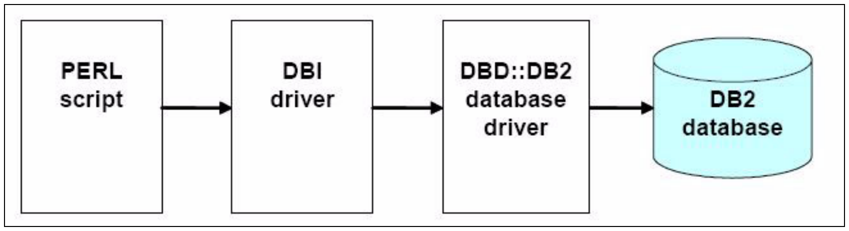


Figure 5-1 Perl/DB2 environment

You can download Perl source code or a precompiled binary distribution at:

http://www.perl.org/get.html

In addition to Perl, you need to download and install the following modules to use the Perl driver for DB2:

- DBI
- DBD::DB2

You can download these modules from the Comprehensive Perl Archive Network (CPAN) at:

http://www.CPAN.org

For the installation instructions, see the module documentation.

More information about Perl and DB2: For the latest information about Perl and DB2 and related Perl modules, including installation advice, see:

http://www.ibm.com/software/data/db2/perl/

PHP extensions

IBM supports access to DB2 databases from PHP applications through the following extensions, which offer distinct sets of features:

- ibm_db2

  This extension is an extension that is written, maintained, and supported feature by IBM for accessing DB2 databases and Cloudscape. It is an optimized driver that is built on top of the DB2 Call Level Interface (CLI) driver. It offers a procedural API that, in addition to the normal create, read, update, and delete database operations, offers extensive access to the database metadata. 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.

An easy method of installing and configuring PHP on Linux, UNIX, or Windows operating systems is to use Zend Server Community Edition, which provides a ready to use experience. You can download and install Zend Server for use in production systems from the following website:

http://www.zend.com/downloads

Additionally, precompiled binary versions of PHP are available for download at:

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 can be compiled.

To learn how to set up the PHP environment on Linux, UNIX, or Windows operating systems and develop applications, see *Developing Perl, PHP, Python, and Ruby on Rails Applications*, SC27-3876.

5.1.2 Embedded SQL

SQL statements can be embedded within a host programming language source, where the SQL statements provide the database interface while the host language provides the remaining functionality. Embedded SQL applications require a specific precompiler for each language environment to preprocess (or translate) the embedded SQL calls in to the appropriate API calls.

Building embedded SQL applications involves two prerequisite steps before application compilation and linking. An advantage of the static embedded SQL method is that it is often more efficient and can yield better performance.

To build DB2 embedded SQL applications, complete the following steps:

1. Prepare the source files that contain the embedded SQL statements using the DB2 precompiler.

   The **PREP** (**PRECOMPILE**) CLP command is used to start 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 modified source file. The precompiler produces access plans for the embedded SQL statements, which are stored together as a package in the database catalog.

2. Bind the package to the target database.

   Binding is done, by default, during precompilation (through the **PREP** command). If binding is deferred (for example, running the **BIND** command later), then the **BINDFILE** option needs to be specified with **PREP** for a bind file to be generated.

Figure 5-2 shows the precompile-compile-bind process for creating a program with embedded SQL. This process is different from building embedded SQL applications for Oracle database access, as the Oracle database does not have the concept of binding applications to the database before run time.

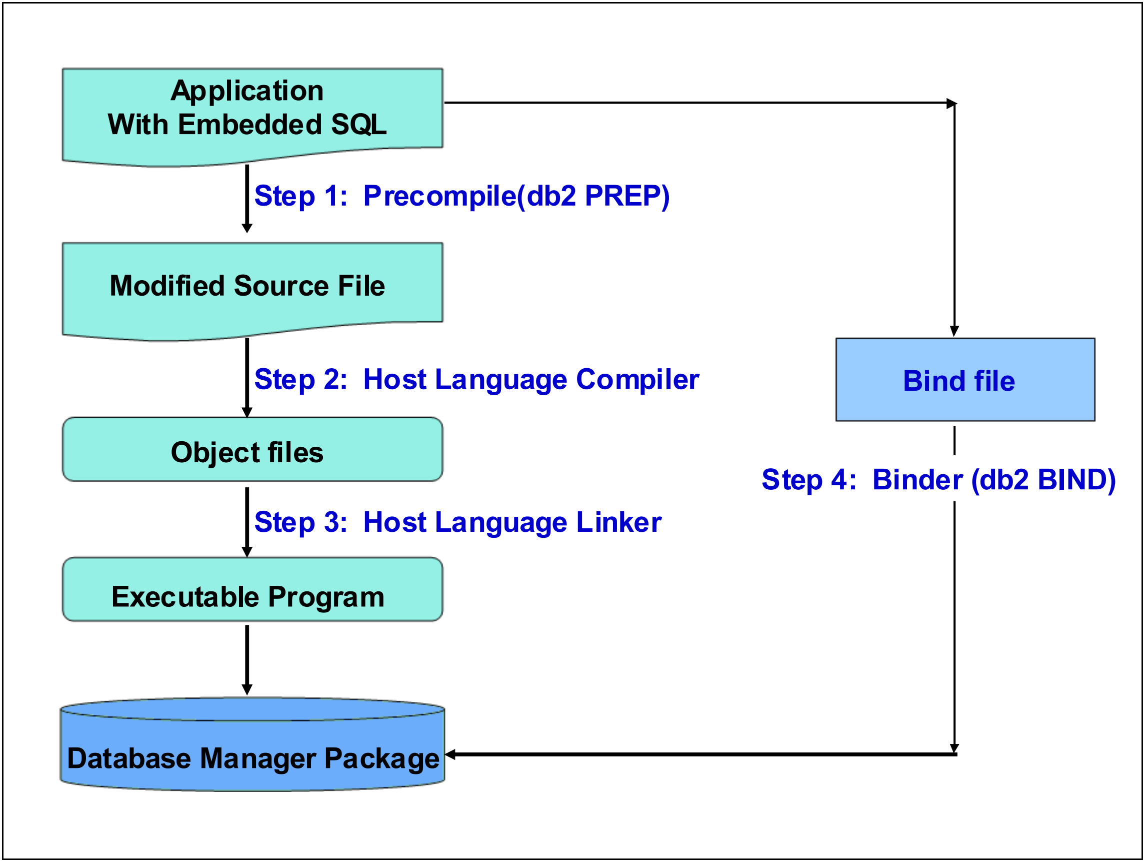

Application
With Embedded SQL
Step 1: Precompile(db2 PREP)
Modified Source File
Step 2: Host Language Compiler
Object files
Step 3: Host Language Linker
Executable Program
Database Manager Package
Bind file
Step 4: Binder (db2 BIND)

Figure 5-2 Process for creating DB2 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 this case, you are required to specify complete SQL statements before the program compilation. This situation means that all database objects (tables, columns, and so on) referenced in a statement, and the data types of all host variables, must be fully known at precompilation time. 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 without recompilation, 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 every database object in the SQL statement is not known at precompilation time, you can 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 defined and run later. Dynamic SQL statements are prepared and run during the program run time.

  In other words, dynamic SQL statements can be used 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 tabname FROM syscat.tables" );
strcat( st, " WHERE tabname <> ? 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;
```

The host variable **`PARM_VAR`** still must be declared in the **`EXEC SQL DECLARE SECTION`**.

5.2 Application enablement planning

Application enablement is another major step in the enablement project. This process includes the following steps:

- Checking software and hardware availability and compatibility
- Educating developers and administrators
- Analyzing application logic and source code
- Setting up the target environment
- Changing vendor-specific database API use
- Application testing
- Application tuning
- Production rollout procedures
- User education

Planning includes the creation of a project plan. Plan enough time and resources for each task. IBM and IBM Business Partners can help you define a well-prepared project.

When migrating applications developed in-house, the entire enablement effort falls in to the hands of the development team. With packaged applications, you can contact the vendor for a recommended enablement process.

5.2.1 Checking software and hardware availability and compatibility

The application architecture profile is one of the important outputs at this stage of the enablement process. While you prepare the architecture profile, you must assess the availability and compatibility of all software and hardware components in the new environment.

5.2.2 Educating developers and administrators

Ensure that your staff has the appropriate, up-to-date skills for all products and the system environment you use for the enablement project. Understanding the capabilities, limitations, and the application development approaches of the new product is essential for fully analyzing the source system.

5.2.3 Analyzing application logic and source code

In this phase, you should identify all the affected sources. With the availability of DB2 10.5, the new Oracle Database compatibility features have closed most of the gaps. Minor incompatibilities might still exist in the use of the Oracle Data Dictionary, optimizer hints, some built-in functions and packages, and certain SQL constructs. You also must analyze the use by the applications of the Oracle proprietary APIs and Oracle extensions to the standard interfaces, such as JDBC.

5.2.4 Setting up the target environment

The target system, either the same or a different one, must be set up for the DB2 application development. The environment can include:

- An Integrated Development Environment (IDE)
- Database client software and the required drivers
- Source code repository
- Additional development and build tools
- Configuration management tools
- Documentation tools
- Automated testing tools

A complex system environment usually consists of products from different vendors. Check their availability and compatibility before you start the project.

5.2.5 Changing vendor-specific database API use

You might need to change the database API calls in the applications if they are proprietary or present vendor-specific extensions to the standard interfaces.

The following changes might be needed:

- Language syntax changes

  The syntax of database calls varies in different programming languages. We describe various C/C++, Java, and other applications. For information about other languages, contact IBM Technical Sales.

- SQL query changes

  You can use nonstandard SQL syntax in Oracle, such as the use of unnamed columns in subqueries. Many Oracle database DDL statements include features that are not applicable to DB2. Although DB2 now supports or tolerates many of these SQL extensions, in some cases you must modify incompatible statements. For supported SQL syntax and features, see:

  http://pic.dhe.ibm.com/infocenter/db2luw/v10r5/index.jsp

  SQL Reference, Volume 1, SC27-3872 and *SQL Reference, Volume 2*, SC27-3872 also contain useful information.

  In some cases, you might have to modify the SQL queries to the Oracle Data Dictionary. DB2 supports only a subset of the Oracle Data Dictionary views, as the database metadata differs between the Oracle Database and DB2.

- Changes in calling procedures and functions

  Sometimes you must change procedures to functions (and vice versa) or modify the routine signatures. In such cases, you must change all of the affected routine calls, and the parameter passing and processing of the routine results in PL/SQL code within the database and in the client applications.

- Logical changes

  In rare cases, changes in the program logic might be required because of architectural differences between the Oracle Database and DB2. For example, you might need to carefully analyze the application behavior if it relies on certain features of the Oracle database concurrency model.

5.2.6 Application testing

A complete application functional test is necessary after the database conversion and application enablement to ensure that all data and functionality is migrated without errors.

It is prudent to perform the enablement process several times in a development environment to verify the process, later perform the same enablement on a test system with existing test data, and then on a copy or subset of production data, before eventually deploying the enabled application into production.

5.2.7 Application tuning

Tuning is a continuous activity for any database because data volume, the number of users, and applications change from time to time. After the successful enablement end-to-end application, tuning should be performed with particular focus on the architectural differences between Oracle and DB2. There are many publications that offer DB2 performance tuning advice and preferred practices, for example, *Tuning and Monitoring Database System Performance*, at:

http://www.ibm.com/developerworks/db2/bestpractices/systemperformance/

Alternatively, you can use *Troubleshooting and Tuning Database Performance*, SC27-3889.

5.2.8 Production rollout procedures

The rollout procedures vary and depend on the type of application and the database connection that you have:

- Prepare client workstations and application servers by installing correct DB2 Data Server Client editions or individual drivers, appropriate for the DB2 Data Server version.
- Plan and implement database maintenance procedures, monitoring infrastructure, high availability, and disaster recovery features using the available DB2 features and add-on products according to your requirements.

5.2.9 User education

If there are changes in the user interface, business logic, or the application behavior because of system improvements, user education is required. Be sure to provide correct user education, because acceptance of the new environment is largely tied to the skills and satisfaction of the users.

5.3 Converting XML features

Although SQL/XML and XQuery are each defined by their own particular standards, there are still differences in how Oracle Database and DB2 adhere to these standards. As a result, there are differences in how you create, store, query, and modify XML content. DB2 has supported XML natively since DB2 9.1 in the form of its DB2 pureXML feature, and has continually enhanced the XML support to include non-Unicode database, XML Load utility, and JDBC support.

5.3.1 SQL/XML

SQL/XML is an extension of SQL that is part of the ISO SQL specification. The SQL/XML functions start XPath or XQuery expressions and are used in SQL statements to access XML data or to generate (publish) XML documents from relational data.

SQL/XML functions can be categorized into two groups:

- Functions that query and access XML content
- Functions that generate (publish) XML content from SQL data

For those SQL/XML functions that query and access XML content, Oracle Database 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 methods belong to the XMLType data type. Some examples are **getStringVal()**, **getClobVal()**, **getNumberVal()**, **getNamespace()**, and **getBlobVal()**. In addition to the XMLType methods, Oracle Database also provides other SQL/XML functions, such as **EXTRACT()**, **EXISTSNODE()**, and **EXTRACTVALUE()**.

Oracle Database also supports the ISO/IEC standard SQL/XML functions, **XMLQuery**, **XMLTable**, and **XMLExists**. These functions, known as the XQuery functions, are also supported on DB2, with **XMLExists**.

In addition to the SQL/XML querying functions, both DB2 and Oracle databases support several other types of functions, such as **XMLCast**, **XMLParse**, and **XMLSerialize**. Oracle Database also supports the casting function **XMLType** that converts an XML string value to an **XMLType** value. **XMLType** is most similar to the DB2 **XMLParse** function. To cast XML values to scalar string values, you can use the DB2 **XMLCast** function.

DB2 supports standard functions that generate XML documents and fragments from relational data. These functions are referred to as the "publishing" functions, and include **XMLDocument**, **XMLElement**, **XMLAgg**, **XMLAttribute**, **XMLConcat**, and **XMLForest**.

In short, you can use the DB2 pureXML feature to map the Oracle database XMLType data type and the corresponding methods and functions to the DB2 XML data type and DB2 pureXML functions.

Table 5-1 lists some of the frequently used SQL/XML functions that are supported by Oracle Database and maps them to DB2 equivalents.

Table 5-1 SQL/XML function mapping

Oracle Database SQL/XML	SQL/XML category	Oracle specific	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 an SQL scalar value.
`getstringVal`	Access	Yes	`XMLCAST`	Converts an XML value to an SQL scalar value.
`getNumberVal`	Access	Yes	`XMLCAST`	Converts an XML value to an SQL 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.
`XMLFOREST`	Generate	No	`XMLFOREST`	Returns a sequence of element nodes.

To convert Oracle SQL/XML functions to DB2, find the best equivalent DB2 pureXML functions and see *pureXML Guide*, SC27-3892-00 and *DB2 SQL Reference, Volume 1*, SC27-3872-00 for correct syntax and function usage guidelines. There might be some syntax differences even if a particular function is a part of the ISO/IEC or W3C standard and is supported both by Oracle Database and DB2.

Example 5-3 and Example 5-4 demonstrate some of these differences. In Example 5-3, both **extract()** and **existsNode()** functions are Oracle SQL/XML functions; the XML namespace is specified.

Example 5-3 Querying XML data with SQL/XML functions in an Oracle database

```
SELECT
extract(info,'/customerinfo//addr','xmlns="http://posample.org"')
FROM customer
WHERE
   existsnode(info,'/customerinfo//addr[city="Aurora"]',
              'xmlns="http://posample.org"')=1;
```

Example 5-4, the DB2 example, uses the wildcard notation (*:) for namespace, which is prefixed to the XML elements. The wildcard matches any name space. Although the wildcard notation is part of the W3C standard, it is not supported by Oracle.

Example 5-4 Querying XML data with pureXML functions in DB2

```
SELECT XMLQUERY('$R/*:customerinfo//*:addr' PASSING info AS "R")
FROM customer
WHERE XMLEXISTS('$R/*:customerinfo//*:addr[*:city="Aurora"]'
                 PASSING info as "R");
```

Example 5-5 shows how to specify a name space in DB2 XMLQuery when not using the wildcard notation.

Example 5-5 Specifying XML namespace in XMLQUERY

```
SELECT XMLQUERY ('declare default element namespace
"http://posample.org"; $R/customerinfo//addr' PASSING INFO AS "R")
```

Example 5-6 and Example 5-7 demonstrate the casting of XML values in an Oracle database and DB2.

- Oracle database

Example 5-6 Extracting XML sequences and scalar values using Oracle functions

```
cityxml := incust.extract('/customerinfo//city');
city := cityxml.extract('//text()').getstringval();
```

- DB2

Example 5-7 Extracting XML sequences and scalar values using the DB2 pureXML feature

```
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 Database runs the expression based on the type of XMLType storage used. If XMLtype uses unstructured storage, when it is represented by a CLOB instance, then the value is parsed and a DOM tree of the XML document is created in memory to process the XPath expression. If XMLtype uses structured storage, which translates into a set of relational tables, the Oracle database rewrites the XPath expression into the equivalent SQL statement query.

In DB2, values of the XML data type are stored in a special hierarchical binary format, and the DB2 pureXML query engine can traverse XML documents using XPath expressions. As a result, the XPath or XQuery expressions are not rewritten into SQL statements and XML documents do not need to be loaded into memory as DOM trees for processing.

5.3.2 XQuery

Although standardization was a goal of the World Wide Web Consortium (W3C) when it designed the XQuery language, there are differences in how Oracle and IBM implemented XQuery within their 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 (CLP). An XQuery statement is prefixed with the keyword **`XQUERY`** and is not limited to being started only from an SQL/XML function. The keyword indicates that the primary language is XQuery.

Two DB2 defined functions are used in In XQuery to access relational data in a DB2 database:

- db2-fn:xmlcolumn: For retrieving an XML sequence from an XML column.
- db2-fn:sqlquery: For retrieving a sequence of XML values based on the result of an SQL query.

In an Oracle database, the XQuery statement cannot be embedded directly within SQL applications. The XQuery language statements in applications are run with the functions **XMLQuery()** and **XMLtable()**. The XQuery command can be executed natively only from the SQL*Plus command processor environment. However, before you run XQuery from SQL*Plus, the environment must be properly initialized, which is accomplished by running an Oracle-provided script. After you run 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 functions are ora:view, ora:contains, and ora:replace. These extension functions do not have equivalents in DB2. To convert the Oracle XQuery extension functions to DB2, you must rewrite the XQuery expressions that contain them. For example, ora:view is used to create XML views on relational data so that the data can be manipulated as an XML document. In DB2, this task is accomplished by using the SQL/XML publishing functions that were described earlier.

Oracle Database 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 in the Oracle XML DB repository. In DB2, XML documents are always stored in tables as column values, and the db2-fn:xmlcolumn function can be used instead.

Example 5-8 compares the differences between the **XMLQuery** function that is used in the Oracle and DB2 databases.

Example 5-8 XQuery differences

```
-- In Oracle database ------------------------------------------------
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 5-9 shows how to loop through XML content in an Oracle PL/SQL application.

Example 5-9 Looping through XML content in Oracle PL/SQL application

```
CURSOR cur1(vcity IN VARCHAR2) IS
SELECT info from customer_us
WHERE
existsnode(info,'/customerinfo//addr[city="'||vcity||'"]','xmlns="http:
//posample.org"') = 1;
...
FOR c IN cur1(city) LOOP
customer := c.info.extract('//name','xmlns="http://posample.org"');
...
END LOOP;
```

In a DB2 application, the same iteration through XML content can be done, except that the FOR loop is replaced, because DB2 does not currently allow XML variables in the FOR block. See Example 5-10.

Example 5-10 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//name' passing customer as
"cust");
END LOOP;
```

Example 5-11 and Example 5-12 on page 259 compare the use of the **XMLTABLE** function in the Oracle and DB2 databases. The **XMLTABLE** function is a standard SQL/XML function, and the same name space declaration can be used in both cases.

Example 5-11 Using the XMLTABLE function in Oracle database

```
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 5-12 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 Modifying XML data

The initial release of the XQuery language only specified querying of XML and did not provide for updating or deleting individual XML elements or sequences. IBM implemented preliminary XQuery Update specifications since DB2 9.5. Oracle Database 11g has also made functions available that allow modification of XML documents, such as **`updateXML`**, **`insertXML`**, and **`deleteXML`** that are not part of the ISO/IEC or W3C standard.

Consider the XML document that is shown in Example 5-13; the two examples that follow (Example 5-14 on page 260and Example 5-15 on page 260) illustrate the differences between Oracle Database and DB2 when you modify XML documents.

Example 5-13 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 that you want to update the address of the student and remove his phone number. In an Oracle database application, these changes can be carried out as shown in Example 5-14.

Example 5-14 Delete/update XML elements in a document in an Oracle database

```
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, the same changes are performed by using the standard XQuery syntax, as shown in Example 5-15.

Example 5-15 Delete/update XML elements in a document in a DB2 database

```
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 ...
```

For a detailed description of DB2 pureXML features, see DB2 documentation, particularly *pureXML Guide*, SC27-3892.

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, and an enriched set of built-in functions), support for embedding SQL into other host languages (C/C++) has not changed in a practical sense.

The Oracle embedded SQL solution is Oracle Pro*C, which supports C/C++ and COBOL. It uses a precompiler to process SQL statements that are embedded in the program source code.

This section explains the general procedure of converting Oracle Pro*C code to the DB2 embedded SQL programs.

5.4.1 Connecting to the database

There is a difference in how C programs connect to the Oracle and DB2 databases. Each Oracle instance can manage only one database. DB2 instances can manage multiple databases, so the database name should be explicitly provided by a **CONNECT** statement.

To connect to the Oracle database, you must specify the Oracle user and password, while the Oracle instance (service name) is specified in the application environment:

```
EXEC SQL CONNECT :user_name IDENTIFIED BY :password;
```

In DB2, you must specify the database name, user ID, and password. The **CONNECT** statement looks like the following command:

```
EXEC SQL CONNECT TO :dbname USERID :user_name PASSWORD :password;
```

The `dbname`, `user_name`, and `password` must be declared as host variables.

Starting with DB2 10, the Oracle **CONNECT** statement syntax is also accepted when the compatibility mode is enabled.

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 it uses 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 host programming language compiler.

As the C program manipulates the values from the tables using host variables, the first step is to ensure compatibility of DB2 and C data types.

All host variables in a C program must be declared in a special declaration section, so that the DB2 precompiler can identify the host variables and their 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 might be different from Oracle precompiler rules. Oracle precompiler permits host variables to be declared as `VARCHAR`. `VARCHAR[n]` is a pseudo-type that is 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 10 supports this simple `VARCHAR` type declaration as well. For example:

```
EXEC SQL BEGIN DECLARE SECTION;
VARCHAR emp_name [n+1];
EXEC SQL END DECLARE SECTION;
```

The use of a null-terminated char array (`char emp_name[n+1]`) 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` is 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 that are 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 must take it out of E**XEC SQL DECLARE SECTION** and define the host variable `myUser` as a structure.

You can use DB2 to declare a host variable as a pointer. However, if a host variable is declared as a pointer, no other host variable can be declared with that same name within the same source file.

The host variable declaration `char *ptr` is accepted but should not point to a null-terminated character string of an undetermined length. It is assumed that the pointer is to a fixed-length, single-character host variable. This value might not be what was intended for the Oracle host variable declaration.

Use **sqlint32** and **sqlint64** for **INTEGER** and **BIGINT** host variables. By default, the usage 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, which you use if you need to deal with a 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 follows:

```
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;
```

For more information about handling XML types within C applications, see *DB2 Express-C: The Developer Handbook for XML, PHP, C/C++, Java, and .NET*, SG24-7301 and *Developing Embedded SQL Applications*, SC27-3874.

Example 5-16 shows a fragment of PRO*C code that demonstrates array host variables. The last statement places all 10 rows from the cursor into arrays.

*Example 5-16 Array host variables in Oracle Pro*C*

```
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;
```

DB2 10 and higher accept bulk fetch instructions when the compatibility mode is set. Example 5-17 shows the conversion of the arrays host variable declaration for DB2 9.x and lower.

Example 5-17 Array host variable conversion in DB2 before Version 10

```
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_numb[i] = h_dept_numb;
    strcpy(dept_name[i], h_dept_name);
}
```

5.4.3 Exception handling

The mechanisms for handling exceptions are similar in the Oracle and DB2 embedded SQL applications, 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 (SQL Communication Area) after each SQL statement, and to perform the action that is 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 until the next suitable **WHENEVER** statement is found or the end of file is reached. No functions or programming blocks are considered in that analysis. For example, you might have two different **SELECT** statements. One statement must return at least one row, and the other statement might not return any rows. You 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 = :pnone_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;
 ……
```

Convert the **DO** and **STOP** Oracle keywords in a **WHENEVER** statement 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 an Oracle database. SQLCA provides information for diagnostics and event handling.

To get the full text of longer or nested error messages, you must use the **sqlglm()** function:

```
sqlglm(message_buffer, &buffer_size, &message_length);
```

message_buffer is the character buffer in which you want the Oracle driver to store the error message, buffer_size specifies the size of message_buffer in bytes, and the actual length of the error message is placed in *message_length. The maximum length of an Oracle error message is 512 bytes.

DB2 provides a special runtime API function to return an error message that is based on **SQLCODE**:

```
rc=sqlaintp(msg_buffer, 1024, 80, sqlca.sqlcode);
```

80 stands for the number of characters after which a line break is inserted in the message. DB2 searches 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 starting this function, the allocated buffer contains 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 that is associated with the specific **SQLSTATE**:

```
rc=sqlogstt(msg_sqlstate_buffer, 1024, 80, sqlca.sqlcode);
```

As a result of starting this function, `char msg_sqlstate_buffer[1024]` contains, 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

To start a remote database procedure, which can be part of a PL/SQL package, in an Oracle embedded SQL application, use the following statements:

```
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 can be achieved through arguments. You can choose one of the following modes for each argument:

- **IN**
- **OUT**
- **INOUT**

For example, declare the previous stored procedure as follows:

```
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 start, values that are passed from the calling program are accepted by the stored procedure.

A DB2 client application start a stored procedure by using the **CALL** statement, which can pass parameters to the stored procedure and receive parameters that are returned from the stored procedure. Using the previous example, it uses the following syntax:

```
EXEC SQL CALL package_name.SP_name (:arg_in1, :arg_in2, :status_out);
```

As with all SQL statements, you can also prepare the CALL statement with parameter markers and then supply values for the markers using **SQLDA**:

```
EXEC SQL CALL package_name.SP_name USING DESCRIPTOR :*psqlda;
```

You must set up the SQL Data Area (**SQLDA**) before use. **SQLDA** can be helpful if you have an unknown number of host variables or many variables.

To start 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 for each **IN** and **INOUT** parameter of the stored procedure should be declared and initialized.

Consider Example 5-18, in which the program is written to give a raise to each employee whose current salary is less than certain value. The program passes that value to a stored procedure, performs an update, and returns the status. The client code in C looks as shown in Example 5-18.

Example 5-18 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;
}
```

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 scripts are 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 an *.sqc file.

To help you compile and link the source files, build files are provided 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.

The utilemb.sqc and utilemb.h files contain functions for error handling. Compile and link these utility functions along with the target application sources. Both the makefile and build files in the sample directories perform this task for the programs that require error-checking utilities.

For more information about building C applications, see *Developing Embedded SQL Applications*, SC27-3874.

5.5 Converting Oracle Java applications to DB2

For Java programmers, DB2 offers the following programming interfaces:

- 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 distributed as a package called *IBM Data Server Driver for JDBC and SQLJ (JCC Driver)*.
- SQLJ is a standard development model for embedding SQL statements in 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 be used by 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 straightforward. 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 components, such as user name and database name, are then configured declaratively within the application.

However, you must change your Java source code with regard to:

- The API driver name (JDBC or SQLJ)
- The database connect string (JDBC URL)
- Any incompatible SQL statements
- Vendor-specific extensions to the JDBC API used in the application

DB2 provides a different method for optimizer directives, but tolerates and ignores Oracle optimizer hints that appear in SQL statements. 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 about the Java environment, drivers, programming, and other relevant information, see *Developing Java Applications*, SC27-3875-00.

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 in to 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 IBM WebSphere® Application Server)
  - JavaServer Pages (JSPs) (JDBC)
  - Servlets (SQLJ or JDBC)
  - Enterprise JavaBeans (EJBs) (SQLJ or JDBC)

5.5.2 JDBC driver for DB2

The IBM Data Server Driver for JDBC and SQLJ supports:

- All of the methods that are described in the JDBC 3.0 and JDBC 4.0 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 implement 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 later.
- 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.

Type 4 connectivity

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 5-19.

Example 5-19 getConnection syntax for Type 4 connectivity

```
getConnection(String url, user, password);
getConnection(String url, java.util.Properties info);
```

The following is the syntax for a URL for IBM DB2 Driver for JDBC and SQLJ Type 4 connectivity.

```
>>-+-jdbc:db2:------+-//server--+-------+--/database------------>
   '-jdbc:db2j:net:-'           '-:port-'

>--+----------------------------+-----------------------------><
   |   .----------------------. |
   |   V                      | |
   '-:---property--=--value--;-+-'
```

Example 5-20 shows how to set the user ID and password in the **user** and **password** parameters.

Example 5-20 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);
```

5.5.3 JDBC driver declaration

Oracle provides a JDBC OCI driver, among many other JDBC drivers, to enable a Java application to use OCI to access the Oracle database through SQL*NET. In this case, the C layer talks to the Oracle database and then returns to the Java layer. To connect from a Java application to an Oracle database using the OCI driver, complete the following steps:

1. Import the Oracle driver.
2. Register the driver manager.
3. Connect with a user ID, the password, and the database name.

Example 5-21 shows an Oracle JDBC connection through OCI.

Example 5-21 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");

    // ...
  }
}
```

DB2 does not support the JDBC OCI driver. You do not need to run a Java application using OCI, even though DB2 provides OCI support since DB2 9.7 Fix Pack 1. Previous versions of DB2 JDBC drivers used DB2 Call Level Interface to communicate with the database manager, but the newer DB2 JCC Driver is rewritten to eliminate the DB2 Call Level Interface layer. It is a pure Java implementation of the IBM DRDA® communication protocol.

If an Oracle application is using the Oracle JDBC OCI driver, you can change it directly to use the JCC driver. It is not necessary to import a JDBC library when you connect to DB2. The registration and connection to DB2 is demonstrated in Example 5-22. The parameters for the getConnection method are determined by the connection type.

Example 5-22   DB2 JDBC connection

```
import java.sql.*;

class rsetClient
{
   public static void main (String args []) throws SQLException  {

   // Load DB2 JDBC application driver
   try
   {
   // IBM Data Server Driver for JDBC and SQLJ
      Class.forName("com.ibm.db2.jcc.DB2Driver");
   }
   catch (Exception e)
   {
      e.printStackTrace();
   }
```

```
    // Connect to the database
    Connection conn =
    DriverManager.getConnection("jdbc:db2://hostname:50000/dbname"
       ,"uid","pwd");
     // ...
  }
}
```

5.5.4 New binary XML API

Starting with DB2 10, the communication protocol allows transferring XML data in binary (XDBX) format, which allows applications that are aware of the new JDBC API to increase performance of the XML data type manipulation

For JDBC and SQLJ applications, you can now choose to transmit data to and from a DB2 10.5 server in binary XML format. For applications that work with data in a non-textual representation, such as those using SAX or StAX objects, the binary format provides a faster way to transmit and receive XML data. In earlier releases, only character-based representation of XML data was supported. Now you can use whichever format best suits your data processing needs. Binary XML format is only used for data transmission. XML data continue to be stored in the database as before in the pre-parsed hierarchical form.

For JDBC and SQLJ applications that work with data in a non-textual representation, binary XML format eliminates unnecessary XML parsing and serialization costs, therefore improving performance. For example, you should see performance improvements if your application uses any of the following methods to retrieve and update XML data:

- `getSource(SAXSource.class)`, `getSource(StAXSource.class)`
- `setResults(SAXResults.class)`, `setResults(StAXResult.class)`

The degree of performance improvement also depends on the structure of the XML documents, the length of tags, the number of repeating tags, and the depth of data within the document.

To benefit from the new binary XML format, you must use Version 4.9, or later, of the IBM Data Server Driver for JDBC and SQLJ and connect to DB2 10.1, or later, server. For SQLJ applications, you also must use Version 4.9, or later, of the `sqlj4.zip` package.

For JDBC and SQLJ applications that use Version 4.9, or later, of the IBM Data Server Driver for JDBC and SQLJ, binary XML is the default format when the application connects to a DB2 10.1 or later release server. You can use the `xmlFormat` property in the DriverManager and DataSource interfaces to control whether the transmission of XML data is in textual or binary format.

You can use binary XML format with any valid SQL/XML or XQuery statements, as shown in Example 5-23.

Example 5-23 Enable XML binary usage with JDBC driver

```
...
properties.put("xmlFormat", DB2BaseDataSource.XML_FORMAT_BINARY);
DriverManager.getConnection(url, properties);
...
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT XMLCOL FROM XMLTABLE");
ContentHandler handler = new MyContentHandler();
while (rs.next()) {
   SQLXML sqlxml = rs.getSQLXML(1);
   SAXSource source = sqlxml.getSource(SAXSource.class);
   XMLReader reader = source.getXMLReader();
   reader.setContentHandler(handler);
   reader.parse(source.getInputSource());
}
...
```

For more information, go to:

http://publib.boulder.ibm.com/infocenter/db2luw/v10r5/index.jsp?topic=%2Fcom.ibm.db2.luw.xml.doc%2Fdoc%2Fc0056290.html

5.5.5 Stored procedure calls

The handling of input and output parameters in stored procedures calls differ between the Oracle and DB2 databases. 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 was created in an Oracle database as follows:

```
CREATE OR REPLACE  PROCEDURE sproc1(
in_parm1 IN INTEGER, out_parm2 OUT VARCHAR2)
```

The same procedure appears in a DB2 database.

The procedure has one input parameter and one output parameter. There is no difference in the call between the two database platforms. In both cases, the parameter values need to be set *before* the stored procedure can be run. Example 5-24 demonstrates this point.

Example 5-24 Calling a stored procedure with input and output parameters from Java

```
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 that does not have an input parameter, but defines a result set as an output parameter. The result set is CURSOR defined and opened 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 5-25.

Example 5-25 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 the output parameter as a similar DB2Type. See Example 5-26.

Example 5-26 Java call of DB2procedure 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 output parameter of CURSOR type, then you do not need to register the result set with the **registerOutParameter()** method in the Java application. To get the result set, call the **getResultSet()** method instead of **getObject()**, as demonstrated in Example 5-27.

Example 5-27 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 described in “Stored procedure with a result set” on page 276. 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 an Oracle application, you can retrieve the cursor ResultSet in Java with a special syntax, for example, SP_CALL (shown in Example 5-28).

Example 5-28 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 that returns a cursor directly with the DB2 JDBC driver, convert the Oracle function to a stored procedure in DB2. To avoid changes to the function source code, you can also wrap the existing function into a procedure that returns the cursor as an output parameter. Example 5-29 illustrates a function wrapper.

Example 5-29 Function wrapper

```
CREATE OR REPLACE PROCEDURE myfunction_wrapper(C OUT sys_refcursor
,arg1, ...argn ) IS
   BEGIN
      c:= myfuntion(arg1, ...argn  );
   END;
/

CREATE OR REPLACE FUNCTION myfunction(arg1, ...argn) IS
   BEGIN
      ...UDF logic here...
   ENd;
/
```

5.6 Converting Oracle Call Interface applications

In DB2 9.7 and later, the DB2 Call Interface (DB2CI) provides compatibility for the Oracle Call Interface (OCI). OCI is one of the many programming interfaces that are used by C/C++ developers to create applications for Oracle databases. 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.

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, and DATE) functions, date and time, large object processing, arrays, stored procedure executions, and file I/O. These supported functions have a syntax that is compatible with the Oracle OCI functions.

The DB2CI driver is evolving and provides the OCI functions listed in Table 5-2 in the current release of the DB2 OCI driver (DB2 10.5). The list can be expanded but is accurate at the time of the writing of this book.

Table 5-2 DB2 OCI-compatible functions

OCIAttrGet	OCILobGetLength	OCINumberTan
OCIAttrSet	OCILobIsEqual	OCINumberToInt
OCIBindArrayOfStruct	OCILobIsTemporary	OCINumberToReal
OCIBindByName	OCILobIsOpen	OCINumberToRealArray
OCIBindByPos	OCILobLocatorAssign	OCINumberToText
OCIBindDynamic	OCILobLocatorIsInit	OCINumberTrunc
OCIBreak	OCILobRead	OCIParamGet
OCIClientVersion	OCILobTrim	OCIParamSet
OCIDateAddDays	OCILobWrite	OCIPasswordChange
OCIDateAddMonths	OCILogoff	OCIPing
OCIDateAssign	OCILogon	OCIRawAllocSize
OCIDateCheck	OCILogon2	OCIRawAssignBytes
OCIDateCompare	OCINumberAbs	OCIRawAssignRaw
OCIDateDaysBetween	OCINumberAdd	OCIRawPtr
OCIDateFromText	OCINumberArcCos	OCIRawResize
OCIDateLastDay	OCINumberArcSin	OCIRawSize
OCIDateNextDay	OCINumberArcTan	OCIReset
OCIDateSysDate	OCINumberArcTan2	OCIResultSetToStmt
OCIDateToText	OCINumberAssign	OCIServerAttach
OCIDefineArrayOfStruct	OCINumberCeil	OCIServerDetach

OCIDefineByPos	OCINumberCmp	OCIServerVersion
OCIDefineDynamic	OCINumberCos	OCISessionBegin
OCIDescribeAny	OCINumberDec	OCISessionEnd
OCIDescriptorAlloc	OCINumberDiv	OCISessionGet
OCIDescriptorFree	OCINumberExp	OCISessionRelease
OCIEnvCreate	OCINumberFloor	OCIStmtExecute
OCIEnvInit	OCINumberFromInt	OCIStmtFetch
OCIErrorGet	OCINumberFromReal	OCIStmtFetch2
OCIFileClose	OCINumberFromText	OCIStmtGetBindInfo
OCIFileExists	OCINumberHypCos	OCIStmtGetPieceInfo
OCIFileFlush	OCINumberHypSin	OCIStmtPrepare
OCIFileGetLength	OCINumberHypTan	OCIStmtPrepare2
OCIFileInit	OCINumberInc	OCIStmtRelease
OCIFileOpen	OCINumberIntPower	OCIStmtSetPieceInfo
OCIFileRead	OCINumberIsInt	OCIStringAllocSize
OCIFileSeek	OCINumberIsZero	OCIStringAssign
OCIFileTerm	OCINumberLn	OCIStringAssignText
OCIFileWrite	OCINumberLog	OCIStringPtr
OCIHandleAlloc	OCINumberMod	OCIStringResize
OCIHandleFree	OCINumberMul	OCIStringSize
OCIInitialize	OCINumberNeg	OCITerminate
OCILobAppend	OCINumberPower	OCITransCommit
OCILobAssign	OCINumberPrec	OCITransDetach
OCILobClose	OCINumberRound	OCITransForget
OCILobCopy	OCINumberSetPi	OCITransMultiPrepare
OCILobCreateTemporary	OCINumberSetZero	OCITransPrepare
OCILobDisableBuffering	OCINumberShift	OCITransRollback
OCILobEnableBuffering	OCINumberSign	OCITransStart

`OCILobErase`	`OCINumberSin`	`xaoEnv`
`OCILobFreeTemporary`	`OCINumberSqrt`	`xaosterr`
`OCILobFlushBuffer`	`OCINumberSub`	`xaoSvcCtx`

If your application uses OCI calls that are not currently available in DB2, you can replace them using combinations of supported calls.

Because the DB2CI provides a high level of compatibility with Oracle OCI calls, you need only to compile and link the application using the DB2 specific include file `db2ci.h` and the DB2CI library, for example, `libdb2ci.a` on AIX. A sample DB2 OCI program is provided in Appendix D, “DB2CI sample program” on page 345.

5.7 Converting Open Database Connectivity applications

Open Database Connectivity (ODBC) is similar to the DB2 Call Level Interface (DB2 CLI). Applications that are based on ODBC can connect to the most popular databases. Thus, the ODBC application conversion is relatively simple. All the ODBC Core Level, Level 1, and Level 2 functions are supported in DB2 CLI, except for SQLDriver. The SQL data types that are defined in DB2 DB2 CLI are the same as the ODBC data types.

You might have to convert the following types of vendor-specific features in your application:

- Non-standard SQL statement syntax
- Possible changes in calling stored procedures and functions
- Possible logic changes because of the different concurrency models

Your current development environment remains the same. For a more detailed description of the necessary steps, see 5.2, “Application enablement planning” on page 249.

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 the use of 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 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 were added to help the application programmer specifically use unique 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 the IBM Data Server Driver Package (DS Driver), IBM Data Server Driver for ODBC and CLI (CLI Driver), and IBM Data Server Runtime Client. Support for building and running DB2 CLI applications is contained in the IBM Data Server Client.

The DB2 CLI/ODBC driver automatically binds on the first connection to the database, provided the user has the appropriate privilege or authorization. The administrator might want to perform the first connection or explicitly bind the required packages.

Procedure

For a DB2 CLI application to access a DB2 database successfully, complete the following 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 by running 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

You can use Perl to connect to Oracle and DB2 databases. The examples in this section create a stored procedure and a Perl program to demonstrate the following syntactical differences between the Oracle and DB2 databases:

- Connecting to a database using Perl
- Calling a stored procedure with an input and an output parameter
- Returning an output parameter

Example 5-30 is an Oracle stored procedure named Greeting. It contains an input parameter *name*, and an output parameter *message*.

Example 5-30 Oracle stored procedure named Greeting

```
CREATE OR REPLACE PROCEDURE Greeting (name IN VARCHAR2, message OUT
VARCHAR2)
AS
BEGIN
   message := 'Hello ' || UPPER(name) ||', the date is: ' || SYSDATE;
END;
```

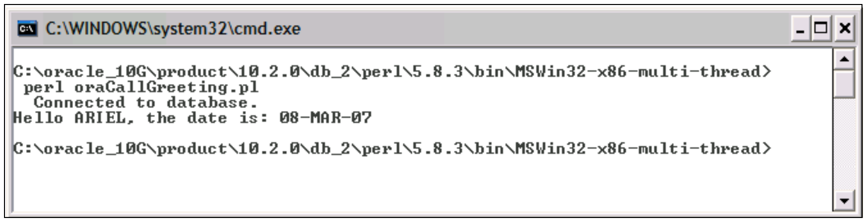
Example 5-31 shows the `oraCallGreeting.pl` Perl program. This program connects to the Oracle database, binds the input and output parameters, run the call to the Greeting stored procedure, and returns the output parameter.

Example 5-31 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:\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 running the oraCallGreeting.pl Perl program

This section demonstrates how to connect to DB2 using Perl.

Example 5-32 is a DB2 stored procedure that has the same function and same name as the Oracle stored procedure described earlier. Because DB2 also supports the Oracle PL/SQL syntax, the procedure looks the same.

Example 5-32   DB2 stored procedure named 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 might be necessary to convert the Oracle Perl application to use DB2. In addition to entering the correct values for the user ID and password, you must complete the following steps:

- Observe the syntax difference in the parameters for the *connect* method and make the necessary changes.
- Observe the syntax differences for calling stored procedures and make the necessary changes.

5.8.1 DB2 Connect method syntax

The syntax for a database connection to DB2 is shown in Example 5-33.

Example 5-33 Generic syntax for a DB2 connection string in a Perl application

```
$dbhandle = DBI->connect('dbi:DB2:dbalias', $userID, $password)
```

The parameters of this connection are as follows:

- `dbhandle`: Represents the database handle returned by the connect statement.
- `dbalias`: Represents a DB2 alias that is cataloged in the DB2 database directory.

  When you connect to an Oracle database, the `sid` of the database is used 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`: Represents the user ID used to connect to the database.
- `password`: Represents the password for the user ID that is used to connect to the database.

5.8.2 Syntax for calling a DB2 stored procedures

In an Oracle Perl application, 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 5-34 demonstrates these points.

Example 5-34 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 run by issuing a **CALL** statement from within a **PREPARE** statement. Also, the stored procedure input and output parameters are indicated by parameter markers (?, ?), as shown in Example 5-35.

Example 5-35 Calling a stored procedure in a DB2 Perl program

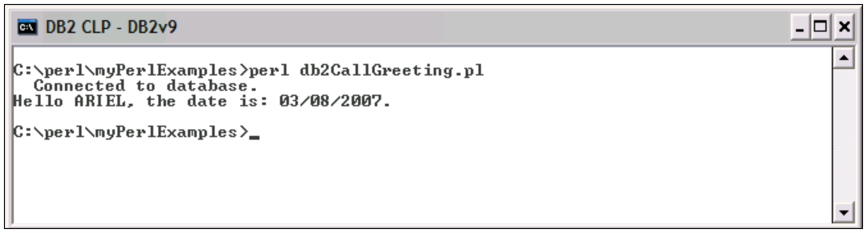
```
$sth = $dbh->prepare(q{
        CALL Greeting(?,?);
});
```

The complete Perl program, converted to DB2, is shown in Example 5-36.

Example 5-36 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 running `db2CallGreeting.pl` are shown in Figure 5-4.



```
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 running 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 extension is the preferred one for PHP connections to an Oracle database.

5.9.1 Connecting to Oracle using PDO

Example 5-37 shows the `oraGreeting.php` Oracle PHP program. This program connects to the Oracle default database, because the optional dbname is not specified after OCI.

Example 5-37 Connecting to Oracle using PDO

```
<?php
try {
  $dbh = new PDO('OCI:', 'userid', 'password');
  echo "Connected\n";
} catch (Exception $e) {
  echo "Failed: " . $e->getMessage();
}
?>
```

5.9.2 Connecting to DB2 using PDO

Because 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 the PDO class. Only the data source name is mandatory. However, you can pass its user ID, password, and more parameters for passing in tuning information. The program that is shown in Example 5-38 connects to the DB2 sample database.

Example 5-38 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 5-39 shows the call to the Greeting procedure after the connection is established.

Example 5-39 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 described in 5.1.1, “Driver support” on page 244.

The following section contains a sample program that demonstrates differences between PHP programming for Oracle Database and DB2. This sample program uses the same stored procedure, Greeting, that was shown in 5.8, “Converting Perl applications” on page 284.

5.9.3 Connecting to an Oracle database using PHP (OCI8)

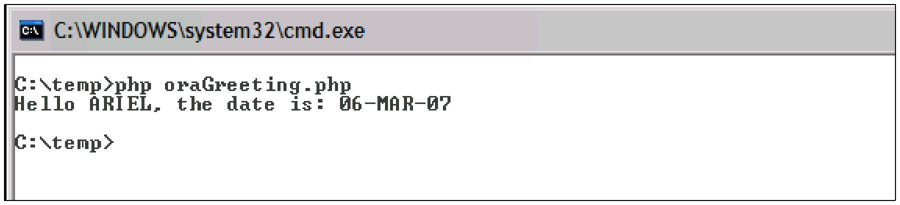
Example 5-40 shows the `oraGreeting.php` Oracle PHP program. This program connects to the Oracle database using the OCI8 Extension Module, binds the input and output parameters, runs the call to the Greeting stored procedure, and returns the output parameter.

Example 5-40 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 running `oraGreeting.php` is shown in Figure 5-5.



```
C:\temp>php oraGreeting.php
Hello ARIEL, the date is: 06-MAR-07

C:\temp>
```

Figure 5-5 The result of running oraGreeting.php

5.9.4 Connecting PHP applications to a DB2 database

Two extensions, `ibm_db2` and `PDO_ODBC`, can be used to access DB2 databases from a PHP application. For more information, see “PHP extensions” on page 245. For the DB2 conversion of the Oracle PHP program that is shown in Example 5-40 on page 290, we used the ibm_db2 extension.

Example 5-41 shows the source code for this converted program.

Example 5-41 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=$user;PWD=$passwo//r
d;";

// $conn = db2_connect($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 shown in this example, you must complete a few changes before the application can use DB2. These changes 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`**

The oci_connect and db2_connect functions

The **`oci_connect`** function takes the following required and optional parameters (shown in [ ]):

```
(string $username, string $password [, string $db [, string $charset [,
int $session_mode]]] )
```

Parameter note: Database (`$db`) is an *optional* parameter. If the database is *not* specified, PHP uses the environment variables `ORACLE_SID` and `TWO_TASK` to determine the name of the local Oracle instance and the location of the `tnsnames.ora` file.

Replace the **`oci_connect`** function with the **`db2_connect`** function, which takes the following required and optional (shown in [ ]) parameters:

```
(string database, string username, string password, [array options])
```

When you connect to DB2 in a PHP application, the connection can be made to either a cataloged or an uncataloged database.

Additional information: For detailed information about cataloging a DB2 database, see *Database Administration Concepts and Configuration Reference*, SC27-3871.

For an uncataloged connection to a database, the *database* parameter represents a complete connection string in the format that is shown in Example 5-42.

Example 5-42 Connection string for an uncataloged DB2 database

```
DRIVER={IBM DB2 ODBC DRIVER};DATABASE=database;HOSTNAME=hostname;
PORT=port;PROTOCOL=TCPIP;UID=username;PWD=password
```

Example 5-43 shows the relevant code.

Example 5-43 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, ‘’, ‘’);
```

The oci_bind_by_name and db2_bind_param functions

The **`oci_bind_by_name`** function takes the following required and optional (shown in [ ]) parameters:

```
( resource $statement, string $ph_name, mixed &$variable [, int
$maxlength [, int $type]] )
```

Replace the **`oci_bind_by_name`** function with the **`db2_bind_param`** function, 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 parameters are as follows:

stmt	A prepared statement that is returned from `db2_prepare()`.
parameter-number	Specifies the 1-indexed position of the parameter in the prepared statement.
variable-name	A string that specifies 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, you bind the stored procedure input and the output parameters are converted, as shown in Example 5-44.

Example 5-44 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);
```

The oci_parse and db2_prepare functions

The **`oci_parse`** function prepares a query. The function accepts the following parameters:

```
( resource $connection, string $query )
```

The **`oci_parse`** function is converted to the **`db2_prepare`** function, which accepts the following required and optional (shown in [ ]) parameters:

```
(resource connection, string statement, [array options])
```

The parameters are defined as follows:

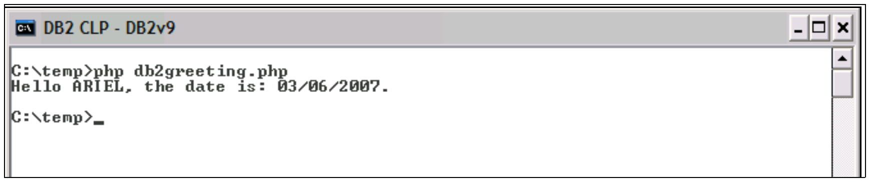
`connection`	A valid database connection resource variable as returned from **`db2_connect()`** or **`db2_pconnect()`**.
`statement`	An SQL statement, optionally containing one or more parameter markers.
`options`	(Optional) An associative array that contains statement options. You can use this parameter 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 5-45.

Example 5-45 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 these changes are implemented, the application is fully converted to DB2. Figure 5-6 shows the result of running this program.



```
C:\temp>php db2greeting.php
Hello ARIEL, the date is: 03/06/2007.

C:\temp>_
```

Figure 5-6 The result of running db2Greeting.php

Note: For complete information about PHP, see *Developing Perl, PHP, Python, and Ruby on Rails Applications*, SC27-3876.

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)

DB2 10.5 does not support the .NET Framework 1.1. The supported operating systems for developing and deploying .NET Framework 2.0, 3.0, 3.5, and 4.0 applications are:

- Windows XP, Service Pack 2 (32-bit and 64-bit editions)
- Windows Server 2003 (32-bit and 64-bit editions)
- Windows Vista (32-bit and 64-bit editions)
- Windows Server 2008 (32-bit and 64-bit editions)
- Windows Server 2008 R2 (64-bit edition)
- Windows 7 (32–bit and 64–bit editions)

5.10.1 Supported development software for .NET Framework applications (DB2 9.7)

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)

5.10.2 Supported development software for .NET Framework applications (DB2 10.5)

The following Visual Studio versions are supported:

- Visual Studio 2008
- Visual Studio 2010

5.10.3 Supported deployment software for .NET Framework applications (in general)

The following .NET Framework versions are supported:

- .NET Framework 1.1 Redistributable Package (DB2 9.7 only)
- .NET Framework 2.0 Redistributable Package
- .NET Framework 3.0 Redistributable Package
- .NET Framework 3.5 Redistributable Package
- .NET Framework 4.0 Redistributable Package

5.10.4 .NET Data Providers

DB2 for Linux, AIX, and Windows includes the following .NET Data Providers:

- DB2 .NET Data Provider

  A high performance, managed ADO.NET Data Provider. This provider is the *recommended* .NET Data Provider for access to DB2 family databases. ADO.NET database access using the DB2 .NET Data Provider has fewer restrictions, and provides better performance than the OLE DB and ODBC .NET bridge providers. This provider 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 IBM Data Studio and DB2 for Linux, UNIX, and Windows and the host databases using IBM 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 website:

`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 5-46.

Example 5-46 Examples of the required Imports or using statement

```
[Visual Basic]
Imports IBM.Data.DB2

[C#]
using IBM.Data.DB2;
```

Also, you need to add references to `IBM.Data.db2.dll` and `IBM.Data.DB2.Server.dll` to the project.

5.10.5 Visual Basic .NET conversion example

In general, converting a .NET application from Oracle to DB2 is simple. In most cases, converting means *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, replacing OracleConnection with DB2Connection or OracleCommand with DB2Command.

The example in this section demonstrates this approach using a simple Visual Basic .NET application that connects to a database, runs a **`SELECT`** statement, and returns a result set. The DB2 example outlines the changes that are necessary for converting from Oracle to DB2.

Components of the GUI for the conversion example

The GUI used for both examples consists of several *controls*:

- A RUN QUERY button

  Clicking this button connects to the database and runs the query.

- A Query Results text box (for aesthetic purposes only)
- A list box

  When the query runs, the results display in this list box.
- A Quit button

  Clicking this button ends the application.

Figure 5-7 shows the GUI used to demonstrate the Visual Basic .NET application conversion example.

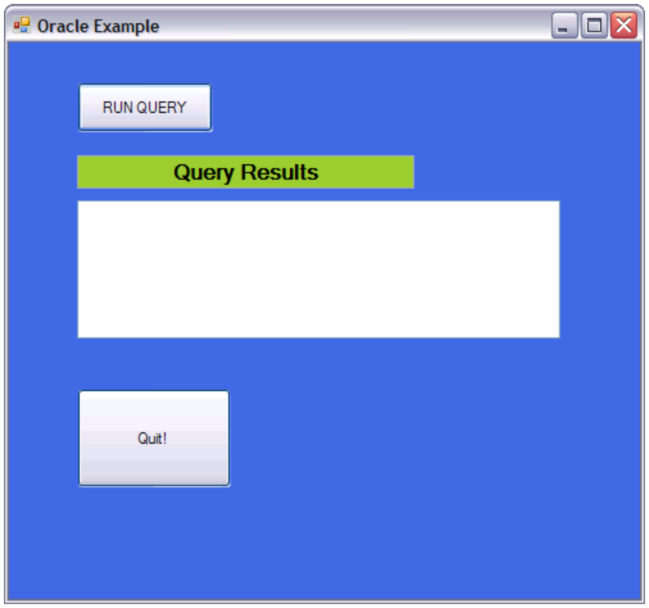


Figure 5-7 GUI for the Visual Basic .NET application conversion example

The essential components of this application are contained within the Click event for the RUN QUERY control button. Example 5-47 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 5-47 The Button1_Click event

```
Imports Oracle.DataAccess.Client ' ODP.NET Oracle managed provider [1]
```

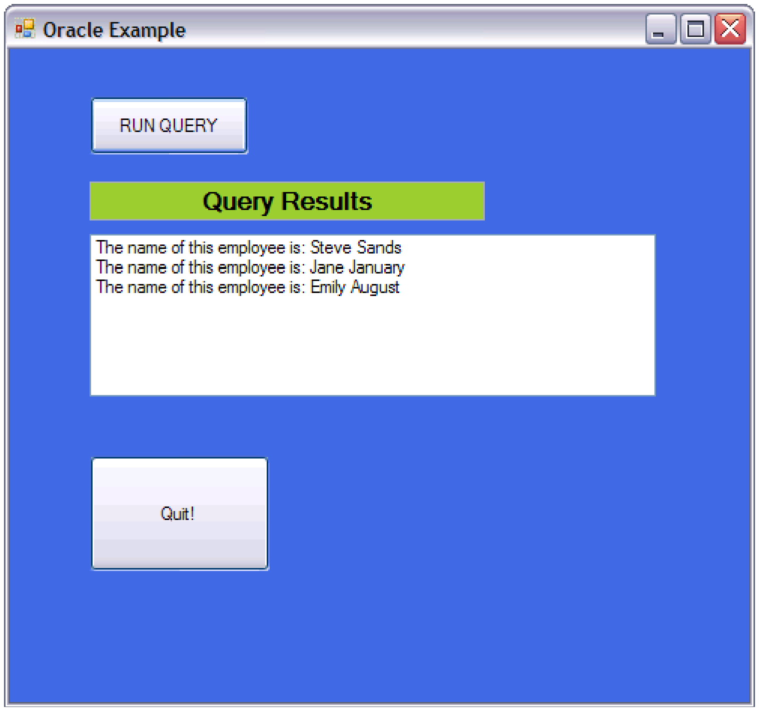
```
Public Class Form1

    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 about the Button1_Click event:

- **`IMPORT Oracle.DataAccess.Client`** is added to the application .DLL.
- A string (`OraDb`) is declared as the connection string for the Oracle database.
- A connection (`conn`) is defined as an OracleConnection.
- A command (`cmd`) is defined as an OracleCommand and populated with the text of the query.
- A DataReader (`dr`) is defined as an OracleDataReader and the query is run.
- The List Box is populated with the results of the query.

When the application runs, clicking **RUN QUERY** yields the results that are shown in Figure 5-8.



The name of this employee is: Steve Sands
The name of this employee is: Jane January
The name of this employee is: Emily August

Figure 5-8 The results of the Oracle Example are displayed in the list box

Converting the DB2 Example application

Figure 5-7 on page 299 shows the GUI for the DB2 Example conversion.

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 5-48 shows the code in the Button1_Click event as it appears after conversion to DB2. The numbers that show to the right of the code example lines correspond to the numbers in the explanatory note that follows the example.

Example 5-48 The code in the Button1_Click event after conversion

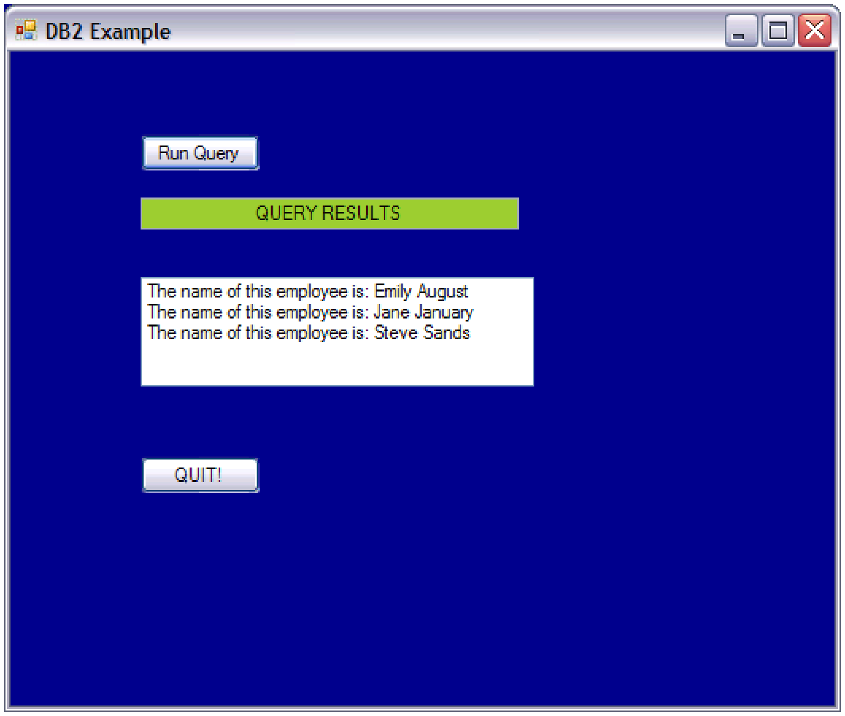
```
Imports IBM.Data.DB2                                                   [1]
_______________________________________________________________________
Public Class Form1

   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 about the Button1_Click event (after conversion):

1. To use the DB2 .NET Data Provider, you must add the `Imports` (Visual Basic) 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.

After you implement the changes, click **RUN QUERY** to produce the results that are shown in Figure 5-9.



```
The name of this employee is: Emily August
The name of this employee is: Jane January
The name of this employee is: Steve Sands
```

Figure 5-9 The results of running the DB2 Example application

For further information: You can find more information about the DB2 .NET provider at:

`http://pic.dhe.ibm.com/infocenter/db2luw/v10r5/topic/com.ibm.swg.im.dbclient.adonet.doc/doc/c0010960.html`

For in-depth information about .NET programming, see *Developing ADO.NET and OLE DB Applications*, SC27-3873.

A

Terminology mapping

All relational database systems, including Oracle Database and DB2, are based upon similar concepts. At the same time, each database platform operates with a set of terms that might be different. These differences in terminology can present difficulties when you are converting from one platform to another.

Table A-1 introduces the most common Oracle database terms and provides the DB2 equivalent of the term.

Table A-1 Mapping of Oracle database terminology to DB2

Oracle term	DB2 term	Comments
Alert log	Diagnostic log	A file where instance- and database-level error messages and warnings are recorded.
Data blocks	Pages	The smallest storage entities in the storage model.
Data cache	Buffer pools	Memory areas that cache table and index pages when they are read from disk. A DB2 database must have at least one buffer pool, and each table space can be assigned a separate buffer pool.
Data dictionary	System catalog	A collection of tables and views that store the database metadata.

Oracle term	DB2 term	Comments
Data files	Containers	Physical structures on a disk that hold the table space contents. The *automatic storage* feature in DB2 is responsible for the container management (allocation, resizing, collation of empty space, and so on.)
Database	Database	A collection of objects that represents data and routines (stored procedures, functions, and so on).
Database link (DBlink)	Federated server	In an Oracle database, this object describes a connection from one database to another. In DB2, a federated server definition is used for this purpose. Depending on the DB2 edition, you might need to install IBM InfoSphere Federation Server to be able to create servers and other federated objects.
Extents	Extents	Physical space allocation units within table spaces.
Instance	Instance	Includes processes and shared memory. In DB2, it also includes a permanent directory structure. An instance is usually created at installation time and must exist before a database can be created. A DB2 instance is also known as the *database manager* (DBM). A DB2 instance can support multiple databases. An Oracle database instance can support only one database.
Large pool	Utility heap	Memory area that is used by database utilities, such as backup and restore.
Oracle EE	DB2 Enterprise Server Edition	Enterprise-level database server product.
Oracle Transparent Gateway	DB2 Connect InfoSphere Federation Server	Access to DB2 databases on IBM i and System z platforms and as heterogeneous data sources.
Package	Module	A database object that logically groups PL/SQL (SQL PL) program objects (routines), such as stored procedures and functions.
N/A	Package	A precompiled access plan for an embedded static SQL application that is stored in the server.
`pfile` and `spfile`	DBM and database configuration	Configuration settings for the instance and the database.

Oracle term	DB2 term	Comments
PL/SQL	SQL Procedural Language (SQL PL)	Programming language extension to SQL. DB2 stored procedures can be programmed in SQL PL (a subset of the PSM standard), Java, C, C++, COBOL, Fortran, OLE, and REXX. DB2 functions can be programmed in Java, C, C++, OLE, or SQL PL.
Process Global Area (PGA)	Agent / application shared memory	Shared memory area to store user-specific data that is passed between an application process and the database server.
Redo logs	Database logs	Files that store information required for the database recovery in case of failure.
Statement cache	Package cache	A memory area that caches prepared dynamic SQL statements.
SQL*PLUS	CLPPlus	The command-line interface to the database server.
System Global Area (SGA)	Instance and database shared memory	Shared memory area or areas for the database server. In Oracle, there is one shared memory area. In DB2, there is one shared memory area at the database manager (instance) level and one for each active database.
Table spaces	Table spaces	Logical structures that contain database tables and other objects.
Undo table spaces	N/A	Store the *before* image of data when they are being modified. In DB2, the *before* image of data are stored in the database log along with the *after* image.

B

Data types

This appendix explains data types in the following 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 type.

For more information about mapping between SQL data types and C and C++ data types, see 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/v10r5/topic/com.ibm.db2.luw.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/v10r5/topic/com.ibm.db2.luw.apdv.embed.doc/doc/r0006094.html

Table B-1 Oracle to DB2 data type mapping

Item	SQL data type sqltype	C/C++ type	sqllen	Description
Integer	SMALLINT (500 or 501)	short short int sqlint 16	2	▸ 16-bit signed integer. ▸ Range of -32,768 to 32,767). ▸ Precision of 5 digits.
	INTEGER INT (496 or 497)	long long int sqlint32	4	▸ 32-bit signed integer. ▸ Range of -2,147,483,648 to 2,147,483,647). ▸ Precision of 10 digits.
	BIGINT (492 or 493)	long long long __int64 sqlint64	8	64-bit signed integer.

Item	SQL data type sqltype	C/C++ type	sqllen	Description
Floating point	REAL FLOAT (480 or 481)	float		► Single precision floating point. ► 32-bit approximation of a real number. ► FLOAT(n) can be a synonym for REAL if 0 < n < 25.
	DOUBLE (480 or 481) DOUBLE PRECISION	double	8	► Double precision floating point. ► 64-bit approximation of a real number. ► Ranges of 0, -1.79769E+308 to -2.225E-307, and 2.225E-307 to 1.79769E+308). ► FLOAT(n) can be a synonym for DOUBLE if 24 < n < 54.

Decimal	DECIMAL(p,s)
	DEC(p,s)
	(484 or 485)
	
	NUMERIC(p,s)
	NUM(p,s)
		double / decimal	p/2+1	► Packed decimal.
				► If precision /scale is not specified, the default is (5.0).
				► The max precision is 31 digits, and the max range is between -10E31+1 to 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	► Null-terminated character form (11 characters) or varchar struct form (10 characters).
				► struct can be divided as wanted to obtain the individual fields.
				► Example: 11/02/2000.
				► Stored internally as a packed string of 4 bytes.

	TIME (388 or 389)	char	8	► Null-terminated character form (9 characters) or varchar struct form (8 characters). ► struct can be divided as wanted to obtain the individual fields. ► Example: 19:21:39. ► Stored internally as a packed string of 3 bytes.
	TIMESTAMP (392 or 393)	char	26	► Null-terminated character form or varchar struct form. ► Allows 19 - 32 characters. ► struct can be divided as wanted 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**	**sqllen**	**Description**
character	CHAR(1) (452 or 453)	char	1	Single character.
	CHAR(n) (452 or 453)	char	n	▸ Fixed-length character string that consists of n bytes. ▸ Use char[n+1] where 1 <= n <= 254. ▸ If the length is not specified, it defaults to 1.
	VARCHAR (460 or 461)	char	n	▸ 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	▸ 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	▸ Non-null-terminated varying character string with 2-byte string length indicator. ▸ se char[n] in struct form where 32673<= n <= 32700.
	CLOB(n) (408 or 409)	clob	n	▸ 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 on the server.
	CLOB (920 or 921)	clob_file		Descriptor for the file that contains the CLOB data.
Binary	BLOB(n) (404 or 405)	blob	n	▸ 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 se.rver.
	BLOB (916 or 917)	blob_file		Descriptor for the file that contains the BLOB data.

Item	SQL data type sqltype	C/C++ type	sqllen	Description
Double-byte	GRAPHIC(1) GRAPHIC(n) (468 or 469)	sqldbchar	24	► sqldbchar is a single double-byte character string. ► For a fixed-length graphic string of length integer, which can range from 1 to 127. If the length specification is omitted, a length of 1 is assumed, ► Precompiled with the **WCHARTYPE NOCONVERT** option.
	VARGRAPHIC(n) (464 or 465)	struct { short int; sqldbchar[n] } tag; alternately: sqldbchar[n+1]	n*2+4	► For a varying-length graphic string of maximum length integer, which can range 1 - 16336. ► Precompiled with the **WCHARTYPE NOCONVERT** option. ► Null terminated variable-length.
	LONG VARGRAPHIC(n) (472 or 473)	struct { short int; sqldbchar[n] } tag;		► For a varying-length graphic string with a maximum length of 16350 and a 2-byte string length indicator 16337<=n <=16350. ► Precompiled with the **WCHARTYPE NOCONVERT** option.
	DBCLOB(n) (412 or 413)	dbclob		► For non-null-terminated varying double-byte character large object maximum length in double-byte characters. ► 4-byte string length indicator. ► Use dbclob(n) where 1<=n <= 1073741823 double-byte characters. ► Precompiled with the **WCHARTYPE NOCONVERT** option.
	DBCLOB	dbclob_locator		► Identifies DBCLOB entities on the server. ► Precompiled with the **WCHARTYPE NOCONVERT** option.
	DBCLOB	dbclob_file		► Descriptor for a file that contains the DBCLOB data. ► Precompiled with the **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]; }		► 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 that is exchanged between the application and the database by 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, see *Developing Embedded SQL Applications*, SC27-2445 or the DB2 Information Center, found at:

`http://publib.boulder.ibm.com/infocenter/db2luw/v10r5/topic/com.ibm.db2.luw.apdv.routines.doc/doc/r0006275.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 - 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 - 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 - 254	Fixed-length character string data of length *n*
CHARACTER VARYING (*n*)	VARCHAR (*n*) *n* = 1 - 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 (*n*)	CLOB (*n*) n = 1 - 2 147 483 647	Fixed-length long character string data of length *n*
DATE	DATE [a]	Date and time data (expressed to the second)
DEC	DEC (9, 2)	Decimal numeric data
DEC (**p**)	DEC (*p*) *p* = 1 - 31	Decimal numeric data of precision *p*
DEC (p, s)	DEC (*p, s*) *p* = 1 - 31; *s* = 1 - 31	Decimal numeric data of precision p and scale *s*
DECIMAL	DECIMAL (9, 2)	Decimal numeric data
DECIMAL (*p*)	DECIMAL (*p*) *p* = 1 - 31	Decimal numeric data of precision *p*
DECIMAL (*p, s*)	DECIMAL (*p, s*) *p* = 1 - 31; *s* = 1 - 31	Decimal numeric data of precision p and scale *s*
DOUBLE	DOUBLE	Double precision floating-point number
DOUBLE PRECISION	DOUBLE PRECISION	Double precision floating-point number
FLOAT	FLOAT	Float numeric data
FLOAT (*n*) *n* = 1 - 24	REAL	Real numeric data
FLOAT (*n*) n = 25 - 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 (*n*) *n* = 1 - 2000	GRAPHIC (*n*) *n* = 1 - 127	Fixed-length graphic string data of length *n*
NCLOB[b] 2	DBCLOB (1 MB)	Double-byte character large object data
NCLOB (*n*)	DBCLOB (2000)	Double-byte long character string data of maximum length *n*
NVARCHAR2	VARGRAPHIC (2048) or NVARCHAR2(2048)	Variable-length graphic string data or national string that is coerced back to VARGRAPHIC

PL/SQL data type	DB2 SQL data type	Description
NVARCHAR2 (*n*)	VARGRAPHIC (*n*) NVARCHAR2(n)	Variable-length graphic string data of maximum length *n*, or national string that is coerced back to VARGRAPHIC
NUMBER	NUMBER[c]	Exact numeric data
NUMBER (*p*)	NUMBER (*p*)[c]	Exact numeric data of maximum precision *p*
NUMBER (*p, s*)	NUMBER (*p, s*)[c] p = 1 - 31	Exact numeric data of maximum precision *p* and scale *s*
NUMERIC	NUMERIC (9.2)	Exact numeric data
NUMERIC (*p*)	NUMERIC (*p*) *p* = 1 - 31	Exact numeric data of maximum precision *p*
NUMERIC (*p, s*)	NUMERIC (*p, s*) *p* = 1 - 31; *s* = 0 - 31	Exact numeric data of maximum precision p and scale *s*
PLS_INTEGER	INTEGER	Integer numeric data
RAW	BLOB (32767)	Binary large object data
RAW (*n*)	BLOB (*n*) *n* = 1 - 32 767	Binary large object data
SMALLINT	SMALLINT	Signed 2-byte integer data
TIMESTAMP (0)	TIMESTAMP (0)	Date data with time stamp information
TIMESTAMP (*p*)	TIMESTAMP (*p*)	Date and time data with optional fractional seconds and precision *p*
VARCHAR	VARCHAR (4096)	Variable-length character string data with a maximum length of 4096 characters
VARCHAR (*n*)	VARCHAR (*n*)	Variable-length character string data with a maximum length of n characters
VARCHAR2 (*n*)	VARCHAR2 (*n*)[d]	Variable-length character string data with a maximum length of n 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 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)	NVARCHAR2(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	The Oracle default format is DD-MON-YY.
DATE (only the date)	DATE (MM/DD/YYYY)	Use the Oracle `TO_CHAR()` function to extract data for a subsequent DB2 load.
DATE (only the time)	TIME (HH24:MI:SS)	Use the Oracle `TO_CHAR()` function to extract for a subsequent DB2 load.
TIMESTAMP (0)	TIMESTAMP (0)	Date data with time stamp 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 that are created in a Unicode database or a column that is created with a CCSID clause to a Unicode compatible code set.

C

Built-in modules

This appendix provides details for the following built-in modules, equivalent to Oracle Database supplied PL/SQL packages, that are supported in DB2 10:

- DBMS_ALERT
- DBMS_DDL
- DBMS_JOB
- DBMS_LOB
- DBMS_OUTPUT
- DBMS_PIPE
- DBMS_SQL
- DBMS_UTILITY
- UTL_DIR
- UTL_MAIL
- UTL_SMTP

C.1 DBMS_ALERT

The DBMS_ALERT module 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 module for a particular database. The DBMS_ALERT module requires that the database configuration parameter **`CUR_COMMIT`** to be set to `ON`.

Table C-1 lists the system-defined routines included in the DBMS_ALERT module.

Table C-1 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-1 shows the use of DBMS_ALERT routines. Assume the “AlertFromTrigger” alert is signaled as a result of an insert, which fires a trigger named TRIG1.

Example: C-1 Signaling from a 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 through the WAITONE routine after you register the alert name with a 60-second timeout, as shown in Example C-2.

Example: C-2 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:

http://publib.boulder.ibm.com/infocenter/db2luw/v10r5/topic/com.ibm.db2.luw.apdv.sqlpl.doc/doc/r0053671.html

C.2 DBMS_DDL

The DBMS_DDL module implements the routines that are listed in Table C-2 for obfuscating ("wrapping" in Oracle terms) DDL statements and the database objects that are created by such statements. You can use this function to deploy database objects without displaying the procedural logic in them.

Table C-2 System-defined routines available in the DBMS_DDL module

Routine name	Description
WRAP function	Produces an obfuscated version of the DDL statement that is provided as an argument.
CREATE_WRAPPED procedure	Deploys a DDL statement in the database in an obfuscated format.

Example C-3 shows the use of DBMS_DDL.WRAP for generating the obfuscated version of a user-defined function.

Example: C-3 Generating an obfuscated user-defined function.

```
VALUES(DBMS_DDL.WRAP('CREATE OR REPLACE TRIGGER ' ||
                      'trg1 BEFORE INSERT ON emp ' ||
                      'REFERENCING NEW AS n ' ||
                      'FOR EACH ROW ' ||
                              'WHEN (n.bonus IS NULL) ' ||
                              'SET n.bonus = n.salary * .04'))
/
CREATE OR REPLACE TRIGGER trg1  WRAPPED SQL10010
ablGWmdiWmtiTmdqTmtGTmtmUnteUmdCUnZa3mti5idaWmdaWmdaXmdyWncaGicaGUJO7oU
H:g3mwlXdtHb6:oPhVssP6gnLJPu4wN_yhwnGKJSFJcz8PHP79VYw79mMzxNtL:beJXAUZr
CwSnLFxNwtMI7LFStkj8J9IInSFmC2FR:hh_4e:zskecrzu9o6zsS5WhgRKudjOkbKT
```

For more information about this module, see the Information Center at:

http://publib.boulder.ibm.com/infocenter/db2luw/v10r5/topic/com.ibm.db2.luw.apdv.sqlpl.doc/doc/r0057137.html

C.3 DBMS_JOB

This module provides a set of procedures for creating, scheduling, and managing jobs. DBMS_JOB is an alternative interface for the DB2 Administrative Task Scheduler (ATS). To use the DBMS_JOB module, you must activate the ATS. This facility is turned off by default, although you are still able to define and modify jobs (tasks). You can enable the ATS by setting the registry variable as follows:

```
db2set DB2_ATS_ENABLE=YES
```

For more information, see the "Setting up the administrative task scheduler" topic in the Information Center at:

http://publib.boulder.ibm.com/infocenter/db2luw/v10r5/topic/com.ibm.db2.luw.admin.gui.doc/doc/t0054396.html

Table C-3 shows the routines in the DBMS_JOB module.

Table C-3 Procedures that are provided by the DBMS_JOB module

Routine name	Description
BROKEN procedure	Specifies that a job is either broken or not broken.
CHANGE procedure	Changes the parameters of the job.
INTERVAL procedure	Sets the execution frequency through a date function that is recalculated each time the job runs. This value becomes the next date and time for execution.

Routine name	Description
NEXT_DATE procedure	Sets the next date and time when the job is to be run.
REMOVE procedure	Deletes the job definition from the database.
RUN procedure	Forces execution of a job even if it is marked as broken.
SUBMIT procedure	Creates a job and stores the job definition in the database.
WHAT procedure	Changes the stored procedure that is run by a job.

Example C-4 shows the creation of a job that is called job_proc that first runs immediately (at SYSDATE) and is scheduled to repeat every 24 hours.

Example: C-4 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;
/
```

For more information about the DBMS_JOB module, see the Information Center at:

http://publib.boulder.ibm.com/infocenter/db2luw/v10r5/topic/com.ibm.db2.luw.apdv.sqlpl.doc/doc/r0055103.html

C.4 DBMS_LOB

This module offers a set of routines for operating on large objects (LOBs). These routines are listed in Table C-4.

Table C-4 Routines in the DBMS_LOB module

Routine name	Description
`APPEND` procedure	Appends one large object to another.
`CLOSE` procedure	Closes an open large object.
`COMPARE` function	Compares two large objects.
`CONVERTTOBLOB` procedure	Converts character data to binary data.
`CONVERTTOCLOB` procedure	Converts binary data to character data.
`COPY` procedure	Copies one large object to another.
`ERASE` procedure	Erases a large object.
`GET_STORAGE_LIMIT` function	Gets the storage limit for large objects.
`GETLENGTH` function	Gets the length of the large object.
`INSTR` function	Gets the position of the *n*th occurrence of a pattern in the large object starting at offset.
`ISOPEN` function	Checks if the large object is open.
`OPEN` procedure	Opens a large object.
`READ` procedure	Reads a large object.
`SUBSTR` function	Gets part of a large object.
`TRIM` procedure	Trims a large object to the specified length.
`WRITE` procedure	Writes data to a large object.
`WRITEAPPEND` procedure	Writes data from the buffer to the end of a large object.

In Example C-5, the APPEND and ERASE routines are started from the DB2 command line processor to demonstrate some actions that can be performed on a LOB.

Example: C-5 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
```

Example C-6 shows the use of DBMS_LOB routines in a PL/SQL block.

Example: C-6 DBMS_LOB in an 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.5 DBMS_OUTPUT

The DBMS_OUTPUT module provides a set of procedures that you can us to work with the message buffer by putting lines of text (messages) in the message buffer and getting messages from the buffer. These procedures can be useful during application debugging when you must write messages to standard output. You could use the command line processor command **`SET SERVEROUTPUT ON`** to redirect messages to standard output.

Table C-5 lists the system-defined routines included in the DBMS_OUTPUT module.

Table C-5 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-7 shows the use of DBMS_OUTPUT, and the messages that are produced by the **`PUT`** and **`PUT_LINE`** procedures.

Example: C-7 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.6 DBMS_PIPE

The DBMS_PIPE module provides a set of routines for sending messages through a pipe within a session 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. 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 be accessed only 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 module. Access level for a pipe is specified in a call to the **CREATE_PIPE** function. If no value is specified, the default is to create a private pipe. All implicit pipes are private.

Table C-6 lists the system-defined routines included in the DBMS_PIPE module.

Table C-6 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.

Routine name	Description
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. You can use the separate message buffers 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-8 shows a simple anonymous block that creates a pipe and sends a message through the pipe.

Example: C-8 Sending a message through a 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-9 shows reading the message from a pipe.

Example: C-9   Receiving a message from a 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;
/
```

You can find more information and examples about DBMS_PIPE in the DB2 Information Center at:

http://publib.boulder.ibm.com/infocenter/db2luw/v10r5/topic/com.ibm.db2.luw.apdv.sqlpl.doc/doc/r0053678.html

C.7 DBMS_SQL

The DBMS_SQL module provides a collection of procedures for running dynamic SQL. The routines in the DBMS_SQL module are useful when you want to construct and run SQL statements dynamically at run time or call a function that uses dynamic SQL from within an SQL statement. DDL statements, such as **ALTER TABLE** or **DROP TABLE**, can also be prepared and run when needed.

Table C-7 lists the system-defined routines included in the DBMS_SQL module.

Table C-7 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.
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.

Procedure name	Description
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.
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).

Procedure name	Description
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 that are 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.
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).

Procedure name	Description
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.

Some of the names of the procedures in this module are different from the corresponding names in the Oracle DBMS_SQL package. For example, the names of the **DBMS_SQL.COLUMN_VALUE_*** procedures reflect the data types they handle: COLUMN_VALUE_NUMBER, COLUMN_VALUE_CHAR, or COLUMN_VALUE_DATE.

Table C-8 lists the system-defined types and constants available in the DBMS_SQL module.

Table C-8 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 that is supported for the **language_flag** parameter of the **PARSE** procedure

The **EMPLOYEE_DYNAMIC_QUERY** procedure in Appendix E, “Code samples” on page 351 shows an example of using DBMS_SQL.

C.8 DBMS_UTILITY

The DBMS_UTILITY module provides various utility routines for gathering statistics for the entire database or a schema, compiling or validating objects, running DDL statements, and getting information about database objects.

Table C-9 lists the system-defined routines available in the DBMS_UTILITY module.

Table C-9 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, and modules) 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 processor time in hundredths of a second from some arbitrary point in time.
`GET_DEPENDENCY` procedure	Lists all objects that are dependent upon the object.
`GET_HASH_VALUE` function	Computes a hash value for a 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 quotation marks are put into uppercase, and double quotation marks are stripped from names with double quotation marks.)

Routine name	Description
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-10 lists the system-defined variables and types available in the DBMS_UTILITY module.

Table C-10 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-10 shows the use of COMMA_TO_TABLE to change a comma delimited list into a table.

Example: C-10 COMMA_TO_TABLE examples

```
CREATE OR REPLACE PROCEDURE list_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;
/
set serveroutput on
call list_to_table('schema.dept, schema.emp, schema.jobhist');
 schema.dept
 schema.emp
 schema.jobhist
```

You can find details and examples about the usage of these procedures and variables in the DB2 Information Center at:

`http://publib.boulder.ibm.com/infocenter/db2luw/v10r5/topic/com.ibm.db2.luw.apdv.sqlpl.doc/doc/r0055155.html`

C.9 UTL_DIR

The UTL_DIR module implements routines for maintaining directory aliases that are used with the UTL_FILE module. You can create, drop, and gather information about the directory alias using this module.

Table C-11 shows the procedures that are defined in this module.

Table C-11 System-defined routines available in the UTL_DIR module

Routine name	Description
CREATE_DIRECTORY procedure	Creates a directory alias for the specified path.
CREATE_OR_REPLACE_DIRECTORY procedure	Creates or replaces a directory alias for the specified path.
DROP_DIRECTORY procedure	Drops the specified directory alias.
GET_DIRECTORY_PATH procedure	Gets the corresponding path for the specified directory alias.

Example C-11 shows how to create a directory alias.

Example: C-11 Create a directory alias

```
BEGIN
  UTL_DIR.CREATE_DIRECTORY('mydir', '/home/user/temp/mydir');
END;
/
```

If the directory alias exists, you can use the **CREATE_OR_REPLACE_DIRECTORY** procedure.

Directory information is stored in the SYSTOOLS.DIRECTORIES table, which is created in the SYSTOOLSPACE when you first reference this module for each database.

Example C-12 demonstrates how to retrieve the corresponding path for a directory alias.

Example: C-12 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-13 uses the **DROP_DIRECTORY** procedure to drop the specified directory alias.

Example: C-13 Drop a directory alias

```
BEGIN
   UTL_DIR.DROP_DIRECTORY('mydir');
END;
```

C.10 UTL_MAIL

The UTL_MAIL module provides the capability to send email messages with or without attachments. To successfully send an email message using this module, the database configuration parameter **SMTP_SERVER** must contain one or more valid Simple Mail Transfer Protocol (SMTP) server addresses.

Table C-12 lists the system-defined routines included in the UTL_MAIL module.

Table C-12 System-defined routines available in the UTL_MAIL module

Routine name	Description
SEND procedure	Packages and sends an email 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-14 shows how to set up your SMTP server entries. To set up a list of SMTP servers, separate them by commas. You can specify non-default port numbers if necessary. When it sends email messages, DB2 tries to connect to the first listed server. If it is unreachable, DB2 tries other servers in the listed order.

Example: C-14 SMTP server entry setup

```
-- Set up a single SMTP serMver 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-15 demonstrates an anonymous block that sends an email message.

Example: C-15 Sending an email with the UTL_MAIL module

```
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 module in DB2';
    v_msg := ' UTL_MAIL module 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 module provides a set of routines for sending email over SMTP. Compared to the UTL_MAIL module, UTL_SMTP gives you more control over sending email messages.

Table C-13 lists the system-defined routines included in the UTL_SMTP module.

Table C-13 System-defined routines available in the UTL_SMTP module

Routine name	Description
`CLOSE_DATA` procedure	Ends an email message.
`COMMAND` procedure	Runs an SMTP command.
`COMMAND_REPLIES` procedure	Runs an SMTP command where multiple reply lines are expected.
`DATA` procedure	Specifies the body of an email 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.
`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 email message.
`RSET` procedure	Terminates the current mail transaction.
`VRFY` procedure	Validates an email address.
`WRITE_DATA` procedure	Writes a portion of the email message.
`WRITE_RAW_DATA` procedure	Writes a portion of the email message that consists of RAW data.

Table C-14 lists the public variables available in the module.

Table C-14 System-defined types available in the UTL_SMTP module

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 **send_mail** procedure in Example C-16 constructs and sends a text email message using the UTL_SMTP module and the UTL_SMTP.DATA method. This example also demonstrates how to call this procedure.

Example: C-16 Sending a message with the UTL_SMTP module

```
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 module on DB2','Please, setup properly your
server!','smtp.mycompany.com');
```

Example C-17 uses the **OPEN_DATA**, **WRITE_DATA**, and **CLOSE_DATA** procedures instead of the **DATA** procedure. **WRITE_DATA** gives you more control over the email message header and contents. The call to the **send_mail_2** procedure is identical to the call of **send_mail** procedure in Example C-16 on page 342.

Example: C-17 Sending a message with the UTL_SMTP module

```
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;
/
```

D

DB2CI sample program

This appendix provides a sample DB2CI program.

Example D-1 shows a simplified program that uses Oracle Call Interface (OCI) functions to read data from the ORG table in the DB2 SAMPLE database. It illustrates some of the common OCI calls. This example uses utility functions that are defined in `utilci.c`, and `utilci.h` that are part of the DB2CI samples provided with DB2 10.

Example: D-1 DB2CI 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( OCIEnv * envhp, OCISvcCtx * svchp, OCIError * errhp );

int main(int argc, char *argv[])
{
  sb4 ciRC = OCI_SUCCESS;
  int rc = 0;
  OCIEnv * 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, OCIError * errhp )
{
sb4 ciRC = OCI_SUCCESS;
int rc = 0;
OCIStmt * 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 = OCIStmtExecute(
      svchp,
      hstmt,
      errhp,
      0,
      0,
      NULL,
      NULL,
      OCI_DEFAULT );
  ERR_HANDLE_CHECK(errhp, ciRC);

  /* bind column 1 to variable */
  ciRC = OCIDefineByPos(
      hstmt,
      &defnhp1,
      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,
      &defnhp2,
      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 = OCIStmtFetch(
      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 = OCIStmtFetch(
        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 */
```

The easiest way to compile this sample program is to use the **bldapp** script that is provided along with the DB2CI samples in `sqllib/samples/db2ci`, as shown in Example D-2. Copy the utility source code to the directory where you created the sample program (`myoci.c`) and run **bldapp**.

Example: D-2 Compiling and linking a DB2CI program myoci.c

```
cp /home/db2inst1/sqllib/samples/db2ci/utilci.* .
/home/db2inst1/sqllib/samples/db2ci/bldapp myoci
```

E

Code samples

This appendix provides the following Oracle enablement PL/SQL code samples:

- Oracle DDL statements that are used to create all database objects in the source Oracle database
- DB2 DDL statements that define the corresponding database objects that are enabled in the DB2 database
- An example of the DB2 10.5 deep nested objects feature

You can download these DDL scripts from the IBM Redbooks website as described in Appendix F, “Additional material” on page 425.

E.1 Code disclaimer

IBM does not warrant or represent that the code provided is complete or up-to-date. IBM does not warrant, represent or imply reliability, serviceability, or function of the code. IBM is under no obligation to update content nor provide further support.

All code is provided “as is,” with no warranties or guarantees whatsoever. IBM expressly disclaims to the fullest extent permitted by law all express, implied, statutory, and other warranties, guarantees, or representations, including, without limitation, the warranties of merchantability, fitness for a particular purpose, and non-infringement of proprietary and intellectual property rights. You understand and agree that you use these materials, information, products, software, programs, and services, at your own discretion and risk and that you will be solely responsible for any damages that may result, including loss of data or damage to your computer system.

In no event will IBM be liable to any party for any direct, indirect, incidental, special, exemplary, or consequential damages of any type whatsoever related to or arising from use of the code found herein, without limitation, any lost profits, business interruption, lost savings, loss of programs, or other data, even if IBM is expressly advised of the possibility of such damages. This exclusion and waiver of liability applies to all causes of action, whether based on contract, warranty, tort, or any other legal theories.

E.2 Oracle DDL statements

The sample Oracle DDL statements in this section are used in this book for demonstration purposes.

E.2.1 Tables and views

Example E-1 is the sample Oracle DDL statements for the tables and views.

Example: E-1 Tables and views Oracle DDL statements

```
CREATE SEQUENCE EMPLOYEE_SEQUENCE
 MINVALUE 1
 MAXVALUE 999999999999999999999999999
 INCREMENT BY 1
 START WITH  2
 CACHE 20 NOCYCLE  NOORDER
/
```

```
CREATE SEQUENCE CUSTOMER_SEQUENCE
 MINVALUE 1
 MAXVALUE 999999999999999999999999999
 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: Dispaly hierarcy 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.2.2 Packages, procedures, and functions

Example E-2 is the sample Oracle DDL statements for packages, procedures, and functions.

Example: E-2 Packages, procedures, and functions Oracle DDL statements

```
--============================================================================
-- 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 successfuly 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_Department
     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;

   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  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 employeed 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;
/
```

E.2.3 Triggers and anonymous blocks

Example E-3 is the sample Oracle DDL statements for triggers and anonymous blocks.

Example: E-3 Triggers and anonymous blocks Oracle DDL statements

```
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 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;
/
```

E.3 DB2 DDL statements

This section lists the sample DB2 DDL statements as a reference for possible code modifications when you enable Oracle database objects for DB2.

E.3.1 Tables and views

Example E-4 is the sample DB2 DDL statements for tables and views. The "@" symbol is used as the statement terminator.

Example: E-4 Tables and views DB2 DDL statements

```
CREATE SEQUENCE EMPLOYEE_SEQUENCE
 MINVALUE 1
 MAXVALUE 999999999999999999999999999
 INCREMENT BY 1
 START WITH  2
 CACHE 20 NOCYCLE  NOORDER
@

CREATE SEQUENCE CUSTOMER_SEQUENCE
 MINVALUE 1
 MAXVALUE 999999999999999999999999999
 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(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)
@

CREATE table  CUSTOMERS
(
   "CUST_ID"          NUMBER(5),
   "CUST_DETAILS_XML" XML ,
   "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 (cust_details_xml)
GENERATE KEYS USING XMLPATTERN '//customer-details/addr/city/text()'
AS SQL VARCHAR(50)
@

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 hierarcy 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 SALES.EMPLOYEE_SEQUENCE
@

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
@

CREATE PUBLIC SYNONYM ORGANIZATION_STRUCTURE FOR SALES.ORGANIZATION_STRUCTURE
@

CREATE PUBLIC SYNONYM HELPER  FOR SALES.HELPER
@
CREATE PUBLIC SYNONYM  ACCOUNT_PACKAGE FOR SALES.ACCOUNT_PACKAGE
@

CREATE PUBLIC SYNONYM  EMPLOYEE_DYNAMIC_QUERY FOR SALES.EMPLOYEE_DYNAMIC_QUERY
@

CREATE PUBLIC SYNONYM  SAVE_ORG_TO_FILE   FOR SALES.SAVE_ORG_TO_FILE
@
```

E.3.2 PL/SQL packages, procedures, and functions

Terminology usage: The following example uses the at sign (@) as the statement terminator to avoid conflicts with semicolons (;) that are used to terminate individual statements within PL/SQL routines.

Example E-5 is the sample DB2 DDL statements for PL/SQL packages, procedures, and functions.

Example: E-5 DB2 DDL statements for PL/SQL packages, procedures, and functions

```
CREATE OR REPLACE  TYPE EMP_INFO_TYPE AS OBJECT (
    EMP_ID     NUMBER(5),
    FIRST_NAME VARCHAR2(20),
    LAST_NAME  VARCHAR2(20),
    BAND       CHAR(1))
@

CREATE OR REPLACE FUNCTION EMP_INFO_TYPE (
    P_EMP_ID     NUMBER(5),
    P_FIRST_NAME VARCHAR2(20),
    P_LAST_NAME  VARCHAR2(20),
    P_BAND       CHAR(1))
RETURN EMP_INFO_TYPE
AS
 RET EMP_INFO_TYPE;
BEGIN
 RET.EMP_ID := P_EMP_ID;
 RET.FIRST_NAME := P_FIRST_NAME;
 RET.LAST_NAME := P_LAST_NAME;
 RET.BAND := P_BAND;
 RETURN RET;
END;
@

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 successfuly 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;


  FUNCTION AVERAGE_BAND_28036 ( 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_Department
       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_28036;

   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
    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_28036(
                                  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  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 employeed 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 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 no. of employees assigned to account
  v_MaxEmployees     NUMBER;  -- Maximum no. 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 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 ;
@

CALL UTL_DIR.CREATE_OR_REPLACE_DIRECTORY('MYDIR','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('Error 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 O.</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">416-555-5555</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('$customer/customer-details/addr/city'
               PASSING cust_details_xml AS "customer") AS VARCHAR(50))
    INTO v_city
    FROM CUSTOMERS
    WHERE cust_id=v_cust_id;
  DBMS_OUTPUT.PUT_LINE('Customers located in the same city: ');
  FOR i IN (SELECT x.cust_name
    FROM sales.CUSTOMERS c, XMLTABLE(
      '$CUST_DETAILS_XML//customer-details[addr/city=$city]'
      PASSING v_city as "city"
      COLUMNS
         cust_name VARCHAR(128) PATH 'name'
   ) x
   Where cust_id<>v_cust_id )
  LOOP
      DBMS_OUTPUT.PUT_LINE(i.cust_name);
 END LOOP;
 EXCEPTION
    WHEN others THEN
       DBMS_OUTPUT.PUT_LINE('Problem inserting XML data in customers table');
END Insert_Customer_in_XML;
@
```

E.3.3 Triggers

Example E-6 is the sample DB2 DDL statements for triggers

Example: E-6 DB2 DDL statements for triggers

```
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 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;
@
```

E.4 Deep nested objects sample

Example E-7 shows the DB2 10.5 deep nested objects feature. This feature handles a data structure whose XML equivalent view looks similar to this example.

Example: E-7 Deep nested objects example

```
<?xml version="1.0" encoding="UTF-8"?>
<customer> <!-- Object, deep nested L0 -->
   <info> <!-- Object, deep nested L1 -->
      <name> <!-- Object L2 -->
         <first_name>John</first_name> <!-- Leaf value -->
         <last_name>S.</last_name>
      </name>
      <birth_date>1963-03-29</birth_date>
   </info>
   <adresses> <!-- Object Array, deep nested L1 -->
      <address> <!-- Object, deep nested L2 -->
         <phones> <!-- Object Array, deep nested L3 -->
            <phone> <!-- Object L4 -->
               <phone_provider>Bell</phone_provider>
               <phone_number>000-111-2222</phone_number>
            </phone>
            <phone>
               <phone_provider>Rogers</phone_provider>
               <phone_number>000-111-2223</phone_number>
            </phone>
         </phones>
         <country>Canada</country>
         <country_div>Ontario</country_div>
         <city>Toronto</city>
         <street>Warden Ave.</street>
         <number>8200</number>
         <code>M2H-2P7</code>
         <building_address> <!-- Object, deep nested L3 -->
            <entry>A1</entry>
            <floor>2</floor>
            <apartment_number>120</apartment_number>
         </building_address>
      </address>
   </adresses>
</customer>
```

This code is organized in a standard Eclipse project, and the samples are driven by few use cases that are backed by their positive path JUnit test cases.

The sections that follow present the content of the files in the project. You can use the actual Eclipse project as a starting point for experimenting with the deep nested objects feature.

E.4.1 Creating a database

Example E-8 shows the create database script file location in the project and its content.

Example: E-8 The create database SQL file and its content

```
file:DB2_10_deep_nested_objects_explorer/src/com/ibm/imte/db2/sample/de
epnested/ddl/database.create.ddl.sql:
create database testdb automatic storage yes pagesize 32 K;
```

E.4.2 Creating schema objects (default schema)

Example E-9 shows the create database script file location in the project and its content.

Example: E-9 The create database script file and its content

```
file:DB2_10_deep_nested_objects_explorer/src/com/ibm/imte/db2/sample/de
epnested/ddl/schema_objects.create.ddl.sql:

CREATE OR REPLACE PACKAGE DNOBJ AS

-- modeling the "name" element of the XML schema
   TYPE  NAME_TYPE IS RECORD
   (
     FIRST_NAMEVARCHAR2(32),
     LAST_NAME VARCHAR2(32)
   );

-- adding the default object constructor
   FUNCTION NAME (
     FIRST_NAMEVARCHAR2(32),
     LAST_NAME VARCHAR2(32)
   ) RETURN NAME_TYPE;

-- modeling the "info" element of the XML schema
   TYPE  INFO_TYPE IS RECORD
   (
     NAME   NAME_TYPE, -- nesting a record type in another record type
     BIRTH_DATEDATE
   );

-- adding the default object constructor
   FUNCTION INFO (
     NAME   NAME_TYPE,
     BIRTH_DATEDATE
   ) RETURN INFO_TYPE;

-- modeling the "phone" element of the XML schema
   TYPE  PHONE_TYPE IS RECORD
   (
     PHONE_PROVIDERVARCHAR2(32),
     PHONE_NUMBERVARCHAR2(32)
   );

-- adding the default object constructor
   FUNCTION PHONE (
     PHONE_PROVIDERVARCHAR2(32),
     PHONE_NUMBERVARCHAR2(32)
   ) RETURN PHONE_TYPE;

-- modeling the "phones" element of the XML schema using an array of
   -- phone records
   TYPE PHONE_ARRAY_TYPE IS TABLE OF PHONE_TYPE INDEX BY INTEGER;

-- adding the default object constructor
   FUNCTION PHONES (
     A1PHONE_TYPE
   ) RETURN PHONE_ARRAY_TYPE;
   FUNCTION PHONES (
     A1PHONE_TYPE,
     A2PHONE_TYPE
   ) RETURN PHONE_ARRAY_TYPE;
   FUNCTION PHONES (
     A1PHONE_TYPE,
     A2PHONE_TYPE,
     A3PHONE_TYPE
   ) RETURN PHONE_ARRAY_TYPE;

-- modeling the "buiding_address" element of the XML schema
   TYPE  BUILDING_ADDRESS_TYPE IS RECORD
   (
     ENTRY     VARCHAR2(8),
     FLOOR     NUMBER(4),
     APARTMENT_NUMBERNUMBER(3)
   );

-- adding the default object constructor
   FUNCTION BUILDING_ADDRESS (
     ENTRY     VARCHAR2(8),
     FLOOR     NUMBER(4),
     APARTMENT_NUMBERNUMBER(3)
   ) RETURN BUILDING_ADDRESS_TYPE;

-- modeling the "address" element of the XML schema
   TYPE  ADDRESS_TYPE IS RECORD
   (
     PHONES PHONE_ARRAY_TYPE,
     COUNTRYVARCHAR2(32),
     COUNTRY_DIVVARCHAR2(32),
     CITY      VARCHAR2(32),
     STREET VARCHAR2(32),
     CODE      VARCHAR2(32),
     BUILDING_ADDRESSBUILDING_ADDRESS_TYPE
   );

-- adding the default object constructor
   FUNCTION ADDRESS (
     PHONES PHONE_ARRAY_TYPE,
     COUNTRYVARCHAR2(32),
     COUNTRY_DIVVARCHAR2(32),
     CITY      VARCHAR2(32),
     STREET VARCHAR2(32),
     CODE      VARCHAR2(32),
     BUILDING_ADDRESSBUILDING_ADDRESS_TYPE
   ) RETURN ADDRESS_TYPE;
```

```
-- modeling the "adresses" element of the XML schema using an array of
-- adress records
   TYPE ADDRESS_ARRAY_TYPE IS TABLE OF ADDRESS_TYPE INDEX BY INTEGER;
-- adding the default object constructor
   FUNCTION ADDRESSES (
     A1ADDRESS_TYPE
   ) RETURN ADDRESS_ARRAY_TYPE;
   FUNCTION ADDRESSES (
     A1ADDRESS_TYPE,
     A2ADDRESS_TYPE
   ) RETURN ADDRESS_ARRAY_TYPE;

-- modeling the "customer" element of the XML schema
   TYPE  CUSTOMER_TYPE IS RECORD
   (
     KEY       INTEGER,
     INFOINFO_TYPE,
     ADDRESSESADDRESS_ARRAY_TYPE
   );

-- adding the default object constructor
   FUNCTION CUSTOMER (
     KEY       INTEGER,
     INFOINFO_TYPE,
     ADDRESSESADDRESS_ARRAY_TYPE
   ) RETURN CUSTOMER_TYPE;

-- provision to return multiple addresses in a single operation
   TYPE CUSTOMER_ARRAY_TYPE IS TABLE OF CUSTOMER_TYPE INDEX BY INTEGER;
-- adding the default object constructor
   FUNCTION CUSTOMERS (
     A1CUSTOMER_TYPE
   ) RETURN CUSTOMER_ARRAY_TYPE;
   FUNCTION CUSTOMERS (
     A1CUSTOMER_TYPE,
     A2CUSTOMER_TYPE
   ) RETURN CUSTOMER_ARRAY_TYPE;

   PROCEDURE
   GENERATE_CUSTOMER_SAMPLE_OBJECT_ARRAY(
     CUSTOMERS        OUT DNOBJ.CUSTOMER_ARRAY_TYPE,
     SEED              IN INTEGER,
     CUSTOMER_COUNT    IN INTEGER,
     ADDRESS_MAX_COUNT IN INTEGER,
     PHONE_MA_COUNT    IN INTEGER
   );

END DEEP_NESTED_OBJECTS_SAMPLE;
/

CREATE OR REPLACE PACKAGE BODY DNOBJ AS

   FUNCTION NAME (
     FIRST_NAMEVARCHAR2(32),
     LAST_NAME VARCHAR2(32)
   ) RETURN NAME_TYPE
   IS
      OBJ NAME_TYPE;
   BEGIN
    OBJ.FIRST_NAME:= FIRST_NAME;
      OBJ.LAST_NAME:= LAST_NAME;
      RETURN OBJ;
   END NAME;

   FUNCTION INFO (
     NAME   NAME_TYPE,
     BIRTH_DATEDATE
   ) RETURN INFO_TYPE
   IS
      OBJ INFO_TYPE;
   BEGIN
    OBJ.NAME:= NAME;
      OBJ.BIRTH_DATE:= BIRTH_DATE;
      RETURN OBJ;
   END INFO;

   FUNCTION PHONE (
     PHONE_PROVIDERVARCHAR2(32),
     PHONE_NUMBERVARCHAR2(32)
   ) RETURN PHONE_TYPE
   IS
      OBJ PHONE_TYPE;
   BEGIN
    OBJ.PHONE_PROVIDER:= PHONE_PROVIDER;
      OBJ.PHONE_NUMBER:= PHONE_NUMBER;
      RETURN OBJ;
   END PHONE;

   FUNCTION PHONES (
     A1PHONE_TYPE
   ) RETURN PHONE_ARRAY_TYPE
   IS
      OBJ PHONE_ARRAY_TYPE;
   BEGIN
      OBJ(1):=A1;
      RETURN OBJ;
   END PHONES;
   FUNCTION PHONES (
     A1PHONE_TYPE,
     A2PHONE_TYPE
   ) RETURN PHONE_ARRAY_TYPE
   IS
      OBJ PHONE_ARRAY_TYPE;
   BEGIN
      OBJ(1):=A1;
      OBJ(2):=A2;
      RETURN OBJ;
   END PHONES;
   FUNCTION PHONES (
     A1PHONE_TYPE,
     A2PHONE_TYPE,
     A3PHONE_TYPE
   ) RETURN PHONE_ARRAY_TYPE
   IS
      OBJ PHONE_ARRAY_TYPE;
   BEGIN
      OBJ(1):=A1;
      OBJ(2):=A2;
      OBJ(3):=A3;
      RETURN OBJ;
   END PHONES;


   FUNCTION BUILDING_ADDRESS (
     ENTRY    VARCHAR2(8),
     P_FLOORNUMBER(4),
     APARTMENT_NUMBERNUMBER(3)
   ) RETURN BUILDING_ADDRESS_TYPE
   IS
      OBJ BUILDING_ADDRESS_TYPE;
   BEGIN
      OBJ.ENTRY:= ENTRY;
      OBJ.FLOOR:= P_FLOOR;
      OBJ.APARTMENT_NUMBER:= APARTMENT_NUMBER;
      RETURN OBJ;
   END BUILDING_ADDRESS;

   FUNCTION ADDRESS (
     PHONES PHONE_ARRAY_TYPE,
     COUNTRYVARCHAR2(32),
     COUNTRY_DIVVARCHAR2(32),
     CITY      VARCHAR2(32),
     STREET VARCHAR2(32),
     CODE      VARCHAR2(32),
     BUILDING_ADDRESSBUILDING_ADDRESS_TYPE
   ) RETURN ADDRESS_TYPE
   IS
      OBJ ADDRESS_TYPE;
   BEGIN
    OBJ.PHONES:= PHONES;
      OBJ.COUNTRY:= COUNTRY;
      OBJ.COUNTRY_DIV:= COUNTRY_DIV;
      OBJ.CITY := CITY;
      OBJ.STREET:= STREET;
      OBJ.CODE := CODE;
      OBJ.BUILDING_ADDRESS:= BUILDING_ADDRESS;
      RETURN OBJ;
   END ADDRESS;

   FUNCTION ADDRESSES (
     A1ADDRESS_TYPE
   ) RETURN ADDRESS_ARRAY_TYPE
   IS
      OBJ ADDRESS_ARRAY_TYPE;
   BEGIN
      OBJ(1):=A1;
      RETURN OBJ;
   END ADDRESSES;
   FUNCTION ADDRESSES (
     A1ADDRESS_TYPE,
     A2ADDRESS_TYPE
   ) RETURN ADDRESS_ARRAY_TYPE
   IS
      OBJ ADDRESS_ARRAY_TYPE;
   BEGIN
      OBJ(1):=A1;
      OBJ(2):=A2;
      RETURN OBJ;
   END ADDRESSES;
```

```
   FUNCTION CUSTOMER (
     KEY       INTEGER,
     INFOINFO_TYPE,
     ADDRESSESADDRESS_ARRAY_TYPE
   ) RETURN CUSTOMER_TYPE
   IS
      OBJ CUSTOMER_TYPE;
   BEGIN
       OBJ.KEY         := KEY;
    OBJ.INFO:= INFO;
      OBJ.ADDRESSES:= ADDRESSES;
      RETURN OBJ;
   END NAME;

   FUNCTION CUSTOMERS (
     A1CUSTOMER_TYPE
   ) RETURN CUSTOMER_ARRAY_TYPE
   IS
      OBJ CUSTOMER_ARRAY_TYPE;
   BEGIN
      OBJ(1):=A1;
      RETURN OBJ;
   END CUSTOMERS;
   FUNCTION CUSTOMERS (
     A1CUSTOMER_TYPE,
     A2CUSTOMER_TYPE
   ) RETURN CUSTOMER_ARRAY_TYPE
   IS
      OBJ CUSTOMER_ARRAY_TYPE;
   BEGIN
      OBJ(1):=A1;
      OBJ(2):=A2;
      RETURN OBJ;
   END CUSTOMERS;

-- simple random generator or nested object tree
 PROCEDURE
   GENERATE_CUSTOMER_SAMPLE_OBJECT_ARRAY(
     CUSTOMERS        OUT CUSTOMER_ARRAY_TYPE,
     SEED              IN INTEGER,
     CUSTOMER_COUNT    IN INTEGER,
     ADDRESS_MAX_COUNT IN INTEGER,
     PHONE_MAX_COUNT   IN INTEGER
   ) AS
```

```
    BEGIN
      if    SEED              = 1
        and CUSTOMER_COUNT    = 1
        and ADDRESS_MAX_COUNT = 1
        and PHONE_MAX_COUNT   = 1
      then
         -- at this time we generate a simple constant deep nsted object
         -- tree to showcase the usagage of the function based factories
         -- to build nested trees which can be conpared with XML, JSON
etc
         CUSTOMERS :=
         CUSTOMERS(
          CUSTOMER(
              0, -- key
             INFO(
               NAME('John','Smith'),
               '1967-02-23'
             ),
             ADDRESSES(
                ADDRESS(
                  PHONES(
                    PHONE(
                      'Rogers',
                      '000-111-2223'
                    )
                  ), -- phone array
                  'Canada', -- country
                  'Ontario', -- country_div
                  'Toronto', -- city
                  'Warden Ave.', -- street
                  'M2H-2P7', -- code
                  BUILDING_ADDRESS(
                    'A1', -- entry
                    3, -- floor
                    120 -- appartment
                  ) -- building_ddress
                ) -- address
             )-- address array
          )
        );
     else
      -- TODO: implement random generator
       customers := customers(cast(null as customer_type));
     end if;
 END;
```

```
END;
/

--
-----------------------------------------------------------------------
--
--    The second case is when standalone types are to be converted
--    The process is the same just that this tim there are no plsql
packages
--
-----------------------------------------------------------------------
--
-- modeling the "name" element of the XML schema
create or replace
   TYPE  NAME_TYPE AS ROW
   (
     FIRST_NAMEVARCHAR2(32),
     LAST_NAME VARCHAR2(32)
   )
/
create or replace
   TYPE NAME_ARRAY_TYPE as NAME_TYPE ARRAY[]
/

-- modeling the "info" element of the XML schema
create or replace
   TYPE  INFO_TYPE AS ROW
   (
     NAME   NAME_TYPE, -- nesting a record type in another record type
     BIRTH_DATEDATE
   )
/
create or replace
   TYPE INFO_ARRAY_TYPE as INFO_TYPE ARRAY[]
/

-- modeling the "phone" element of the XML schema
create or replace
   TYPE  PHONE_TYPE AS ROW
   (
     PHONE_PROVIDERVARCHAR2(32),
     PHONE_NUMBERVARCHAR2(32)
   )
/
```

```
-- modeling the "phones" element of the XML schema using an array of
   -- phone records
create or replace
   TYPE PHONE_ARRAY_TYPE as PHONE_TYPE ARRAY[]
/

-- modeling the "buiding_address" element of the XML schema
create or replace
   TYPE  BUILDING_ADDRESS_TYPE AS ROW
   (
     ENTRY     VARCHAR2(8),
     FLOOR     NUMBER(4),
     APARTMENT_NUMBERNUMBER(3)
   )
/
create or replace
   TYPE BUILDING_ADDRESS_ARRAY_TYPE as BUILDING_ADDRESS_TYPE ARRAY[]
/

-- modeling the "address" element of the XML schema
create or replace
   TYPE  ADDRESS_TYPE AS ROW
   (
     PHONES PHONE_ARRAY_TYPE,
     COUNTRYVARCHAR2(32),
     COUNTRY_DIVVARCHAR2(32),
     CITY      VARCHAR2(32),
     STREET VARCHAR2(32),
     CODE      VARCHAR2(32),
     BUILDING_ADDRESSBUILDING_ADDRESS_TYPE
   )
/
-- modeling the "adresses" element of the XML schema using an array of
-- adress records
create or replace
   TYPE ADDRESS_ARRAY_TYPE AS ADDRESS_TYPE ARRAY[]
/

-- modeling the "customer" element of the XML schema
create or replace
   TYPE  CUSTOMER_TYPE AS ROW
   (
     key       int,
     INFOINFO_TYPE,
     ADDRESSESADDRESS_ARRAY_TYPE
   )
/
-- provision to return multiple addresses in a single operation
create or replace
   TYPE CUSTOMER_ARRAY_TYPE AS CUSTOMER_TYPE ARRAY[]
/

-- create supporting object instance function factories (constructors)
-- all those constructors can be automatically generated for the
existing
-- row types already defined in the database
-- the generator can also include the array types and their constructor
-- function counterparts
create or replace
   FUNCTION NAME (
     FIRST_NAMEVARCHAR2(32),
     LAST_NAME VARCHAR2(32)
   ) RETURNS NAME_TYPE
   LANGUAGE SQL
   BEGIN
       declare OBJ NAME_TYPE;
    set OBJ.FIRST_NAME= FIRST_NAME;
      set OBJ.LAST_NAME= LAST_NAME;
      RETURN OBJ;
   END
/

create or replace
   FUNCTION INFO (
     NAME   NAME_TYPE,
     BIRTH_DATEDATE
   ) RETURNS INFO_TYPE
   LANGUAGE SQL
   BEGIN
      declare OBJ INFO_TYPE;
    set OBJ.NAME= NAME;
      set OBJ.BIRTH_DATE= BIRTH_DATE;
      RETURN OBJ;
   END
/

create or replace
   FUNCTION PHONE (
     PHONE_PROVIDERVARCHAR2(32),
     PHONE_NUMBERVARCHAR2(32)
   ) RETURNS PHONE_TYPE
   LANGUAGE SQL
   BEGIN
      declare OBJ PHONE_TYPE;
    set OBJ.PHONE_PROVIDER= PHONE_PROVIDER;
      set OBJ.PHONE_NUMBER= PHONE_NUMBER;
      RETURN OBJ;
   END
/

create or replace
   FUNCTION PHONES (
     a1PHONE_TYPE
   ) RETURNS PHONE_ARRAY_TYPE
   LANGUAGE SQL
   BEGIN
      declare OBJ PHONE_ARRAY_TYPE;
      set OBJ[1] = a1;
      RETURN OBJ;
   END
/
create or replace
   FUNCTION PHONES (
     a1PHONE_TYPE,
     a2PHONE_TYPE
   ) RETURNS PHONE_ARRAY_TYPE
   LANGUAGE SQL
   BEGIN
      declare OBJ PHONE_ARRAY_TYPE;
      set OBJ[1] = a1;
      set OBJ[2] = a2;
      RETURN OBJ;
   END
/
create or replace
   FUNCTION PHONES (
     a1PHONE_TYPE,
     a2PHONE_TYPE,
     a3PHONE_TYPE
   ) RETURNS PHONE_ARRAY_TYPE
   LANGUAGE SQL
   BEGIN
      declare OBJ PHONE_ARRAY_TYPE;
      set OBJ[1] = a1;
      set OBJ[2] = a2;
      set OBJ[3] = a3;
      RETURN OBJ;
   END
/
create or replace
   FUNCTION BUILDING_ADDRESS (
     ENTRY     VARCHAR2(8),
     P_FLOORNUMBER(4),
     APARTMENT_NUMBERNUMBER(3)
   ) RETURNS BUILDING_ADDRESS_TYPE
   LANGUAGE SQL
   BEGIN
      declare OBJ BUILDING_ADDRESS_TYPE;
      set OBJ.ENTRY = ENTRY;
      set OBJ.FLOOR = P_FLOOR;
      set OBJ.APARTMENT_NUMBER = APARTMENT_NUMBER;
      RETURN OBJ;
   END
/

create or replace
   FUNCTION ADDRESS (
     PHONES PHONE_ARRAY_TYPE,
     COUNTRYVARCHAR2(32),
     COUNTRY_DIVVARCHAR2(32),
     CITY      VARCHAR2(32),
     STREET VARCHAR2(32),
     CODE      VARCHAR2(32),
     BUILDING_ADDRESSBUILDING_ADDRESS_TYPE
   ) RETURNS ADDRESS_TYPE
   LANGUAGE SQL
   BEGIN
      declare OBJ ADDRESS_TYPE;
    set OBJ.PHONES = PHONES;
      set OBJ.COUNTRY = COUNTRY;
      set OBJ.COUNTRY_DIV = COUNTRY_DIV;
      set OBJ.CITY = CITY;
      set OBJ.STREET = STREET;
      set OBJ.CODE = CODE;
      set OBJ.BUILDING_ADDRESS = BUILDING_ADDRESS;
      RETURN OBJ;
   END
/
create or replace
   FUNCTION ADDRESSES (
     a1ADDRESS_TYPE
   ) RETURNS ADDRESS_ARRAY_TYPE
   LANGUAGE SQL
   BEGIN
      declare OBJ ADDRESS_ARRAY_TYPE;
      set OBJ[1] = a1;
      RETURN OBJ;
   END
/
create or replace
   FUNCTION ADDRESSES (
     a1ADDRESS_TYPE,
     a2ADDRESS_TYPE
   ) RETURNS ADDRESS_ARRAY_TYPE
   LANGUAGE SQL
   BEGIN
      declare OBJ ADDRESS_ARRAY_TYPE;
      set OBJ[1] = a1;
      set OBJ[1] = a2;
      RETURN OBJ;
   END
/

create or replace
   FUNCTION CUSTOMER (
     KEY       INT,
     INFOINFO_TYPE,
     ADDRESSESADDRESS_ARRAY_TYPE
   ) RETURNS CUSTOMER_TYPE
   LANGUAGE SQL
   BEGIN
      declare OBJ CUSTOMER_TYPE;
      set OBJ.KEY         = KEY;
    set OBJ.INFO = INFO;
      set OBJ.ADDRESSES= ADDRESSES;
      RETURN OBJ;
   END
/
create or replace
   FUNCTION CUSTOMERS (
     a1CUSTOMER_TYPE
   ) RETURNS CUSTOMER_ARRAY_TYPE
   LANGUAGE SQL
   BEGIN
      declare OBJ CUSTOMER_ARRAY_TYPE;
      set OBJ[1] = a1;
      RETURN OBJ;
   END
/
create or replace
   FUNCTION CUSTOMERS (
     a1CUSTOMER_TYPE,
     a2CUSTOMER_TYPE
   ) RETURNS CUSTOMER_ARRAY_TYPE
   LANGUAGE SQL
   BEGIN
      declare OBJ CUSTOMER_ARRAY_TYPE;
      set OBJ[1] = a1;
      set OBJ[2] = a2;
      RETURN OBJ;
   END
/
create or replace
   FUNCTION CUSTOMERS (
     a1CUSTOMER_TYPE,
     a2CUSTOMER_TYPE,
     a3CUSTOMER_TYPE
   ) RETURNS CUSTOMER_ARRAY_TYPE
   LANGUAGE SQL
   BEGIN
      declare OBJ CUSTOMER_ARRAY_TYPE;
      set OBJ[1] = a1;
      set OBJ[2] = a2;
      set OBJ[3] = a3;
      RETURN OBJ;
   END
/

-- simple random generator or nested object tree
CREATE OR REPLACE
PROCEDURE
   GENERATE_CUSTOMER_SAMPLE_OBJECT_ARRAY(
     CUSTOMERS        OUT CUSTOMER_ARRAY_TYPE,
     SEED              IN INTEGER,
     CUSTOMER_COUNT    IN INTEGER,
     ADDRESS_MAX_COUNT IN INTEGER,
     PHONE_MAX_COUNT   IN INTEGER
   ) AS

   BEGIN
```

```
     if    SEED              = 1
       and CUSTOMER_COUNT    = 1
       and ADDRESS_MAX_COUNT = 1
       and PHONE_MAX_COUNT   = 1
     then
        CUSTOMERS :=
        CUSTOMERS(
         CUSTOMER(
             0,-- key
            INFO(
              NAME('John','Smith'),
              '1967-02-23'
            ),
            ADDRESSES(
               ADDRESS(
                 PHONES(
                   PHONE(
                     'Rogers',
                     '000-111-2223'
                   )
                 ), -- phone array
                 'Canada', -- country
                 'Ontario', -- country_div
                 'Toronto', -- city
                 'Warden Ave.', -- street
                 'M2H-2P7', -- code
                 BUILDING_ADDRESS(
                   'A1', -- entry
                   3, -- floor
                   120 -- appartment
                 ) -- building_ddress
               ) -- address
            )-- address array
         )
       );
    else
     -- TODO: implement random generator
      CUSTOMERS := customers(null);
    end if;
 END;
/
```

E.4.3 JUnit tests (class file excerpt that includes only relevant information for the printed book)

Example E-10 shows the JUnit test case that runs the deep nested objects function at the client side. The test code provides setup and tear down logic and the following test cases:

- testReadNestedObjectPackage
- testReadNestedObjectStandalone

Both tests use stored procedures that generate pseudo-random deep nested structures. These structures exercise object nesting procedures that use PL/SQL stored procedures (testReadNestedObjectPackage) and native SQL PL procedures (testReadNestedObjectStandalone). The test logic calls the stored procedures and then converts the result in a textual format to be easier to read. The test logic uses an expected string constant as a way to validate the test result (string compare). The test case is derived from a TestBase class that encapsulates all the common logic. The test case can be reused by the reader to quickly build his own tests to exercise various scenarios with the database.

Example: E-10 The Java and JDBC JUnit test case file location and content

```
file:DB2_10_deep_nested_objects_explorer/src/com/ibm/imte/db2/sample/de
epnested/ReadNestedObjectTest.java
public class ReadNestedObjectTest extends TestBase {

   @Override
   protected void setUp() throws Exception {
      assertEquals(
            0,

create_test_initial_state("com/ibm/imte/db2/sample/deepnested/ddl/schem
a_objects.create.ddl.sql"));
   }

   @Override
   protected void tearDown() throws Exception {

revert_test_state("com/ibm/imte/db2/sample/deepnested/ddl/schema_object
s.drop.ddl.sql");
      closeConnection();
   }

   public void testReadNestedObjectPackage() throws SQLException,
         ClassNotFoundException {
      Connection con = getConnection();
      CallableStatement cstmt = null;

      try {
         cstmt = con
               .prepareCall("CALL
dnobj.generate_customer_sample_object_array(?,?,?,?,?)");

         cstmt.registerOutParameter(1, java.sql.Types.ARRAY);
         cstmt.setInt(2, 1);// using same seed to keep the test happy
:)
         cstmt.setInt(3, 1);
         cstmt.setInt(4, 1);
         cstmt.setInt(5, 1);

         cstmt.execute();

         // simply dump of the received data tree by walking
         // the tree down for any "A" array or "S" structure
         // entity detected, leaf field discrimination is positional
         String result = sqlObjectToText("customers->",
cstmt.getArray(1));

         // log the current result to be used in case of failure
         logInfo("Result\n" + result);

         // verify by comparing to the expected value
         String expected = "customers->A[0]S[0]=0\n"
               + "customers->A[0]S[1]S[0]S[0]=John\n"
               + "customers->A[0]S[1]S[0]S[1]=Smith\n"
               + "customers->A[0]S[1]S[1]=1967-02-23 00:00:00.0\n"
               + "customers->A[0]S[2]A[0]S[0]A[0]S[0]=Rogers\n"
               + "customers->A[0]S[2]A[0]S[0]A[0]S[1]=000-111-2223\n"
               + "customers->A[0]S[2]A[0]S[1]=Canada\n"
               + "customers->A[0]S[2]A[0]S[2]=Ontario\n"
               + "customers->A[0]S[2]A[0]S[3]=Toronto\n"
               + "customers->A[0]S[2]A[0]S[4]=Warden Ave.\n"
               + "customers->A[0]S[2]A[0]S[5]=M2H-2P7\n"
               + "customers->A[0]S[2]A[0]S[6]S[0]=A1\n"
               + "customers->A[0]S[2]A[0]S[6]S[1]=3\n"
               + "customers->A[0]S[2]A[0]S[6]S[2]=120\n";
         logInfo("Expected\n" + expected);
         assertEquals(expected, result);
      } finally {
         if (cstmt != null) {
            cstmt.close();
         }
      }
   }

   public void testReadNestedObjectStandalone() throws SQLException,
         ClassNotFoundException {
      Connection con = getConnection();
      CallableStatement cstmt = null;
      try {
         cstmt = con
               .prepareCall("CALL
generate_customer_sample_object_array(?,?,?,?,?)");

         cstmt.registerOutParameter(1, java.sql.Types.ARRAY);
         cstmt.setInt(2, 1);// using same seed for random to keep the
test
         // happy :)
         cstmt.setInt(3, 1);
         cstmt.setInt(4, 1);
         cstmt.setInt(5, 1);

         cstmt.execute();

         // simply dump of the received data tree by walking
         // the tree down for any "A" array or "S" structure
         // entity detected, leaf field discrimination is positional
         String result = sqlObjectToText("customers->",
cstmt.getArray(1));

         // log the current result to be used in case of failure
         logInfo("Result\n" + result);

         // verify by comparing to the expected value
         String expected = "customers->A[0]S[0]=0\n"
               + "customers->A[0]S[1]S[0]S[0]=John\n"
               + "customers->A[0]S[1]S[0]S[1]=Smith\n"
               + "customers->A[0]S[1]S[1]=1967-02-23 00:00:00.0\n"
               + "customers->A[0]S[2]A[0]S[0]A[0]S[0]=Rogers\n"
               + "customers->A[0]S[2]A[0]S[0]A[0]S[1]=000-111-2223\n"
               + "customers->A[0]S[2]A[0]S[1]=Canada\n"
               + "customers->A[0]S[2]A[0]S[2]=Ontario\n"
               + "customers->A[0]S[2]A[0]S[3]=Toronto\n"
               + "customers->A[0]S[2]A[0]S[4]=Warden Ave.\n"
               + "customers->A[0]S[2]A[0]S[5]=M2H-2P7\n"
               + "customers->A[0]S[2]A[0]S[6]S[0]=A1\n"
```

```
                + "customers->A[0]S[2]A[0]S[6]S[1]=3\n"
                + "customers->A[0]S[2]A[0]S[6]S[2]=120\n";
            logInfo("Expected\n" + expected);
            assertEquals(expected, result);
         } finally {
            if (cstmt != null) {
               cstmt.close();
            }
         }
      }
   }


file:DB2_10_deep_nested_objects_explorer/src/com/ibm/imte/TestBase.java

public class TestBase extends TestCase {

   public int create_test_initial_state(String id) throws SQLException,
         IOException, ClassNotFoundException {
      // id is the path from where we load all the DDL definitions and
initial
      // table data
      // in order to reproduce a given database state
      return execBatch(getConnection(), id, getLogger());
   }

   public int revert_test_state(String id) throws SQLException,
IOException,
         ClassNotFoundException {
      return execBatch(getConnection(), id, getLogger());
   }

   public String sqlObjectToText(String name, Object obj) {
      ByteArrayOutputStream bos = new ByteArrayOutputStream();
      PrintStream out = new PrintStream(bos);

      if (obj instanceof java.sql.Array) {
         try {
            Object[] a = (Object[]) ((Array) obj).getArray();
            for (int i = 0; i < a.length; ++i) {
               out.print(sqlObjectToText(name + "A[" + i + "]", a[i]));
            }
         } catch (SQLException e) {
            out.println(name + "Array:-exception->" + e.getMessage());
         }
      } else if (obj instanceof java.sql.Struct) {
         Struct a = (Struct) obj;
         try {
            Object[] attrs = a.getAttributes();
            for (int i = 0; i < attrs.length; ++i) {
               out.print(sqlObjectToText(name + "S[" + i + "]",
attrs[i]));
            }
         } catch (SQLException e) {
            out.println(name + "struct:-exception->" + e.getMessage());
         }
      } else {
         out.println(name + "=" + obj);
      }
      out.flush();
      return bos.toString();
   }

   public Logger getLogger(String name, String log_file) {
      Logger logger = Logger.getLogger(name);
      if (log_file != null) {
         try {
            logger.addHandler(new FileHandler(log_file));
         } catch (IOException e) {
         }
         logger.setLevel(Level.ALL);
      }
      return logger;
   }

   boolean trace;
   String database = "testdb";
   String host = "localhost";
   String port = "50000";
   String user = "db2inst1";
   String password = "password";
   String trace_level = "0";
   String trace_file = "jdbc.trace.log";
   String trace_dir = ".";
   String trace_append = "true";

   Properties env;

   Properties getTestEnv() {
      if (env == null) {
         env = new Properties();
         try {
            InputStream fin = new FileInputStream("env.properties");
            env.load(fin);
            fin.close();
         } catch (FileNotFoundException e) {
            logger.info("No env.properties in cwd, using default");
         } catch (IOException e) {
            logger
                  .info("Unable to access env.properties in cwd, using
default");
         }
         if (!env.containsKey("db2.host")) {
            env.setProperty("db2.host", host);
         }
         if (!env.containsKey("db2.port")) {
            env.setProperty("db2.port", port);
         }
         if (!env.containsKey("db2.db")) {
            env.setProperty("db2.db", database);
         }
         if (!env.containsKey("db2.user")) {
            env.setProperty("db2.user", user);
         }
         if (!env.containsKey("db2.password")) {
            env.setProperty("db2.password", password);
         }
         if (!env.containsKey("db2.traceLevel")) {
            env.setProperty("db2.traceLevel", trace_level);
         }
         if (!env.containsKey("db2.traceFile")) {
            env.setProperty("db2.traceFile", trace_file);
         }
         if (!env.containsKey("db2.traceDirectory")) {
            env.setProperty("db2.traceDirectory", trace_dir);
         }
         if (!env.containsKey("db2.traceFileAppend")) {
            env.setProperty("db2.traceFileAppend", trace_append);
         }
      }
      return env;
   }

   private String getConnectionSting() {
      getTestEnv();
      String cstr = "jdbc:db2://"
            + env.getProperty("db2.host")
            + ":"
            + env.getProperty("db2.port")
            + "/"
            + env.getProperty("db2.db")
            + ":retrieveMessagesFromServerOnGetMessage=true;"
            + (env.getProperty("db2.traceLevel").equalsIgnoreCase("0")
? ""
                  : ("traceLevel=" + env.getProperty("db2.traceLevel")
                        + ";traceFile="
                        + env.getProperty("db2.traceFile")
                        + ";traceDirectory="
                        + env.getProperty("db2.traceDirectory")
                        + ";traceFileAppend=" + env
                        .getProperty("db2.traceFileAppend")));

      return cstr;
   }

   private Logger logger;

   public Logger getLogger() {
      if (logger == null) {
         logger = getLogger(this.getClass().getCanonicalName(), this
               .getClass().getCanonicalName()
               + ".log");
      }
      return logger;
   }

   public void logInfo(String msg) {
      if (logger == null) {
         logger = getLogger();
      }
      logger.info(msg);
   }

   public void log(Level level, String msg) {
      if (logger == null) {
         logger = getLogger(this.getClass().getCanonicalName(), this
               .getClass().getCanonicalName()
               + ".log");
      }
```

```
        logger.log(level, msg);
    }

    DatabaseMetaData dbm;
    Connection con;

    protected Connection getConnection() throws SQLException,
          ClassNotFoundException {

        if (con == null) {
            logInfo("Loading DB2 JCC 4 driver");
            Class.forName("com.ibm.db2.jcc.DB2Driver");
            logInfo("Connecting to the DB2 with user:" + user);

            con = DriverManager.getConnection(getConnectionSting(), env
                  .getProperty("db2.user"),
  env.getProperty("db2.password"));
            logInfo("Connected");
            con.setAutoCommit(true);
            // dbm=con.getMetaData();
        }

        return con;
    }

    protected void closeConnection() {
        if (con != null) {
            try {
                con.close();
            } catch (SQLException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
            con = null;
        }
    }

    public boolean canReachDBServer() {
        try {
            getTestEnv();
            Socket sk = new Socket(env.getProperty("db2.host"), Integer
                  .parseInt(env.getProperty("db2.port")));
            sk.close();
            return true;
        } catch (NumberFormatException e) {
         logInfo("Server port number exception:" + port);
      } catch (UnknownHostException e) {
         logInfo("Server address is unknon:" + host);
      } catch (IOException e) {
         logInfo("Server connect IO exception:" + e);
         // e.printStackTrace();
      }
      return false;
   }

   public boolean canConnectToDatabase() {
      try {
         getConnection();
         closeConnection();
         return true;
      } catch (SQLException e) {
         logInfo("Unable to connect to the database host:"
               + getConnectionSting() + " user:"
               + env.getProperty("db2.user") + " error:" +
e.getMessage());
      } catch (ClassNotFoundException e) {
         logInfo("DB2 JDBC driver class not found
(com.ibm.db2.jcc.DB2Driver) chck classpath");
      }
      return false;
   }

   public static void resultSetToCSV(ResultSet rs, PrintStream out)
         throws SQLException {
      if (rs == null) {
         out.println("#null result set");
         return;
      }
      ResultSetMetaData rsm = rs.getMetaData();
      int ncols = rsm.getColumnCount();

      out.print("#");
      for (int i = 0; i < ncols; i++) {
         out.print(rsm.getColumnName(i + 1));
         if (i < (ncols - 1)) {
            out.print(",");
         }
      }
      out.println();
      String value;
      while (rs.next()) {
         for (int i = 0; i < ncols; i++) {
            value = rs.getString(i + 1);
            if (value == null) {
               value = "NULL";
            } else {
               if (value.indexOf(',') >= 0 || value.indexOf('"') >= 0)
{
                  value = '"' + value.replace("\"", "\\\"") + '"';
               }
            }
            out.print(value);
            if (i < (ncols - 1)) {
               out.print(",");
            }
         }
         out.println();
      }
      out.flush();
   }

   public static String resultSetToCSV(ResultSet rs) throws
SQLException {
      ByteArrayOutputStream bos = new ByteArrayOutputStream();
      PrintStream out = new PrintStream(bos);
      resultSetToCSV(rs, out);
      return bos.toString();
   }

   public static List<String> loadStatements(InputStream in, String
delim)
         throws IOException {
      List<String> stmts = new ArrayList<String>();
      BufferedReader rd = new BufferedReader(new
InputStreamReader(in));
      String line, stmt;
      StringBuilder sb = new StringBuilder();

      do {
         line = rd.readLine();
         if (line == null) {
            break;
         }
         if (line.trim().equalsIgnoreCase(delim)) {
            stmt = sb.toString().trim();
            if (stmt.length() > 0) {
               stmts.add(stmt);
               sb = new StringBuilder();
            }
         } else {
            sb.append(line);
            sb.append("\n");
         }
      } while (line != null);
      return stmts;
   }

   public static int execBatch(List<String> stmts, Connection con,
         Logger logger) throws SQLException {
      Statement st = con.createStatement();
      int err_cnt = 0;
      try {
         for (Iterator<String> iterator = stmts.iterator(); iterator
               .hasNext();) {
            String stmt = (String) iterator.next();
            logger.info(stmt);
            try {
               st.execute(stmt);
               logger.info("Success");
            } catch (SQLException e) {
               ++err_cnt;
               logger.info("Exec error:" + e.getMessage());
            }
         }
      } finally {
         con.commit();
      }
      return err_cnt;
   }

   public static int execBatch(Connection con, String resource_id,
         Logger logger) throws SQLException, IOException {
      return execBatch(

loadStatements(Thread.currentThread().getContextClassLoader()
                  .getResourceAsStream(resource_id), "/"), con,
logger);
   }
}
```

E.4.4 Cleaning up

The project provides an optional SQL batch file that selectively removes all the objects that are created by the setup phase. For convenience, a pseudo-test case provides a simple way to run it from inside the Eclipse Env (Example E-11).

Example: E-11 The selective schema cleanup logic

```
file:DB2_10_deep_nested_objects_explorer/src/com/ibm/imte/db2/sample/de
epnested/TestEnvCleanup.java

public class TestEnvCleanup extends TestBase {

   @Override
   protected void tearDown() throws Exception {
      closeConnection();
   }

   public void testEnvCleanup() throws SQLException,
ClassNotFoundException {
      try {

revert_test_state("com/ibm/imte/db2/sample/deepnested/ddl/schema_object
s.drop.ddl.sql");
      } catch (IOException e) {
         logInfo("Drop sql exception:" + e.getMessage());
      }
   }
}
```

F

Additional material

This appendix refers to more material that can be downloaded from the Internet as described in the following sections.

Code disclaimer

IBM does not warrant or represent that the code provided is complete or up-to-date. IBM does not warrant, represent or imply reliability, serviceability, or function of the code. IBM is under no obligation to update content nor provide further support.

All code is provided “as is,” with no warranties or guarantees whatsoever. IBM expressly disclaims to the fullest extent permitted by law all express, implied, statutory, and other warranties, guarantees, or representations, including, without limitation, the warranties of merchantability, fitness for a particular purpose, and non-infringement of proprietary and intellectual property rights. You understand and agree that you use these materials, information, products, software, programs, and services, at your own discretion and risk and that you will be solely responsible for any damages that may result, including loss of data or damage to your computer system.

In no event will IBM be liable to any party for any direct, indirect, incidental, special, exemplary, or consequential damages of any type whatsoever related to or arising from use of the code found herein, without limitation, any lost profits, business interruption, lost savings, loss of programs, or other data, even if IBM is expressly advised of the possibility of such damages. This exclusion and waiver of liability applies to all causes of action, whether based on contract, warranty, tort, or any other legal theories.

Locating the web material

The web material that is 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 website 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:

File name	*Description*
`sg247636testcase.zip`	Compressed test code samples for enablement practice. The compressed file includes the `Oracle_testcase` and `DB2_testcase` files.

System requirements for downloading the web material

The web material requires the following system configuration:

Hard disk space:	0.5 MB minimum
Operating System:	Windows 2000/Linux SUSE or Red Hat
Processor:	Intel 386 or higher
Memory:	16 MB

Downloading and extracting the web material

Create a subdirectory (folder) on your workstation, and extract the contents of the web material compressed file into this folder.

Related publications

The publications that are listed in this section are considered suitable for a more detailed discussion of the topics that are covered in this book.

IBM Redbooks publications

The following IBM Redbooks publication provides more information about the topics in this book:

- *Oracle to DB2 Conversion Guide for Linux, UNIX, and Windows*, SG24-7048

You can search for, view, download, or order this book and other Redbooks, Redpapers, Web Docs, draft and more materials, at the following website:

ibm.com/redbooks

Other publications

These publications are also relevant as further information sources:

- *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 websites are also relevant as further information sources:

DB2

- Database and Information Management home page

  http://www.ibm.com/software/data/
- DB2 Application Development

  http://www.ibm.com/software/data/db2/ad/
- DB2 developerWorks

  http://www.ibm.com/developerworks/db2/
- DB2 Information Center

  http://pic.dhe.ibm.com/infocenter/db2luw/v10r5/index.jsp
- DB2 for Linux

  http://www.ibm.com/software/data/db2/linux/

- DB2 Product Family Library

  http://www.ibm.com/software/data/db2/library/
- DB2 Technical Support

  http://www-947.ibm.com/support/entry/portal/Overview/Software/Information_Management/DB2_for_Linux,_UNIX_and_Windows
- 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/

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

IBM

Redbooks

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 10.5 Oracle Database compatibility features

Use Oracle PL/SQL skills directly with DB2 10.5

This IBM Redbooks publication describes IBM DB2 SQL 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 can help with developing applications that run on both DB2 and Oracle and can help simplify the process of moving from Oracle to DB2.

In addition, IBM now provides tools 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 the IBM Data Studio tool suite.

This Oracle to DB2 migration guide describes new technology, preferred practices for moving to DB2, and common scenarios that can help you as you move from Oracle to DB2. This book is intended for IT architects and developers who are converting from Oracle to DB2.

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

SG24-7736-02 ISBN 0738438758