IBM® DB2® Universal Database™
Porting Guide

Oracle to DB2 UDB for Windows, OS/2 and Unix
Version 7.2

Document Version 3.1

Prepared by:

DB2 Technical Enablement Services
IBM Toronto Lab

May, 2002
Preface

Introduction

Who Should Read This Document?

Architecture

Data Types

SQL Language Elements
Multiple Buffer pools ................................................................. 60
DMS Table spaces ........................................................................ 59
SMS Table spaces ........................................................................ 59
Data Management ......................................................................... 59
Privileges and Authorities ............................................................ 58
Authentication ............................................................................. 58
Command Line Processor ............................................................. 58
Journal .......................................................................................... 58
Script Center .................................................................................. 57
Command Center .......................................................................... 56
Control Center ............................................................................ 55
Database Administration ............................................................... 54
Concurrency and Locks ................................................................. 50
Isolation Levels ........................................................................... 50
Lock Escalation ............................................................................ 52
Deadlocks .................................................................................... 52
Optimistic Locking ....................................................................... 53
How to Improve Concurrency ....................................................... 53
Logging .......................................................................................... 54
Archive Logging ............................................................................ 54
Application Development ............................................................. 39
DB2 SQL Control Statements in Procedures ................................. 39
C / C++ ........................................................................................ 41
Java ............................................................................................. 41
CLI Dynamic SQL ......................................................................... 42
Embedded Static SQL ................................................................... 42
Embedded Dynamic SQL ............................................................. 43
Cursors .......................................................................................... 44
Null Indicators .............................................................................. 45
SQLDA Mappings ......................................................................... 45
SQLCA / ORACA .......................................................................... 47
Error Handling ............................................................................ 47
Referential Integrity ..................................................................... 49
Concurrency Control .................................................................... 50
Database Administration ............................................................... 54
Control Center ............................................................................ 56
Command Center ......................................................................... 57
Script Center ................................................................................. 58
Journal ........................................................................................ 58
Command Line Processor ............................................................. 58
Authentication ............................................................................... 58
Privileges and Authorities ............................................................ 59
Data Management ......................................................................... 59
SMS Table spaces ........................................................................ 59
DMS Table spaces ....................................................................... 60
Multiple Buffer pools .................................................................. 60
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Import</td>
<td>61</td>
</tr>
<tr>
<td>Load</td>
<td>61</td>
</tr>
<tr>
<td>Redirected Restore</td>
<td>62</td>
</tr>
<tr>
<td><strong>Problem Determination</strong></td>
<td>63</td>
</tr>
<tr>
<td>Explain</td>
<td>63</td>
</tr>
<tr>
<td>Visual Explain</td>
<td>64</td>
</tr>
<tr>
<td>Database Snapshot Monitor</td>
<td>65</td>
</tr>
<tr>
<td>Database Event Monitor</td>
<td>66</td>
</tr>
<tr>
<td>Error Conditions</td>
<td>67</td>
</tr>
<tr>
<td>DB2DIAG.LOG</td>
<td>67</td>
</tr>
<tr>
<td>Trace facility</td>
<td>67</td>
</tr>
<tr>
<td>Performance Tuning</td>
<td>68</td>
</tr>
<tr>
<td>Case 1</td>
<td>68</td>
</tr>
<tr>
<td>Case 2</td>
<td>68</td>
</tr>
<tr>
<td>Case 3</td>
<td>68</td>
</tr>
<tr>
<td>Case 4</td>
<td>68</td>
</tr>
<tr>
<td><strong>Conclusion</strong></td>
<td>69</td>
</tr>
<tr>
<td>DB2 Resources</td>
<td>69</td>
</tr>
<tr>
<td>Appendix A - DATE data type</td>
<td>70</td>
</tr>
<tr>
<td>Appendix B - Oracle Identifier Names</td>
<td>72</td>
</tr>
<tr>
<td>Appendix C - ROWID Column / Generate_unique</td>
<td>73</td>
</tr>
<tr>
<td>Appendix D - Sample DB2 SQL Procedure</td>
<td>75</td>
</tr>
<tr>
<td>Appendix E - Sample DB2 User-Defined Functions (UDFs)</td>
<td>76</td>
</tr>
<tr>
<td>Appendix F - Sample DB2 Java Stored Procedure</td>
<td>78</td>
</tr>
<tr>
<td>Appendix G - Sample DB2 Static SQL Program</td>
<td>82</td>
</tr>
<tr>
<td>Appendix H - Sample DB2 Dynamic Embedded Application</td>
<td>84</td>
</tr>
<tr>
<td>Appendix I - Mapping of Oracle8 OCI to DB2 CLI</td>
<td>88</td>
</tr>
</tbody>
</table>
Preface

Notices, Trademarks, Service Marks and Disclaimers
This document contains proprietary information of IBM and is protected by copyright law. The information contained in this publication does not include any product warranties, and any statements provided in this document should not be interpreted as such.

The following terms are trademarks or registered trademarks of the IBM Corporation in the United States and/or other countries: IBM, DB2 Universal Database, AIX, OS/2.

Windows and Windows-based trademarks and logos are trademarks or registered trademarks of Microsoft Corporation.

Unix and Unix-based trademarks and logos are trademarks or registered trademarks of The Open Group.

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

Summary of Changes
Version 3.1: Document revised, and updated to DB2® Version 7.2, by Bill Wilkins. Thanks to Amyris Rada, Tony Lee, Burt Vialpando, Paul Yip, Zamil Janmohamed, and Raul Chong for their contributions, and to Kristin Dalzell for editing. The primary changes were in the following sections:

“Mapping of Oracle Terminology to DB2 UDB” on page 13
“DATE and TIME” on page 18
“Named Savepoints and Compound SQL (Static and Dynamic)” on page 21
“Sequence Objects” on page 22
“ROWNUM” on page 24
“ORDER BY in Subselect” on page 28
“Functions” on page 28
“Sampling Rows from a Table (or Result Set)” on page 29
“Triggers” on page 30
“Views and Materialized Views” on page 32
“Joins” on page 32
“Nulls, NVL, and Concatenation” on page 35
“Set Operators: UNION, INTERSECT, and MINUS” on page 36
“Application Development” on page 39
“DB2 SQL Control Statements in Procedures” on page 40
“Isolation Levels” on page 50
Appendix E - Sample DB2 User-Defined Functions (UDFs) on page 76

Version 3.0: Document substantially revised and updated to DB2 V7.1 by Bill Wilkins. Thanks to Rob Newman for numerous comments and suggestions.

Acknowledgements
Versions 1 and 2 of this document were authored by Ming Wu. She thanks the following people “for the invaluable advice and guidance provided in the production of this document”:

Dennis Bockus, Editor, IBM Canada
Debra Eaton, Reviewer and Contributor of many sample programs
Robert Newman, Contributor of many sample programs
Grant Hutchison, Reviewer
The members of the IBM Software Migration Project Office (SMPO)
Introduction

A common requirement for database applications is to have them run on multiple database servers. The objective of this document is to help IBM® business partners to port Oracle applications to DB2 Universal Database (DB2 UDB). In this document, references to “DB2 UDB” or “DB2” should be interpreted as “DB2 Universal Database for Windows®, OS/2®, and UNIX®.” Unless otherwise specified, “Oracle” implies Oracle8i.

This is a working document; new topics will be added as they become relevant in porting situations. Meanwhile, depending on the type of application, not all topics discussed in this paper will be relevant to a particular situation.

Version 2 of this document was an update to reflect the changes in DB2 UDB V6.1. Also, some of the topics and samples were updated to reflect new information gathered after Version 1.

Version 3 is updated to reflect the changes in DB2 UDB V7.

Please direct any comments regarding this document to Bill Wilkins, wilkins@ca.ibm.com.

Who Should Read This Document?

This document is written for application developers and database administrators who want to convert their applications from Oracle to DB2 UDB. We assume you are currently working with Oracle8, Oracle8i, or Oracle9i, and are porting applications to DB2 UDB Enterprise Edition (EE) V7.2. We do not cover the details of the DB2 UDB Enterprise-Extended Edition (EEE). However, applications developed for Enterprise Edition are portable to the EEE Edition with few modifications. This document covers topics that are relevant primarily to SQL developers porting applications from Oracle to DB2 UDB. It also covers topics that are relevant to database administrators.

We assume that readers are familiar with the concept of a relational database management system (RDBMS) and with Oracle SQL and PL/SQL. We also assume that readers have easy access to DB2 UDB V7.2 documentation. Refer to that documentation for detailed information about SQL statement syntax.

This document takes a top-down approach, starting with the architectures of Oracle and DB2 UDB and ending with specific details and conversion examples. Although both Oracle and DB2 UDB are essentially platform independent, we point out the differences between Windows operating systems and UNIX operating systems when those differences matter.
Architecture

In this section, we give a very high-level overview of the architectures of Oracle and DB2 UDB. Although it is not essential to know the system architecture to develop applications for the database products, it is useful in the context of this document as we try to point out the differences in the system behavior of the two databases.

Client-Server View

DB2 UDB and Oracle both separate the client code and the server code into different address spaces. The application code runs in the client process while the server code runs in separate processes. The client processes can run on the same machine as the database server or a different one, accessing the database server through a programming interface. Both DB2 UDB and Oracle support dynamic and embedded static SQL interfaces.

Oracle PL/SQL is a programming extension of the standard SQL. All Oracle stored procedures, user-defined functions, and triggers must be written in PL/SQL. DB2 UDB has a similar facility called the SQL Control Statements. In addition, users can use any programming language supported by the precompiler, such as C, C++, Java™, COBOL, REXX and Fortran, to construct DB2 UDB stored procedures.

SQL*Plus provides Oracle command line access to the database server. DB2 has a similar tool called the Command Line Processor (CLP). However, the CLP does not have all of the features as SQL*Plus (such as spooling). Instead, we recommend that you use the Control Center Script Center to perform some of the scripting and scheduling activities.

Server Architecture

DB2 UDB implements a dedicated process architecture. For Windows NT®, these processes are implemented with threads. For each active connection, there is a user process to execute the application and the DB2 UDB client code. The user process can reside on a client machine or a server machine. For each active connection, there is a dedicated server process (called the \textit{db2agent}) serving that connection. There may be more associated agent processes (\textit{db2agnt}) if the system is configured for SMP parallelism. The server process runs on the database server. Besides the dedicated server processes, each database server has a suite of background processes, each of which has a dedicated purpose, such as logging or writing to the database files.

DB2 UDB Enterprise-Extended Edition (EEE) extends the architecture to environments with the multiple nodes. For each node in a Nodegroup, a db2agent (or more than one agent for an environment with SMP nodes) is created for the connection. While the application communicates only with the coordinating node, DB2 UDB EEE harnesses the resources on all of the nodes, totally transparent to the application. However, this document does not discuss DB2 UDB EEE in detail.
Oracle implements an architecture similar to DB2 UDB’s, called *dedicated server architecture*. The applications run in the application process and, like DB2 UDB, Oracle has a dedicated server process for each active connection to the database, as well as a number of background server processes performing specific tasks such as logging, or writing to the database files.

Oracle also supports *multi threaded server architecture*, which implements a pooled-agents architecture with a number of server processes serving the applications. The requests from applications are placed on a request queue and are dispatched to available server processes. The multi-threaded server can lead to blocking situations if there are insufficient server processes configured for the system. The only way to resolve the blocking is by manual intervention from the Database Administrator. DB2 UDB does not have a similar pool process model.

The typical configuration for Oracle is the dedicated server architecture.
**Metadata**

Metadata provides the roadmap to interpret the information stored in the database. In Oracle, metadata is stored in the Data Dictionary; in DB2 UDB, metadata is stored in the System Catalog. The preferred way to reference metadata in DB2 UDB is through the catalog views, such as SYSCAT.TABLES, not the underlying tables. The structure and names of these views are much different from those of the Oracle metadata tables, so you must be familiar with the views to map the system information from Oracle to DB2 UDB.

**Physical Database**

DB2 UDB and Oracle share the concept of the physical storage model. This section maps the terminology between the two databases.

Both databases store their data in table spaces. Oracle table spaces are made up of data files, while DB2 table spaces are made up of containers. Conceptually, data files and containers are similar. Oracle does not have default table spaces when a database is created. DB2 UDB creates three table spaces when a database is created: system table space (SYSCATSPACE), temp table space (TEMPSPACE1) and user table space (USERSPACE1).

**Default Database Configuration**

DB2 UDB containers can be defined as system-managed space (SMS) or database-managed space (DMS). By default, when a database is created, SMS is assumed for all containers. Oracle does not have the concept of SMS. Data files in Oracle resemble DB2 UDB DMS containers.

SMS-based table spaces are simple to manage. Space is allocated automatically as required through requests to the operating system. However, there may be more overhead incurred by the operating system. DMS-based table spaces are managed by the database and containers must be explicitly added to the table spaces. DMS requires more database administration overhead, but it is easier to fine-tune performance. In a later section, we show you the best method for mapping Oracle table spaces to DB2 UDB table spaces.

Oracle data is stored in data blocks. The size of the data block is defined when a database is created (from 2K to 64K). An *extent* is the specific number of contiguous data blocks obtained in a single allocation. *Segments* are made up of extents to store a particular type of information. Examples include data segment, index segment, and rollback segment. A segment grows one extent at a time. Data files contain one or more segments.
DB2 UDB data is stored in pages. A table space is defined to have a single page size (4K, 8K, 16K or 32K). In DB2 UDB, an extent is an allocation of space consisting of multiple pages, within a container of a table space, to a single database object. If there is more than one container in a table space, the data is striped across the containers one extent at a time. Objects are made up of pages that store similar information. Examples include table objects and index objects.

Unlike Oracle data files, DB2 SMS containers do not grow one extent at a time. For DMS table spaces, the containers are preallocated. Once the first page of an extent is used in a container, the entire extent is occupied. For SMS table spaces, the space is allocated one page at a time, but this can be changed to one extent at a time by running the `db2empfa` command (which causes the change to apply to all SMS table spaces in a database).

DB2 UDB SMS data storage is page-based storage. DB2 SMS does not require contiguous sets of data blocks to store data, as Oracle does. Hence, there is no fragmentation problem in DB2 UDB SMS table spaces.

**Default DB2 Universal Database Layout**

![Default DB2 Universal Database Layout](image1)

**Default Oracle Database Layout**

![Default Oracle Database Layout](image2)
Memory Model

Both database servers use shared memory areas to store critical information to communicate between server components and applications. However, their organization is quite different. This section gives a quick overview on how each is organized.

Oracle has a System Global Area (SGA) that stores all the shared information for an instance. The contents are used to communicate between server processes. Examples include statement cache, redo log buffers, and data buffer cache.

The shared memory areas used to share information between an application and the Oracle Database Server are called the Program Global Area (PGA) and the User Global Area (UGA). The PGA and UGA contain information such as user context and row-cache-cursors buffers.

DB2 UDB stores similar information but uses different shared memory areas to store different types of information:

- **Database Manager Shared Memory Set** stores all the relevant information for a particular instance, such as lists of all active connections and security information.

- **Database Shared Memory Set** stores information relevant to a particular database, such as package caches, log buffers, and buffer pools. (DB2 refers to the data buffer cache as the buffer pool.)

- **Application Shared Memory Set** stores information that is shared between DB2 and a particular application, primarily rows of data being passed to or from the database.

- **Agent Private Memory Set** stores information that is used by DB2 to service a particular application, such as sort heaps, cursor information, and session contexts.

The Database Manager Shared Memory Set and Database Shared Memory Set can be loosely mapped to the Oracle SGA. The Application Shared Memory Set and Agent Private Memory Set can be loosely mapped to the Oracle PGA and UGA.

In DB2 the **data buffer cache or buffer pools** are used to buffer data in memory to reduce the number of I/O operations to the physical database. Database performance can be dramatically improved if the requested data is buffered in memory. The size of the data buffer cache in Oracle is set by parameter DB_BLOCK_BUFFERS, which specifies the number of blocks in the data buffer cache. Each block is the size of the data block specified in DB_BLOCK_SIZE.

DB2 supports multiple buffer pools and you can assign multiple table spaces to a particular buffer pool. A default buffer pool (IBMDEFAULTBP) is created when a database is first created. New buffer pools are created using the DDL statement CREATE BUFFERPOOL. Use the ALTER BUFFERPOOL statement to change the size of a buffer pool. For more information on DB2 buffer pools, see “Multiple Buffer pools” on page 60.

The hit-ratio is the percentage of database accesses that do not require a physical I/O. Raising the ratio can significantly improve database performance, so tuning the buffer pools for each table space can be very important. One example is to put the indexes for a particular table in a separate table space and assign a dedicated buffer pool to it. The above process can significantly improve database performance in cases where most table access is through indexes.
Buffer Pool Overview

Database Recovery Logs

All RDBMS’s require that changes to the database be logged in order to perform database recovery. The implementations for DB2 UDB and Oracle are, however, quite different. Oracle uses redo logs and rollback segments for database recovery. *Redo logs* record the transaction changes and *rollback segments* are used to store the previous version of the data in a table while the table is being updated or mutated.

DB2 UDB implements *write-ahead logging*, in which the changed data is always written to the log files before the change is committed. DB2 UDB logs changes in its log files, including the old version of the data. It does not have rollback segments. Due to the differences in the database recovery implementations, the two databases have noticeable differences in controlling the concurrent access of data. These differences are discussed in more detail in “Concurrency Control” on page 50.
Mapping of Oracle Terminology to DB2 UDB

The following table gives readers who are familiar with Oracle a quick mapping to DB2 UDB terminology:

<table>
<thead>
<tr>
<th>DB2 UDB</th>
<th>Oracle</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB2 UDB EE</td>
<td>Oracle EE</td>
<td>Enterprise product.</td>
</tr>
<tr>
<td>DB2 UDB EEE</td>
<td>Oracle Parallel</td>
<td>Support node partitioning.</td>
</tr>
<tr>
<td>DB2 Connect</td>
<td>Oracle Gateway</td>
<td>DRDA access to hosts.</td>
</tr>
<tr>
<td>SQL Control Statements</td>
<td>PL/SQL</td>
<td>Programming language extension to SQL. DB2 UDB stored procedures can be programmed in SQL Control Statements (subset of PSM standard), Java, C, C++, COBOL, Fortran, OLE, and REXX. DB2 functions can be programmed in Java, C, C++, OLE, or SQL control statements.</td>
</tr>
<tr>
<td>DB2 CLP</td>
<td>SQL*Plus</td>
<td>Command line interface to the server.</td>
</tr>
<tr>
<td>Instance</td>
<td>Instance</td>
<td>Processes and shared memory. In DB2 it also includes a permanent directory structure: an instance is usually created at install time (or can be later) and must exist before a database can be created. A DB2 instance is also known as the database manager (DBM).</td>
</tr>
<tr>
<td>Database</td>
<td>Database</td>
<td>Physical structure containing data. In Oracle, multiple instances can use the same database, and an instance can connect to one and only one database. In DB2, multiple databases can be created and used concurrently in the same instance.</td>
</tr>
<tr>
<td>DBM and database configuration files, etc.</td>
<td>Control files and .ora files</td>
<td>In Oracle, files that name the locations of files making up the database and provide configuration values. In DB2, each instance (DBM) and database has its own set of configuration parameters stored in a binary file; there are also other internal files and directories: none is manually edited.</td>
</tr>
<tr>
<td>Federated System</td>
<td>Database Link</td>
<td>In Oracle, an object that describes a path from one database to another. In DB2 a federated system is used. One database is chosen as the federated database and within it wrappers, servers, nicknames, and other optional objects are created to define how to access the other databases (including Oracle databases) and objects in them. Once an application is connected to the federated database it can access all authorized objects in the federated system.</td>
</tr>
<tr>
<td>Table spaces</td>
<td>Table spaces</td>
<td>Contains actual database data.</td>
</tr>
<tr>
<td>Containers</td>
<td>Data files</td>
<td>Entities inside the table spaces.</td>
</tr>
<tr>
<td>Objects</td>
<td>Segments</td>
<td>Entities inside the containers/data files.</td>
</tr>
<tr>
<td>Extents</td>
<td>Extents</td>
<td>Entities inside the objects/segments.</td>
</tr>
<tr>
<td>Pages</td>
<td>Data blocks</td>
<td>Smallest storage entity in the storage model.</td>
</tr>
</tbody>
</table>
A logical grouping of PL/SQL blocks that can be invoked by other PL/SQL applications.

A precompiled access plan for an embedded static SQL application stored in the server.

A shared memory area to store user-specific data passed between application process and the database server.

A logical grouping of PL/SQL blocks that can be invoked by other PL/SQL applications.

<table>
<thead>
<tr>
<th>N/A</th>
<th>Clusters</th>
<th>Data structure that allows related data to be stored together on disk; can be table or hash clusters. The closest facility to this in DB2 is a clustering index, which causes rows inserted into a table to be placed physically close to the rows for which the key values of this index are in the same range.</th>
</tr>
</thead>
<tbody>
<tr>
<td>System catalog</td>
<td>Data dictionary</td>
<td>Metadata of the database.</td>
</tr>
<tr>
<td>SMS</td>
<td>N/A</td>
<td>System-managed table space.</td>
</tr>
<tr>
<td>DMS</td>
<td>Data files</td>
<td>Database-managed table space.</td>
</tr>
<tr>
<td>Buffer pools</td>
<td>Data cache</td>
<td>Buffers data in the table spaces to reduce disk I/O.</td>
</tr>
<tr>
<td>Package cache</td>
<td>Statement cache</td>
<td>Caches prepared dynamic SQL statements.</td>
</tr>
<tr>
<td>Log files</td>
<td>Redo logs</td>
<td>Recovery logs.</td>
</tr>
<tr>
<td>N/A</td>
<td>Rollback segments</td>
<td>Store the old version of data for a mutating table. In DB2 the old version of an updated row is stored in the log file along with the new version.</td>
</tr>
<tr>
<td>Database manager and database shared memory</td>
<td>SGA</td>
<td>Shared memory area(s) for the database server. In Oracle there is one, while in DB2 there is one at the database manager (instance) level and one for each active database.</td>
</tr>
<tr>
<td>Agent / application shared memory</td>
<td>UGA</td>
<td>Shared memory area to store user-specific data passed between application process and the database server.</td>
</tr>
<tr>
<td>Package</td>
<td>N/A</td>
<td>A precompiled access plan for an embedded static SQL application stored in the server.</td>
</tr>
<tr>
<td>N/A</td>
<td>Package</td>
<td>A logical grouping of PL/SQL blocks that can be invoked by other PL/SQL applications.</td>
</tr>
</tbody>
</table>
## Data Types

The following table provides a complete list of DB2 data types, their C, C++ and Java data type mappings, their sqllen and sqltype values (from the SQLDA), and a quick description of each. Note that DB2 UDB has multiple types for DATE and multiple types for NUMBER. For more information, see the SQL Reference, the Application Development Guide, and this file that is supplied on DB2 clients: sqlib\include\sql.h

<table>
<thead>
<tr>
<th>Concept</th>
<th>SQL Data Types / Declaration</th>
<th>C/C++ Data Types / Declaration</th>
<th>Java Data Types</th>
<th>sqllen</th>
<th>sqltype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>SMALLINT age;</td>
<td>short age = 32; short int year;</td>
<td>short</td>
<td>2</td>
<td>500/501</td>
<td>16-bit signed integer&lt;br&gt;Range between (-32,768 and 32,767)&lt;br&gt;Precision of 5 digits</td>
</tr>
<tr>
<td></td>
<td>INTEGER salary; INT salary;</td>
<td>long salary; long int deptno;</td>
<td>int</td>
<td>4</td>
<td>496/497</td>
<td>32-bit signed integer&lt;br&gt;Range between (-2,147,483,648 and 2,147,483,647)&lt;br&gt;Precision of 10 digits</td>
</tr>
<tr>
<td></td>
<td>BIGINT serial_num;</td>
<td>long long serial; __int64 serial; sqlint64 serial</td>
<td>long</td>
<td>8</td>
<td>492/493</td>
<td>64-bit signed integer</td>
</tr>
<tr>
<td>Floating point</td>
<td>REAL bonus; FLOAT(n);</td>
<td>float bonus;</td>
<td>float</td>
<td>4</td>
<td>480/481</td>
<td>Single precision floating point&lt;br&gt;32-bit approximation of a real number&lt;br&gt;FLOAT(n) can be synonym for REAL if 0 &lt; n &lt; 25</td>
</tr>
<tr>
<td></td>
<td>DOUBLE wage; DOUBLE PRECISION wage;</td>
<td>double wage;</td>
<td>double</td>
<td>8</td>
<td>480/481</td>
<td>Double precision floating point&lt;br&gt;64-bit approximation of a real number&lt;br&gt;Range in (0, -1.79769E+308 to -2.225E-307, 2.225E-307 to 1.79769E+308)&lt;br&gt;FLOAT(n) can be synonym for DOUBLE if 24 &lt; n &lt; 54</td>
</tr>
<tr>
<td>Decimal</td>
<td>DECIMAL(p,s) price; DEC(p,s) price; NUMERIC (p,s) price; NUM (p,s) price;</td>
<td>double price;</td>
<td>java.math.BigDecimal</td>
<td>(p/2)+1</td>
<td>484/485</td>
<td>Packed decimal&lt;br&gt;No exact equivalent for SQL decimal type - use C double data type&lt;br&gt;If precision /scale not specified, default is (5,0)&lt;br&gt;Max precision is 31 digits, and max range between (-10<strong>31 + 1 ... 10</strong>31 -1)&lt;br&gt;Consider using char / decimal functions to manipulate packed decimal fields as char data</td>
</tr>
<tr>
<td>Date/Time</td>
<td>DATE dt; char dt[11]; struct { short len; char data[10]; } dt;</td>
<td>java.sql.Date</td>
<td>10</td>
<td>384/385</td>
<td>Null-terminated character form (11 characters) or varchar struct form (10 characters); struct can be divided as desired to obtain the individual fields&lt;br&gt;Example: 11/02/2000&lt;br&gt;Stored internally as a packed string of 4 bytes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TIME tm; char tm[9]; struct { short len; char data[8]; } tm;</td>
<td>java.sql.Time</td>
<td>8</td>
<td>388/389</td>
<td>Null-terminated character form (9 characters) or varchar struct form (8 characters); struct can be divided as desired to obtain the individual fields&lt;br&gt;Example: 19:21:39&lt;br&gt;Stored internally as a packed string of 3 bytes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TIMESTAMP ts; char ts[27]; struct { short len; char data[26]; } ts;</td>
<td>java.sql.Timestamp</td>
<td>26</td>
<td>392/393</td>
<td>Null-terminated character form (27 characters) or varchar struct form (26 characters); struct can be divided as desired to obtain the individual fields&lt;br&gt;Example: 2000-12-25-01.02.03.000000&lt;br&gt;Stored internally as a packed string of 10 bytes</td>
<td></td>
</tr>
<tr>
<td>Character</td>
<td>Description</td>
<td>JDBC 1.22:</td>
<td>JDBC 2.0:</td>
<td>Precompiled with WCHARTYPE NOCONVERT option</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
<td>------------</td>
<td>------------</td>
<td>------------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHAR sex;</td>
<td>Fixed-length character string consisting of n bytes</td>
<td>String</td>
<td>java.sql.Clob</td>
<td>NOCONVERT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHAR(5)</td>
<td>Use char[n+1] where 1 &lt;= n &lt;= 254</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHAR zip;</td>
<td>If length not specified, defaults to 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VARCHAR(40) address;</td>
<td>Null-terminated variable length character string</td>
<td>String</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Use char[n+1] where 1 &lt;= n &lt;= 32672</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Default SQL type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VARCHAR(40) address;</td>
<td>Non null-terminated varying character string with 2-byte string length indicator</td>
<td>String</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Use char[n] in struct form where 1 &lt;= n &lt;= 32672</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Default SQL type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LONG VARCHAR</td>
<td>Non null-terminated varying character string with 2-byte string length indicator</td>
<td>String</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Use char[n] in struct form where 32673 &lt;= n &lt;= 32700</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLOB(n)</td>
<td>Non null-terminated varying binary string with 4-byte string length indicator</td>
<td>String</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Use char[n] in struct form where 1 &lt;= n &lt;= 2147483647</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLOB locator variable</td>
<td>Identifies CLOB entities residing on the server</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLOB file reference variable</td>
<td>Descriptor for file containing CLOB data</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BLOB(n)</td>
<td>Non null-terminated varying binary string with 4-byte string length indicator</td>
<td>String</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Use char[n] in struct form where 1 &lt;= n &lt;= 2147483647</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BLOB locator variable</td>
<td>Identifies BLOB entities on the server</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BLOB file reference variable</td>
<td>Descriptor for the file containing BLOB data</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Double-Byte</td>
<td>sql type is blob(1m)</td>
<td>String</td>
<td>java.sql.Blob</td>
<td>NOCONVERT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GRAPHIC(1)</td>
<td>For a fixed-length graphic string of length integer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GRAPHIC(n)</td>
<td>which may range from 1 to 127. If the length specification is omitted, a length of 1 is assumed.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Precompiled with WCHARTYPE NOCONVERT option</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GRAPHIC(n)</td>
<td>sqldbchar dbyte;</td>
<td>String</td>
<td>java.sql.Clob</td>
<td>NOCONVERT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sqldbchar graphic1[n+1];</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>wchar_t graphic2[100];</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VARGRAPHIC(n)</td>
<td>For a varying-length graphic string of maximum length integer, which may range from 1 to 16336. Precompiled with WCHARTYPE NOCONVERT option. Null terminated variable-length</td>
<td>String</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Null terminated variable-length</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LONG VARGRAPHIC(n)</td>
<td>For a varying-length graphic string with a maximum length of 16350 and a 2-byte string length indicator 16337 &lt;= n &lt;= 16350 Precompiled with WCHARTYPE NOCONVERT option</td>
<td>String</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Oracle to DB2 V7.2 Porting Guide

Page 16 of 89
### DBCLOB(n)

- sql type is `dbclb(1m)`
- `tokyo_phone_dir`;

- JDBC 1.22: String
- JDBC 2.0: `java.sql.Clob`

- Precompiled with WCHARTYPE NOCONVERT option

- For non-null-terminated varying double-byte character large object string of the specified maximum length in double-byte characters.

- 4 bytes string length indicator

- Use `dbclb(n)` where `1<=n <= 1073741823` double-byte characters.

- The length of a `DATALINK` column is 200 bytes

### DBCLOB locator variable

- sql type is `dbclb_locator`
- `tokyo_phone_loc`;

- Identifies DBCLOB entities residing on the server

- Precompiled with WCHARTYPE NOCONVERT option

### DBCLOB file reference variable

- sql type is `dbclb_file`
- `tokyo_phone_ref`;

- Descriptor for file containing DBCLOB data

- Precompiled with WCHARTYPE NOCONVERT option

---

### Mapping Oracle Data Types to DB2 UDB Data Types

The following table summarizes the mapping from the Oracle data types to corresponding DB2 data types. The mapping is one to many and depends on the actual usage of the data.

<table>
<thead>
<tr>
<th>Oracle Data Type</th>
<th>Notes</th>
<th>DB2 Data Type</th>
<th>Notes</th>
</tr>
</thead>
</table>
| DATE             |       | DATE TIME TIMESTAMP | If only MM/DD/YYYY is required, use DATE  
|                  |       |               | If only HH:MM:SS is required, use TIME  
|                  |       |               | If both date and time are required (MM/DD/YYYY-HH:MM:SS:000000), use TIMESTAMP  
|                  |       |               | Use Oracle TO_CHAR() function to format a DATE for subsequent DB2 load. Note that the Oracle default DATE format is DD-MON-YY |
| VARCHAR2(n)      | n <=4000 | VARCHAR(n) | n <= 32672 |
| LONG             | n <= 2GB | VARCHAR (n) LONG VARCHAR (n) CLOB(n) | If n <= 32672 bytes, use VARCHAR  
|                  |         |               | If 32672 < n <= 32700 bytes, use LONG VARCHAR or CLOB  
|                  |         |               | If 32672 < n <= 2 GB, use CLOB |
| RAW(n)           | n <= 255 | CHAR(n) FOR BIT DATA VARCHAR(n) FOR BIT DATA BLOB(n) | If n <= 254, use CHAR(n) FOR BIT DATA  
|                  |         |               | If n <= 32672, use VARCHAR(n) FOR BIT DATA  
|                  |         |               | If n<= 2 GB, use BLOB(n) |
| LONG RAW         | n <= 2 GB | VARCHAR(n) FOR BIT DATA LONG VARCHAR FOR BIT DATA BLOB(n) | If n <= 32672 bytes, use VARCHAR(n) FOR BIT DATA  
|                  |         |               | If 32672 < n <= 32700 bytes, use LONG VARCHAR FOR BIT DATA  
|                  |         |               | If n <= 2 GB, use BLOB(n) |
| BLOB             | n<= 4 GB | BLOB(n) | If n <= 2GB use BLOB(n) |
| CLOB             | n<= 4GB | CLOB(n) | If n <= 2GB use CLOB(n) |
| NCLOB            | n<=4GB | DBCLOB(n) | If n <= 2GB use DBCLOB(n/2) |
| NUMBER           | <4GB   | SMALLINT INTEGER BIGINT DECIMAL(p,s) DOUBLE / FLOAT(n) / REAL | If Oracle decl is NUMBER(p) or NUMBER(p,0), use SMALLINT, if 1 <= p <= 4;  
|                  |         |               | INTEGER, if 5 <= p <= 9;  
|                  |         |               | BIGINT, if 10 <= p <= 18.  
|                  |         |               | If Oracle decl is NUMBER(p,s), use DECIMAL(p,s) (s > 0)  
|                  |         |               | If Oracle decl is NUMBER, use DOUBLE / FLOAT(n) / REAL |
DATE and TIME

Oracle data type DATE indicates year, month, day, hour, minute, and second. It does not correspond to the data type DATE of DB2 UDB because the DATE data type in DB2 UDB contains only the year, month, and day. Data type TIME contains only the HH:MM:SS information. Data type TIMESTAMP contains information from the year through to the seconds and microseconds.

TIMESTAMP gives the most complete set of timing information. However, since it also takes the most storage space, only use it if you need the full resolution it provides, or think you may need it in the future. Code fragments in “Appendix A - DATE data type” on page 70 illustrate the differences in date handling between Oracle and DB2.

By default, Oracle expects input dates to be in form DD-MON-YY, and presents output dates in that form. The default date format can be overridden with the NLS_DATE_FORMAT parameter in the INIT.ORA or SPFILE file, or with ALTER SESSION for an individual session.

DB2 does not have default input formats for dates or times. If table T1 has a single DATE column and T2 has a single TIME column, each of these inserts three rows successfully:

\[
\begin{align*}
\text{insert into T1 values ('1991-10-27'), ('10/27/1991'), ('27.10.1991')}; \\
\text{insert into T2 values ('13.30.05'), ('1:30 PM'), ('13:30:05')};
\end{align*}
\]

Because DB2 does not accept the Oracle default date format, dates inserted into DB2 must first be mapped to an acceptable format using a formatting function; for example, TO_CHAR(OracleDate,"MM/DD/YYYY").

DB2 default output date and time formats are determined by the country code of the application, but can be overridden with the DATETIME parameter of the Bind and Prep commands.

There is only one timestamp format allowed in DB2, namely YYYY-MM-DD-HH.MM.SS.MMMMMM (where MMMMMM is microseconds). For more information on date and time formats in DB2, see the Administration Guide index entry “date formats”.

Oracle built-in functions, such as ADD_MONTHS, NEXT_MONTH, and NEXT_DAY, can in some cases be translated directly to equivalent or similar DB2 built-in functions. DB2 provides a suite of durations and datetime arithmetic capability so, for example, “ADD_MONTHS(mydate, 26)” can be mapped to “mydate + 26 MONTHS”. When an Oracle function does not have a DB2 equivalent, or the calling code must be unchanged, a DB2 user-defined function can be created with the same name as the Oracle function. For an example of this, see “Appendix E - Sample DB2 User-Defined Functions (UDFs)” on page 76 for an implemention of the Oracle NEXT_DAY function. For more information on DB2 functions in general, see “Functions” on page 28.

VARCHAR2

Many Oracle applications use VARCHAR2 for very small character strings. In these circumstances, it is better to port it to the fixed-length DB2 datatype CHAR(N), as it is more efficient and takes less storage than VARCHAR. In DB2 UDB, VARCHAR(N) uses n+4 bytes of storage and CHAR(N) uses only N bytes of storage. CHAR should always be used for columns of ten bytes or fewer, and for longer columns that are relatively full of non-blank data.

NUMBER

The Oracle data type NUMBER can be mapped to many DB2 types. The type of mapping depends on whether the NUMBER is used to store an integer (NUMBER(p), or NUMBER(p,0)), or a number with a fixed decimal point (NUMBER(p,s), s > 0), or a floating-point number (NUMBER). Another consideration is the space usage. Each DB2 type requires a different amount of space: SMALLINT uses 2 bytes, INTEGER uses 4 bytes, and BIGINT uses 8 bytes. The space usage for Oracle type NUMBER depends on the parameter used in the declaration. NUMBER, with the default precision of 38 significant digits, uses 20 bytes of storage. Mapping NUMBER to SMALLINT, for example, can save you 18 bytes per column.
DECIMAL

An Oracle NUMBER with non-zero scale (decimal places) should be mapped to DB2 data type DECIMAL. DECIMAL is stored as packed decimal in DB2, with four bits used per decimal digit, plus four bits for the sign, so a DECIMAL column with precision p takes \((p/2 + 1)\) bytes. Decimal processing in DB2 is less efficient than integer processing, so one of the integer data types should be used where possible, particularly when the Oracle NUMBER has a scale of zero (no decimal places).

Here is an example of how a decimal value can be inserted into and retrieved from the database using the CLI interface. The C program could use a double (SQL_C_DOUBLE) or another numeric variable type to store the value, but the default CLI data type for DECIMAL is char (SQL_C_CHAR), and char is used in the example.

```c
SQLCHAR *dec_input = (SQLCHAR *) "-0001234.56780000"; /* input for DEC(15,8) col */
SQLINTEGER ind = 0;                                  /* indicator variable */
SQLCHAR dec_output[18];                              /* output from DEC(15,8) col */

/* bind char variable to parameter marker used to provide the value for updating a decimal column. The length of 18 (2nd-last parameter) includes 15 for precision plus 1 each for sign, decimal point, and null terminator */
rc = SQLBindParameter(hstmt,1, SQL_PARAM_INPUT, SQL_C_CHAR,SQL_DECIMAL, 15, 8,
                      dec_input, 18,&ind);

/* bind char variable to decimal output from Select */
rc = SQLBindCol(hstmt, 1,SQL_C_CHAR, dec_output,18,&ind);
```

RAW

To simulate the Oracle RAW and LONG RAW data types, DB2 provides the FOR BIT DATA clause for the VARCHAR, LONG VARCHAR, and CLOB data types. In addition, DB2 also provides the BLOB data type to store up to 2 GB of binary data. Note that the hextoraw() and rawtohex() functions are not provided in DB2, but it is possible to create a distinct user-defined type (UDT) by using DB2 functions such as hex(), blob() and cast().

Oracle8 extends the LONG type to BLOB and CLOB which can be mapped directly to the BLOB and CLOB data types in DB2 UDB.

DB2 Maximums

DB2 V7.2 supports 4KB, 8KB, 16KB, and 32KB page sizes. Some limits are determined based on the page size. For a full list of limits in DB2, see “Appendix A. SQL Limits” in the SQL Reference.

<table>
<thead>
<tr>
<th>Table Limits</th>
<th>4K page</th>
<th>8K page</th>
<th>16K page</th>
<th>32K page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum length of a row including all overhead</td>
<td>4,005</td>
<td>8,101</td>
<td>16,293</td>
<td>32,677</td>
</tr>
<tr>
<td>Most columns in a table</td>
<td>500</td>
<td>1,012</td>
<td>1,012</td>
<td>1,012</td>
</tr>
<tr>
<td>Maximum size of a table (per partition)</td>
<td>64GB</td>
<td>128GB</td>
<td>256GB</td>
<td>512GB</td>
</tr>
<tr>
<td>Maximum size of index (per partition)</td>
<td>64GB</td>
<td>128GB</td>
<td>256GB</td>
<td>512GB</td>
</tr>
<tr>
<td>Maximum size of a DMS table space</td>
<td>64GB</td>
<td>128GB</td>
<td>256GB</td>
<td>512GB</td>
</tr>
<tr>
<td>Most elements in a select list</td>
<td>500</td>
<td>1,012</td>
<td>1,012</td>
<td>1,012</td>
</tr>
<tr>
<td>Maximum index key length</td>
<td>1,024</td>
<td>1,024</td>
<td>1,024</td>
<td>1,024</td>
</tr>
<tr>
<td>Most columns in an index key</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
</tbody>
</table>
### Database Manager Limits

<table>
<thead>
<tr>
<th>Identifier Length Limits (in bytes)</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Longest SQL statement length, in bytes</td>
<td>65,535</td>
</tr>
<tr>
<td>Longest authorization name (can only be single-byte characters)</td>
<td>30</td>
</tr>
<tr>
<td>Longest constraint name</td>
<td>18</td>
</tr>
<tr>
<td>Longest correlation name</td>
<td>128</td>
</tr>
<tr>
<td>Longest condition name</td>
<td>64</td>
</tr>
<tr>
<td>Longest cursor name</td>
<td>18</td>
</tr>
<tr>
<td>Longest external program name</td>
<td>8</td>
</tr>
<tr>
<td>Longest host identifier</td>
<td>255</td>
</tr>
<tr>
<td>Longest schema name</td>
<td>30</td>
</tr>
<tr>
<td>Longest server (database alias) name</td>
<td>8</td>
</tr>
<tr>
<td>Longest statement name</td>
<td>18</td>
</tr>
<tr>
<td>Longest unqualified column name</td>
<td>30</td>
</tr>
<tr>
<td>Longest unqualified package name</td>
<td>8</td>
</tr>
<tr>
<td>Longest unqualified table name, view name, stored procedure name, nickname, or alias</td>
<td>128</td>
</tr>
<tr>
<td>Longest unqualified user-defined type, user-defined function, buffer pool, table space, nodegroup, trigger or index name</td>
<td>18</td>
</tr>
</tbody>
</table>

### Numerical Limits

<table>
<thead>
<tr>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smallest INTEGER value</td>
</tr>
<tr>
<td>Largest INTEGER value</td>
</tr>
<tr>
<td>Smallest BIGINT value</td>
</tr>
<tr>
<td>Largest BIGINT value</td>
</tr>
<tr>
<td>Smallest SMALLINT value</td>
</tr>
<tr>
<td>Largest SMALLINT value</td>
</tr>
<tr>
<td>Largest decimal precision</td>
</tr>
<tr>
<td>Largest / Smallest DOUBLE value</td>
</tr>
<tr>
<td>Smallest positive / negative DOUBLE value</td>
</tr>
<tr>
<td>Largest / Smallest REAL value</td>
</tr>
<tr>
<td>Smallest positive / negative REAL value</td>
</tr>
</tbody>
</table>

### Length of DB2 Identifier Names

Some identifier names in DB2 UDB have a maximum length of 18 bytes in length, including indexes, constraints, and triggers. However, identifier names in Oracle can be up to 30 characters in length. The conversion of the data definitions from Oracle to DB2 UDB thus requires that some Oracle identifier names exceeding 18 characters be compacted to fall within the 18-character limit of DB2 UDB. “Appendix B - Oracle Identifier Names” on page 72 contains a script to extract Oracle identifier names that are longer than 18 characters that need to be modified when ported to DB2.
SQL Language Elements

This section describes the SQL differences between DB2 UDB and Oracle. For the most part, applications can be migrated by direct mapping of functionalities between the two databases. However, the topics discussed here may require more planning and time for redesign to achieve the same or similar results.

Synonyms

In Oracle, a *synonym* can be created as an alternative name for an object, including a table, view, sequence, procedure, stored function, package, snapshot, or another synonym. DB2 supports the same concept, but with a few differences:

- The alternative name is called an *alias*. However, CREATE SYNONYM is recognized as identical to CREATE ALIAS.
- An alias can only be created on a table, view, or another alias.
- The PUBLIC keyword is not supported (all aliases in DB2 are public).

Named Savepoints and Compound SQL (Static and Dynamic)

Oracle *savepoints* are logical points in a transaction to which the application can choose to roll back. Support for named (or “external”) savepoints has been added to DB2 UDB in V7.1 and is almost identical to that in Oracle.

In addition to savepoints, DB2 UDB includes the concept of compound SQL. A static compound SQL statement is a group of SQL statements that is wrapped between the BEGIN COMPOUND and END COMPOUND statements. By default, DB2 UDB treats them as atomic and rolls back all or commits all of them together. The grouping of statements can reduce network traffic and reduce response times. Some Oracle transactions with named savepoints can be mapped to DB2 UDB transactions with compound SQL statements.

One of the major advantages of compound SQL, and a feature not related to the topic of savepoints, is the performance gain. The entire compound SQL is transported to the database server in one network crossing, which can significantly improve the elapsed time of short transactions.

Starting in DB2 V7.2, dynamic compound SQL is supported. This provides performance benefits similar to those of static compound statements, including network traffic avoidance and the single compilation of statements for multiple executions. Furthermore, dynamic compound statements can be used as the bodies of SQL functions, methods, and triggers, as well as providing logic capability within standalone statements.

Dynamic compound SQL statements can contain the following constructs:

<table>
<thead>
<tr>
<th>Control Statements</th>
<th>SQL statements</th>
<th>Other features</th>
</tr>
</thead>
<tbody>
<tr>
<td>FOR statement</td>
<td>Fullselect</td>
<td>Variable declarations</td>
</tr>
<tr>
<td>GET DIAGNOSTICS</td>
<td>Search UPDATE</td>
<td>Condition declarations</td>
</tr>
<tr>
<td>IF statement</td>
<td>Search DELETE</td>
<td></td>
</tr>
<tr>
<td>ITERATE statement</td>
<td>INSERT</td>
<td></td>
</tr>
<tr>
<td>LEAVE statement</td>
<td>SET variable statement</td>
<td></td>
</tr>
<tr>
<td>SIGNAL statement</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WHILE statement</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For more information on dynamic compound SQL statements, refer to the topic “Compound Statement (Dynamic)” in the DB2 V7.2 Release Notes, and see “DB2 SQL Control Statements in Procedures” on page 40 for some examples of their usage. For the exact definition of fullselect, refer to the SQL Reference, but it is essentially either a Select or Values statement; the latter allows a built-in or user-defined function to be invoked.
The following scenario shows how usage of Oracle savepoints can be mapped to savepoints or compound SQL:

<table>
<thead>
<tr>
<th>Oracle Savepoint</th>
<th>DB2 Savepoint</th>
<th>DB2 Compount SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXEC INSERT ...</td>
<td>EXEC SQL INSERT ...</td>
<td>EXEC SQL INSERT ...</td>
</tr>
<tr>
<td>EXEC DELETE ...</td>
<td>EXEC SQL DELETE ...</td>
<td>EXEC SQL DELETE ...</td>
</tr>
<tr>
<td>SAVEPOINT A;</td>
<td>EXEC SQL SAVEPOINT A ON ROLLBACK RETAIN CURSORS;</td>
<td>EXEC SQL BEGIN COMPOUND ATOMIC</td>
</tr>
</tbody>
</table>
| EXEC UPDATE...   | IF (SQLCODE != 0) THEN ROLLBACK TO SAVEPOINT A; ELSE EXEC SQL INSERT ... | UPDATE ...
| IF SQL%NOTFOUND THEN | INSERT ... | COMMIT, END COMPOUND; |
| COMMIT;          | EXEC SQL COMMIT; | |

**Sequence Objects**

An Oracle sequence object generates an ordered unique number. Typically a sequence object is used to obtain the next unique value for a primary key on an insert. For example, a sequence object can be used to generate the next employee number or the next invoice number. Starting in V7.2, DB2 supports sequence objects, and currently the documentation on them can be found in the V7.2 Release Notes. (As of V7.1, DB2 also supports identity columns, which are similar to sequence objects but of more interest to those porting to DB2 from SQL Server or Sybase. Identity columns are not discussed further in this document.)

The syntax of the CREATE SEQUENCE statement is identical in Oracle and DB2, but there are some differences in how sequences are used. After the following discussion of those differences is an Oracle example and the corresponding DB2 implementation.

**Notes on Sequences and Oracle-DB2 Differences:**

In both Oracle and DB2, a reference to the NEXTVAL of a sequence increments the sequence and returns that new value per row. Multiple references to NEXTVAL in the same statement all return the same value (per row).

Oracle provides CURRVAL as a way of obtaining the most recently generated sequence value, without causing a new value to be generated; so, NEXTVAL and CURRVAL return the same value when referenced in the same Select list. In DB2, PREVVAL returns the sequence value from the previous successful NEXTVAL reference; so, NEXTVAL and CURRVAL do not return the same value in the same Select list. To deal with these differences when converting Oracle statements to DB2:

- If CURRVAL is alone in the statement (no NEXTVAL), replace it directly with PREVVAL.
- If the statement has both CURRVAL and NEXTVAL, replace each CURRVAL with NEXTVAL.
- Leave references to NEXTVAL unchanged.

References to NEXTVAL and CURRVAL in Oracle have the form “sequence_name.NEXTVAL” (or CURRVAL). The DB2 equivalent is “NEXTVAL FOR sequence_name” (or CURRVAL FOR sequence_name).

DB2 sequence values can be cached for improved performance. A log record is written every time the cache is exhausted, so to maximize performance in a high-volume insert scenario, raise the cache size significantly beyond the default of 20.

The syntax to drop a sequence in DB2 is DROP SEQUENCE <sequence name> RESTRICT. (The RESTRICT keyword is not used in Oracle.)

Sequences and identity columns are not currently supported on DB2 EEE.
The DB2 Load utility (see "Load" on page 61) does not provide the ability to generate sequence values as rows are loaded. Instead, the sequence values must be pre-generated in the input flat file or pipe.

**Oracle Example:**

```sql
create table dept
(deptno smallint not null,
 deptname varchar(36) not null,
 mgrno char(6),
 admrdept smallint not null,
 location char(30));

create sequence dept_seq
  start with 500 increment by 1 cache 20;

insert into dept values (dept_seq.nextval, 'SALES', 'SMITH', 50, 'Downtown');
insert into dept values (dept_seq.nextval, 'MARKETING', 'WONG', 100, 'Midtown');
insert into dept values (dept_seq.nextval, 'ACCOUNTING', 'FISHER', 150, 'Uptown');

select * from dept order by deptno;

DEPTNO DEPTNAME   MGRNO  ADMRDEPT LOCATION
------ ------ -------- ------------
 500    SALES    SMITH    50 Downtown
 501   MARKETING WONG     100 Midtown
 502 ACCOUNTING FISHER   150 Uptown

select dept_seq.nextval, dept_seq.nextval, dept_seq.currval from DEPT;

NEXTVAL    NEXTVAL    CURRVAL
---------- ---------- ----------
 503        503        503
 504        504        504
 505        505        505
```

**DB2 Equivalent:**

(The Oracle Create Table and Create Sequence statements work on DB2 without modification.)

```sql
insert into dept values (nextval for dept_seq, 'SALES', 'SMITH', 50, 'Downtown');
insert into dept values (nextval for dept_seq, 'MARKETING', 'WONG', 100, 'Midtown');
insert into dept values (nextval for dept_seq, 'ACCOUNTING', 'FISHER', 150, 'Uptown');

select dept_seq.nextval, dept_seq.nextval, dept_seq.currval from DEPT;

select 1, 2, 3 from DEPT;

1 2 3
--- --- ---
 503 503 502
 504 504 502
 505 505 502
```

(In a CLP environment, to get the same output headings as for Oracle, rewrite the query as:

```sql
select 1, 2, 3 from DEPT;
```

For the sake of completeness, here are alternatives to sequence objects (not that you should use these if you were using sequences in Oracle). Alternatives 3 and 4 can experience unacceptable lock contention in high-volume insert scenarios.)
1. Use an identity column.

2. Use the GENERATE_UNIQUE built-in function to obtain unique, increasing numbers (in the form of CHAR(13) FOR BIT DATA).

3. Create a table (for example, next_number) with a row to record the next number in the sequence for each table. Before inserting a row, retrieve the next number from the next_number table, increment it by 1, and update the next_number row.

4. Use an Insert trigger to select the maximum value of the sequence column and increment it by one before the insert occurs.

**Optimizer Hints**

The Oracle optimizer used to be rule-based. The new cost-based Oracle optimizer is now available. Oracle applications can use hints to force the optimizer to pick a particular access plan. DB2 UDB's optimizer is cost-based and does not support hints, and in fact very rarely requires any database administrator (DBA) action to obtain the best access plan. The DB2 UDB optimizer relies on detailed statistics to determine the best access plan, so it is essential to keep the statistics up-to-date by running RUNSTATS frequently.

If a table grows and shrinks in size so quickly that it is not feasible to update the statistics to continually reflect the current size, you can use a feature added in DB2 UDB V6.1, VOLATILE tables. When a query is issued against a volatile table, the optimizer is essentially forced to use an index scan instead of choosing the access plan based on the statistics stored in the System Catalog. For example, consider a table that is empty at the beginning of the day and grows by thousands of rows each hour. If the statistics are updated only at the start of the day, the optimizer always thinks that the table is empty and table scans when searching for a row. While this is fine at the start of the day, it could prove disastrous as the day goes on. Executing "ALTER TABLE <name> VOLATILE" can prevent such a performance degradation. Another solution is to update the statistics when the table is at its largest size (at the end of the day, for example).

Another way to influence the DB2 optimizer is to change the optimization class. The default optimization class is 5, under which DB2 utilizes an average number of techniques in determining the optimal access plan. By increasing the class number to as high as 9, you instruct the optimizer to use more techniques, which requires more memory and can take significantly longer, but this sometimes produces a better plan and reduces query execution time. Conversely, by decreasing the class number to as low as 0, you instruct the optimizer to use fewer techniques, which requires less memory and can take significantly less time, but may result in a worse plan and longer query execution time. You can change the optimization class during precompile or bind, or at run-time for dynamic SQL. For example:

```
db2 prep  filename.sqc QUERYOPT 1
db2 bind  filename.bnd QUERYOPT 7
EXEC SQL EXECUTE IMMEDIATE :hostvar , where hostvar is set to "set current query optimization 3"
```

**ROWNUM**

The Oracle ROWNUM pseudo-column is a way to number rows in a result set, for identifying them or limiting their number. DB2 UDB does not support ROWNUM, but the equivalent functionality can be obtained through the ROW_NUMBER function or the FETCH FIRST n ROWS ONLY clause. Alternatively, the application can use counters to keep track of the row number, or in CLI, use bookmarks to record and set positions in the result set.

Example 1: Number the rows in a result set.

Oracle query: `Select RowNum, empno, salary from employee`

DB2 equivalent: `Select row_number( ) over( ), empno, salary from employee`

Note that Oracle’s RowNum assigns numbers to rows as they are accessed; that is, before any ordering.
DB2's ROW_NUMBER assigns numbers as they are returned to the application; that is, after any ordering.

Example 2: Limit the size of a result set to 10 rows.

Oracle query:  
Select empno, salary from employee where RowNum < 11

DB2 equivalent:  
Select empno, salary from employee fetch first 10 rows only

FETCH FIRST N ROWS ONLY has the added benefit that N rows will be passed in one network crossing (subject to the size of the client-server request block, per the RQRIOBLK configuration parameter), and buffered on the client side.

Here is another example, showing how ROW_NUMBER with the OVER clause can be used to number rows ordered according to some criterion, and FETCH FIRST N ROWS ONLY to limit the size of the answer set. This query lists the top five earners from the EMPLOYEE table (from the DB2 sample database):

```
T
SELECT EMPNO, LASTNAME, FIRSTNME, SALARY+BONUS AS TOTAL_SALARY,  
ROW_NUMBER() OVER (ORDER BY SALARY+BONUS DESC) AS RANK_SALARY  
FROM EMPLOYEE  
FETCH FIRST 5 ROWS ONLY;
```

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>LASTNAME</th>
<th>FIRSTNME</th>
<th>TOTAL_SALARY</th>
<th>RANK_SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>000010</td>
<td>HAAS</td>
<td>CHRISTINE</td>
<td>53750.00</td>
<td>1</td>
</tr>
<tr>
<td>000110</td>
<td>LUCCHESI</td>
<td>VINCENZO</td>
<td>47400.00</td>
<td>2</td>
</tr>
<tr>
<td>000020</td>
<td>THOMPSON</td>
<td>MICHAEL</td>
<td>42050.00</td>
<td>3</td>
</tr>
<tr>
<td>000050</td>
<td>GEYER</td>
<td>JOHN</td>
<td>40975.00</td>
<td>4</td>
</tr>
<tr>
<td>000030</td>
<td>KWAN</td>
<td>SALLY</td>
<td>39050.00</td>
<td>5</td>
</tr>
</tbody>
</table>

If it is necessary to do an update based on row numbers, a view can be created, based on a different but equivalent form of the previous query, and updated:

```
CREATE VIEW TOP_5_EARNERS AS  
SELECT EMPNO, LASTNAME, FIRSTNME, SALARY, BONUS, TOTAL_SALARY, RANK_SALARY  
FROM (SELECT EMPNO, LASTNAME, FIRSTNME, SALARY, BONUS,  
SALARY+BONUS AS TOTAL_SALARY,  
ROW_NUMBER() OVER (ORDER BY SALARY+BONUS DESC) AS RANK_SALARY  
FROM EMPLOYEE) AS RANKED_EMPLOYEE  
WHERE RANK_SALARY < 6;
```

```
UPDATE TOP_5_EARNERS SET BONUS = BONUS * 1.05;
```

DB2 UDB has another clause, OPTIMIZE FOR N ROWS, that can be used in a SELECT statement. OPTIMIZE FOR N ROWS does not limit the result set. Instead, it may change the access plan to one that returns the first N rows quicker than if the clause had not been specified, which allows the user to deal with those N rows in parallel with the server assembling the rest of the answer set. An example of an access plan change caused by this clause is where the optimizer chooses an index scan instead of a table scan followed by a sort because with the latter access plan no rows could be returned to the application until the sort is finished (or partially finished). OPTIMIZE FOR N ROWS also causes N rows at a time to be returned to the client, with buffering at the client machine. The application can request more than N rows, but in this case the time to retrieve the answer set may be longer than if OPTIMIZE FOR N ROWS had not been specified.

**ROWID**

The Oracle ROWID pseudo-column is a means of identifying or locating a row. It is stored in the index to point to a row in the table. DB2 does not support the ROWID datatype. If the ROWID is used to identify the rows in a particular table, the primary key of the table may be used instead. The primary key values can be generated using a sequence object, as described in "Sequence Objects" on page 22. However, if an application requires a unique identifier for a row across the entire database, an alternative would be to use the GENERATE_UNIQUE built-in function and store the unique data in a new column in the table. It would generate a 13-byte bit data string (CHAR(13) FOR BIT DATA).
that is unique across the system. The GENERATE_UNIQUE function is persistent across database reorganization and migrations. The following is an example of how it can be used in an Insert statement:

```
INSERT INTO WORKLOG VALUES (GENERATE_UNIQUE(), ?, ?, ?)
```

In “Appendix C - ROWID Column / Generate_unique” on page 73 there is sample code to migrate from ROWID to GENERATE_UNIQUE and a trigger. Note that if the column value is to be generated in a trigger, the column cannot be the primary key of the table, because a primary key column must be defined as NOT NULL, and NOT NULL columns must be given values in the VALUES clause of the Insert or have a default value, both of which conflict with a trigger being used to set the column value.

**NO WAIT clause**

The Oracle select-for-update statement allows a NO WAIT clause to prevent the application from being blocked if it cannot obtain the locks required. DB2 UDB does not support the NO WAIT clause because all read access (except for uncommitted-read transactions) in DB2 UDB results in the acquisition of share locks on the selected rows.

The NO WAIT clause can be simulated in DB2 UDB by setting the database configuration parameter LOCKTIMEOUT to 0. LOCKTIMEOUT is the number of seconds that a lock waits permitted to exist before the application is notified. The default is -1, which means “wait forever”. By setting LOCKTIMEOUT to 0, DB2 returns immediately to the application with an SQLCODE -911 and reason code 68 (sqlstate 40001) if the lock cannot be obtained. Note that LOCKTIMEOUT affects all applications connected to the database.

```
DB2 UPDATE DB CONFIGURATION FOR <DBNAME> USING LOCKTIMEOUT 0
```

**TRUNCATE TABLE**

In Oracle, the application can remove all rows from a table by using the TRUNCATE TABLE statement. TRUNCATE TABLE also reclaims the storage used by the table. TRUNCATE TABLE is a DDL command and it cannot be rolled back.

DB2 UDB does not support TRUNCATE TABLE. One solution is to delete all rows in the table with the DELETE statement, but except for very small tables, this solution is too slow to be practical. There are two viable solutions in DB2, both of which can be used to empty a table “instantaneously”:

1. Create the table with the NOT LOGGED INITIALLY attribute (which has no effect until the attribute is activated). To empty the table, issue an ALTER TABLE <name> ACTIVATE NOT LOGGED INITIALLY WITH EMPTY TABLE. This causes the table to be emptied, but is not a recoverable operation; if the table contents prior to the emptying must be recovered, a RESTORE of the table space or database must be performed.

2. The application can simulate TRUNCATE TABLE by importing an empty file to replace the table content. As for approach (1), this reclaims the space used by the table, leaving information associated with the table (such as an index or authorizations) intact. The following is a sample of a UNIX script to empty the table via Import:

```
touch empty.del                                       % Create an empty file
db2 import from empty.del of del replace into <table_name> % Import the empty file
```

**Temporary Tables**

DB2 UDB supports declared temporary tables. Such a table is declared by an application, but unlike a permanent table, is only visible to the application connection in which it is created. Because of the limited scope of a temporary table, DB2 can take the following shortcuts, which result in improved concurrency and performance compared to the use of permanent tables:

No table or row locking is performed.

Changes are not logged.
System catalog tables are not accessed to create or otherwise access a temp table, other than some minor read-only checking; that is, the catalog entries made when creating or dropping a permanent table are avoided.

DB2 declared temporary tables are created using the DECLARE GLOBAL TEMPORARY TABLE statement. See the SQL Reference for details. It is a straightforward exercise to convert Oracle temporary tables to DB2. Here are a few points regarding less obvious details of the DB2 implementation:

A user temporary table space must be created before temporary tables can be declared.

Unless the ON COMMIT PRESERVE ROWS clause is coded in the DECLARE GLOBAL TEMPORARY TABLE statement, the rows in a temporary table are deleted whenever a commit occurs and there is no WITH HOLD cursor open on the temporary table.

References to a temporary table must be explicitly or implicitly qualified by the schema name SESSION.

Here are some alternatives to declared temporary tables:

A common table expression (CTE) is similar to an in-line view in a statement. It can be used in query and insert statements. Temporary tables can in some cases be easily mapped into CTEs as illustrated in the following example. PAYLEVEL and PAYBYED are CTE’s that are subsequently used in the actual SELECT statement. A CTE is only persistent for the duration of the statement.

WITH
  PAYLEVEL AS
    (SELECT EMPNO, YEAR(HIREDATE) AS HIREYEAR, EDLEVEL,
     SALARY+BONUS+COMM AS TOTAL_PAY
     FROM EMPLOYEE
     WHERE EDLEVEL > 16
    ),

  PAYBYED (EDUC_LEVEL, YEAR_OF_HIRE, AVG_TOTAL_PAY) AS
    (SELECT EDLEVEL, HIREYEAR, AVG(TOTAL_PAY)
     FROM PAYLEVEL
     GROUP BY EDLEVEL, HIREYEAR
    )

SELECT EMPNO, EDLEVEL, YEAR_OF_HIRE, TOTAL_PAY, AVG_TOTAL_PAY
FROM PAYLEVEL, PAYBYED
WHERE EDLEVEL=EDUC_LEVEL
  AND HIREYEAR = YEAR_OF_HIRE
  AND TOTAL_PAY < AVG_TOTAL_PAY;

In some cases, particularly where the transaction rate is low, it may be appropriate to simulate Oracle temporary tables with permanent tables in DB2 UDB, using the NOT LOGGED INITIALLY clause on the CREATE TABLE STATEMENT, and the ACTIVATE NOT LOGGED INITIALLY clause on the ALTER TABLE STATEMENT. This approach causes DB2 UDB to not log changes to the table that occur in the same unit of work as the ALTER, improving the efficiency of the changes.
ORDER BY in Subselect

DB2 UDB does not support the ORDER BY clause in a subselect. Queries that use this construct need to be rewritten. The following is a simple example of an Oracle query, using ORDER BY in a subselect with ROWNUM, followed by a DB2 equivalent:

```sql
SELECT *
FROM DEPARTMENT
WHERE MGRNO = (SELECT EMPNO
                FROM EMPLOYEE e
                WHERE ROWNUM = 1
                ORDER BY WORKDEPT asc, FIRSTNME desc)
```

DB2 equivalent (using a common table expression):

```sql
with emp_order(empno, rownum) as
    (SELECT EMPNO, row_number() over(order by WORKDEPT asc, FIRSTNME desc)
     from employee)
SELECT *
FROM DEPARTMENT
WHERE MGRNO = (select empno FROM EMP_ORDER WHERE ROWNUM=1);
```

DECODE Function, CASE Expressions

DB2 does not provide the DECODE function. Instead, it supports the CASE expression. The mapping of DECODE to a CASE expression is very direct:

```sql
DECODE (condition, case1, assign1, case2, assign 2....,default)
```

```sql
CASE condition
    WHEN case1 THEN assign 1
    WHEN case2 THEN assign 2
    ....
    ELSE default
END
```

Case expression can be used for more than just mapping DECODE function calls. Since DB2 UDB triggers can contain only SQL statements, the CASE expression can be used to translate logic in Oracle triggers.

Functions

In Oracle, a function is a named block of PL/SQL code that accepts zero or more variables and returns one value. Many functions are supplied with Oracle, such as SUBSTR and ROUND. Other functions can be written by users to fill more specific requirements.

Many of the functions supplied with Oracle are also supplied with DB2 with the same name and behaviour, so no effort is required to port those invocations. Other functions have a different name in DB2 from in Oracle, but serve the same purpose, and these cases can also be easily ported as sourced user-defined functions (UDFs, see below). The remaining functions used in an Oracle application are either: (1) functions supplied with Oracle that have no equivalent in DB2, or (2) user-written functions. Both of these must be converted to run with DB2, as UDFs. See Appendix E - Sample DB2 User-Defined Functions (UDFs)” on page 76 for examples of Oracle functions converted to DB2 UDF’s.

There are several types of UDFs in DB2, all of which are created using the CREATE FUNCTION statement. The basic UDF types are:

Sourced UDF
A sourced UDF is a reuse of the implementation of an existing function, usually with some attributes, such as input data types, changed. A sourced UDF can also be used as a form of alias. If there were a function in Oracle, say ORAFUNC, that had an equivalent called DB2FUNC in DB2, a sourced UDF called ORAFUNC could be created in DB2, invoking the DB2FUNC function, and Oracle statements calling ORAFUNC would not need to be changed. However, as far as we are aware, all functions in DB2 either have the same names as the Oracle equivalents, or perform different tasks.

SQL-bodied UDF
This type of UDF (new in V7.1) allows the UDF to return a scalar, row, or table as the output of an SQL fullselect or a dynamic compound statement. (See the SQL Reference for the definition of a fullselect; it can include a common table expression, so fairly complex statements are possible.) For example:

```sql
create function max_sal(dept_no int) returns dec(9,2) reads sql data
    return select max(salary) from staff where dept = max_sal.dept_no
```

(Note use of parameter in bold.)

See “Appendix E - Sample DB2 User-Defined Functions (UDFs)” on page 76 for more examples.

External UDF
This type of UDF must be used when the UDF cannot be based on an existing function and cannot be reasonably expressed as a dynamic compound SQL statement. An external UDF must be written in C, C++, or Java, then compiled, linked, and registered in the database via a CREATE FUNCTION statement. See “Create Function” in the SQL Reference, and the Application Building Guide for more information.

Sampling Rows from a Table (or Result Set)
Oracle allows a random sample of rows to be returned from a single table (joins are not allowed). For example:

1. Select * from T1 SAMPLE (10) -- 10% of rows will be returned
2. Select * from T1 SAMPLE BLOCK (.001) -- return .001% of rows by sampling blocks
3. SELECT COUNT(*) * 100 FROM orders SAMPLE (1) -- estimate number of rows in orders

This functionality can be duplicated in DB2 (except for the BLOCK option) by using the rand() function, which returns a value between 0 and 1 for each row examined. The DB2 equivalents of the above examples are:

1. Select * from T1 where rand() < 0.1
2. N/A
3. SELECT COUNT(*) * 100 FROM orders where rand() > 0.99

If an exact number of random rows is required, the value compared to rand() can be changed to allow slightly more rows to qualify, and the answer set can be cut off after the desired number. Strict randomness is lost, however, because rows near the end of the answer set are not returned frequently enough. For example, if the orders table contains 10000 rows and you want to always return 100 near-random rows:

```sql
Select * from T1 where rand() < 0.0115 -- return 115 rows on average ...
fetch first 100 rows only; -- but cut off after first 100
```

Packages
An Oracle package is a named collection of procedures and functions. There is no equivalent collection in DB2 as procedures and functions are separate objects. The components of Oracle packages must be created individually in DB2, but you can use the Oracle package names as schema names in DB2 to logically group the components of the packages.
Redesign is necessary to implement package variables in DB2. One approach is to declare a global temporary table (see “Temporary Tables” on page 26), store one row in it, with a column for each package variable, then Select from (or Update) the temporary table wherever the package variables need to be used.

Note that in DB2, a package is the database object used to store the definition of an embedded SQL application when it is bound to the database, so it bears no relationship to the term package in Oracle.

**Triggers**

DB2 UDB supports BEFORE ROW, AFTER STATEMENT, and AFTER ROW triggers; the BEFORE STATEMENT trigger is not supported. DB2 UDB also supports transition variables, OLD and NEW tables, and OLD and NEW column values.

A BEFORE trigger can invoke a dynamic compound statement (excluding Insert, Update, and Delete statements) or any of the SQL control statements listed in “Named Savepoints and Compound SQL (Static and Dynamic)” on page 21.

An AFTER trigger can invoke a dynamic compound statement or any of the SQL control statements listed in “Named Savepoints and Compound SQL (Static and Dynamic)” on page 21. The dynamic compound statement support for triggers was added in V7.2.

The dynamic compound statement support for triggers was added in V7.2, and provides the ability to support the logic in an Oracle trigger body, which consists of an anonymous PL/SQL block.

Since DB2 UDB does not have the Oracle concept of versioning, one cannot get mutating errors in DB2: there is only one version of a row at any one time.

Another semantic difference between Oracle and DB2 triggers concerns checking the primary key constraint in an Insert statement. Oracle verifies that a primary key exists for an inserted row after all the triggers have fired. DB2 insists on the primary key being defined in the original insert statement; that is, you must supply the primary key value in the insert statement and not in the trigger body.

Some other trigger differences between Oracle and DB2:

- A DB2 trigger body cannot invoke stored procedures. In many cases, the logic and functionality of the stored procedures can be moved into the trigger body.

- DB2 BEFORE triggers cannot trigger INSERT, UPDATE, or DELETE statements, so when these operations are needed, the trigger must be changed to AFTER.

- Whereas in Oracle a particular trigger can be defined to be fired when an insert, update, or delete occurs against the subject table, in DB2 a separate trigger must be created for each pertinent operation (insert, update, or delete).

- DB2 allows triggers to be fired only for INSERT, UPDATE, and DELETE statements. Oracle allows other events, such as DDL operations, to fire triggers.

- DB2 does not have an ALTER TRIGGER statement to allow a trigger to be disabled. To disable a trigger in DB2, it must be dropped.

- INSTEAD OF triggers are not currently supported in DB2.

Here is a simple Oracle trigger and the corresponding DB2 triggers, with the differences highlighted:

**Oracle trigger:**

```
CREATE TRIGGER salary_check
BEFORE
```
INSERT OR UPDATE OF sal, job ON emp
FOR EACH ROW
WHEN (new.job <> 'PRESIDENT')
BEGIN
    insert into salary_hist (emp_num, old_salary, new_salary, chg_date)
    values (:new.emp_num, :old.sal, :new.sal, sysdate);
END;

DB2 triggers: (two are required, one for INSERT and one for UPDATE )

CREATE TRIGGER salary_check1
AFTER INSERT ON emp
REFERENCING new as new
FOR EACH ROW MODE DB2SQL
WHEN (new.job <> 'PRESIDENT')
BEGIN ATOMIC
    insert into salary_hist (emp_num, old_salary, new_salary, chg_date)
    values (new.emp_num, null, new.sal, current date);
END

CREATE TRIGGER salary_check2
AFTER UPDATE OF sal, job ON emp
REFERENCING old as old, new as new
FOR EACH ROW MODE DB2SQL
WHEN (new.job <> 'PRESIDENT')
BEGIN ATOMIC
    insert into salary_hist (emp_num, old_salary, new_salary, chg_date)
    values (new.emp_num, old.sal, new.sal, current date);
END

Here is a simple DB2 trigger that shows the usage of a dynamic compound SQL statement (new in V7.2) in the trigger body. The @ sign is used as the terminator for the CREATE statement.

CREATE TRIGGER find_salary
NO CASCADE BEFORE insert ON emp
REFERENCING NEW AS incoming
FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
    declare salerror condition for sqlstate '99999';
    declare new_salary dec(5,0);
    IF incoming.job is null THEN
        signal salerror set message_text='no job title supplied';
    ELSE
        set new_salary = (select salary from salary_by_job where job=incoming.job);
        if new_salary is null then
            signal salerror set message_text='job title not found in salary table';
        else
            set incoming.sal = new_salary, incoming.chg_date = CURRENT DATE;
        end if;
    END IF;
END @

Here are two examples of DB2 UDB triggers for update of the same table. Note that “!” is used as the statement terminator to allow “;” to be the terminator for possible multiple SQL statements within the trigger.

drop trigger rclass1!
CREATE TRIGGER rclass1 AFTER UPDATE of rid, rclass ON riskclass
REFERENCING NEW_TABLE AS new_rows
FOR EACH STATEMENT MODE DB2SQL
when ((select count(*) from new_rows ) > 0)
BEGIN ATOMIC
    SIGNAL SQLSTATE '78000' ('riskclass updates are not allowed for this field');
END!

drop trigger   rclass2!
CREATE TRIGGER rclass2 AFTER UPDATE of rtype ON riskclass
    REFERENCING NEW_TABLE AS new_rows
    FOR EACH STATEMENT MODE DB2SQL
    when ((select count(*) from new_rows ) > 0)
    BEGIN ATOMIC
        VALUES (CASE
            WHEN (select count(i.rtype) from new_rows i, rhierarchy r
                where i.rid = r.parent and i.rtype = 1) > 0
                THEN raise_error('78000','Can not update to simple rtype - riskclass
                has children')
            WHEN ( select count(i.rtype) from new_rows i, trades t
                where i.rid = t.rid and i.rype = 2)  > 0
                THEN raise_error ('78000','Can not update to super rtype ")
            ELSE 0
            END);
    END;
A convoluted and inadvisable way of executing complex query logic is by making operating system calls from a
UDF. The operating system call can invoke a program that contains SQL statements. This method may have
unexpected timing consequences and is not recommended as a standard porting practice.

Views and Materialized Views

Views are quite similar in Oracle and DB2, and in most cases an Oracle CREATE VIEW statement can be run
without changes against DB2. Some differences:

DB2 does not support the “[Create] or Replace” option. Views must be explicitly dropped before being recreated.

The DB2 Alter View statement is quite limited in functionality (it only allows a scope to be added to a reference
type column). Changing a view in other ways requires that the view be dropped and a (revised) Create View
statement be run.

An Oracle materialized view is an object that can be used to precompute, summarize, or replicate data. The primary
purpose is to improve performance by having work done before it is required, and cost-based optimization allows
the materialized view to be accessed instead of the base table(s).

The concept of a materialized view is implemented in DB2 as a summary table. Here is an example:

create table staffsum1 as (  
    select count(salary) as salcount, sum(salary) as salsum, dept  
    from staff  
    group by dept  
) data initially deferred refresh deferred;

The summary table can be refreshed by using the REFRESH TABLE statement:

refresh table staffsum1;

For more information on summary tables, see the discussion of Create Table in the SQL Reference, and the topic
“Creating a Summary Table” in the Administration Guide.

Joins

Oracle and DB2 provide the same basic types of join techniques at run-time: nested loop, merge, and hash. As of
V7.2, the DB2 optimizer does not consider the hash join technique to be available by default. To override this, set
the registry variable DB2_HASH_JOIN to YES. DB2 also provides another registry variable, DB2_ANTIJOIN, to inform the optimizer whether to consider transforming "NOT EXISTS" subqueries into anti-joins which can be processed more efficiently by DB2.

From a join syntax point of view, there are some differences that must be accounted for when porting:

In Oracle, the NATURAL keyword indicates that a natural join is being performed. A natural join is based on all columns in the two tables that have the same name, and selects rows from the two tables that have equal values in the relevant columns. DB2 does not support this syntax. The Oracle statement Select * from A natural join B, for example, would need to be rewritten as Select * from A, B where A.c1 = B.c1 and A.c2 = B.c2 ..., where c1, c2, ... are all of the columns that have the same names in tables A and B.

DB2 UDB supports the ANSI SQL syntax for three types of outer join: right, left, and full. Oracle supports the same syntax, starting in version 9i. Oracle also has old left and right outer join syntax that DB2 does not support. The following examples demonstrate how to map this old syntax to the DB2 equivalent:

<table>
<thead>
<tr>
<th>Oracle Outer Join</th>
<th>DB2 UDB Outer Join</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT A.last_name, A.id, B.name FROM emp A, Customer B WHERE A.id (+) = B.sales_rep_id;</td>
<td>SELECT A.last_name, A.id, B.name FROM emp A RIGHT OUTER JOIN customer B ON A.id = B.sales_rep_id;</td>
</tr>
<tr>
<td>SELECT A.last_name, A.id, B.name FROM emp A, Customer B WHERE A.id = B.sales_rep_id (+);</td>
<td>SELECT A.last_name, A.id, B.name FROM emp A LEFT OUTER JOIN customer B ON A.id = B.sales_rep_id;</td>
</tr>
</tbody>
</table>

**Stored Procedures**

In Oracle, stored procedures must be programmed in PL/SQL. In DB2 V7.1 or later, SQL stored procedures can be written. The language used for them is a subset of the ANSI/ISO Persistent Stored Modules specification, and is similar to PL/SQL and other stored procedure languages. To facilitate the writing and building of SQL procedures, DB2 provides the Stored Procedure Builder GUI tool.

DB2 still allows stored procedures to be written in any language supported by the DB2 UDB precompilers, including Java, C, C++, REXX, Fortran, and COBOL. Stored procedures can be written with embedded SQL (static or dynamic) or CLI. A DB2 stored procedure is basically the same as any loadable program, except that it is loaded and executed on the server.

When Oracle stored procedures are migrated to DB2, the most appropriate approach is to use SQL procedures, because of their similarity to PL/SQL and their excellent performance. SQL procedures are automatically converted to C code, compiled, and linked when the Create Procedure statement is executed, or through the Stored Procedure Builder.

DB2 SQL procedures have a limit of 90 parameters. When porting PL/SQL procedures with more than 90 parameters, unless some of the parameters are unnecessary and can be eliminated, the original procedure needs to be ported to multiple procedures.

A stored procedure can be invoked by an SQL CALL statement from the client. This can be done through an embedded SQL program, an ODBC or CLI program, the Command Line Processor, or any other 3rd-party front-end DBMS tool that allows stored procedure invocation. Such tools use ODBC to invoke DB2 procedures.

The main advantages of a stored procedure are that changes can be made without client code needing to change, and that network performance is optimized because multiple SQL statements are executed with only one network crossing (the stored procedure invocation).

For more information on stored procedures, consult the Application Development Guide and the Application Building Guide. Sample files of stored procedures and their invocations are shipped with the DB2 Application Development Client and can be found in sqllib/samples/XXXX, where XXXX is the programming language: sqlproc (for SQL procedures), c, cpp (for C++), java, cobol, and so on.
This table shows client code to invoke a stored procedure, and the procedure itself:

<table>
<thead>
<tr>
<th>Action</th>
<th>Oracle PL/SQL</th>
<th>DB2 UDB C client code and SQL Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calling code from a client</td>
<td>product.id%type v_id; product.name%type v_name; execute package1.SP_InsertRow (:v_id,:v_name); ......check result...</td>
<td>EXEC SQL BEGIN DECLARE SECTION; char procname[255]=”/home/sqllib/function/proc!SPInsertRow”; char id[10]; char name[20]; EXEC SQL END DECLARE SECTION; EXEC SQL CALL :procname USING :id, :name; if (sqlcode != 0) ...</td>
</tr>
<tr>
<td>Actual stored procedure</td>
<td>CREATE OR REPLACE PROCEDURE SPInsertRow (v_id in product.id%TYPE, v_name in product.name%TYPE) IS BEGIN INSERT INTO product (id,name) VALUES (v_id, v_name); END SPInsertRow;</td>
<td>CREATE PROCEDURE SPInsertRow (IN v_id CHAR(10), IN v_name CHAR(20)) LANGUAGE SQL BEGIN INSERT INTO product (id,name) VALUES (v_id, v_name); END @</td>
</tr>
</tbody>
</table>

More examples of PL/SQL code conversion to SQL Procedures are found in the table in “DB2 SQL Control Statements in Procedures” on page 40.

The following are two stored procedure enhancements that were made in DB2 V7.1. If you must use an earlier release of DB2 and need these capabilities, some tips are available in an earlier version of this document.

- Stored procedure calls can now be nested to 16 levels, but within SQL, C, and C++ stored procedures only.
- Commit (but not two-phase) and rollback can now be issued within stored procedures.

### Java Stored Procedures

DB2 UDB supports Java stored procedures, through the JDBC and embedded SQL (SQLJ) interfaces. “Appendix F - Sample DB2 Java Stored Procedure” on page 78 contains a Java stored procedure. DB2 UDB also supports JAR files containing one or more stored procedures. Use CALL sqlj.install() to install JAR files. You can also use CALL sqlj.replace_jar() (in V6.1 fixpak 3 or in V7.1 or later) to refresh JAR files and have the new versions picked up without restarting DB2. For more details consult the Application Development Guide.

### FENCED and UNFENCED Stored Procedures

DB2 UDB has two types of stored procedures, FENCED (the default) and UNFENCED. In “Server Architecture” on page 7, we discussed the various processes or threads running in the DB2 database server under the INSTANCE user ID, and that information is pertinent to the following discussion.

For a FENCED stored procedure on UNIX, DB2 spawns a new process under the user ID specified by the fencedID parameter of the Create Instance command (db2icrt). On other platforms a new database manager thread is created. Nested stored procedures run within the same process or thread as the first procedure that is called.

An UNFENCED stored procedure is run in the same memory space as the database server, which reduces inter-process communication and thus improves performance. Performance is the primary reason to use UNFENCED stored procedures instead of FENCED stored procedures. However, DB2 UDB cannot prevent data
corruption caused by invalid pointers in UNFENCED stored procedures. Therefore, it is important to have an UNFENCED stored procedure well tested before it is installed. UNFENCED stored procedures cannot be nested or return result sets.

On UNIX, the default path for a FENCED stored procedure is sqllib/function and the default path for an UNFENCED stored procedure is sqllib/function/unfenced. On Windows and OS/2 platforms the path names are similar but depend on the value of the DB2INSTPROF registry setting.

Oracle does not have the concept of FENCED and UNFENCED stored procedures. The stored procedure is loaded in the SGA area and executed by one of the server processes, called the PL/SQL engine. Basically, the Oracle implementation can be best mapped to an UNFENCED stored procedure in DB2.

CREATE PROCEDURE

Oracle stored procedures are registered in the Data Dictionary automatically. Oracle also stores the interfaces, actual text and the compiled p-code in the Data Dictionary.

It is recommended, and in most cases mandatory, to register a DB2 stored procedure in the System Catalog using the CREATE PROCEDURE statement. In the case of an SQL procedure, Create Procedure also contains the text of the procedure. Other functions of CREATE PROCEDURE:

- If the stored procedure is not installed in the default directory, CREATE PROCEDURE can be used to direct DB2 UDB to the path from which to load the stored procedure.
- CREATE PROCEDURE allows users to store the interface definitions, including parameter style.
- Running CREATE PROCEDURE allows ODBC/CLI applications to identify available procedures and their parameters through the SQLProcedures and SQLProcedureColumns functions.

Select from DUAL

To get system information, such as SYSDATE, Oracle provides a dummy table called DUAL. In DB2 UDB, convert the query to a VALUES clause or create a simple assignment statement from special registers. Or you can create a dummy DUAL table in DB2 and in some cases leave the Oracle statement unchanged. However, DUAL would need to be created under all schemas used implicitly in DB2 for those Select statements ported from Oracle. Another alternative is to use the DB2 catalog table SYSIBM.SYSDUMMY1, which has a single row and one column, IBMREQD, with a value of Y.

<table>
<thead>
<tr>
<th>Oracle</th>
<th>DB2</th>
</tr>
</thead>
<tbody>
<tr>
<td>select SYSDATE from DUAL</td>
<td>VALUES(CURRENT DATE) INTO &lt;variable&gt;</td>
</tr>
<tr>
<td></td>
<td>or select CURRENT DATE from DUAL</td>
</tr>
<tr>
<td></td>
<td>or select CURRENT DATE from SYSIBM.SYSDUMMY1</td>
</tr>
</tbody>
</table>

Nulls, NVL, and Concatenation

In Oracle, function NVL provides a conversion of NULL values to non-null values. NVL(TO_CHAR(MANAGER_ID), ’No Manager’) converts all of the NULL values in the manager_id column to the string ’No Manager’. AVG(NVL(GRADE, 0)) takes the average of column GRADE, treating the null values as 0. In DB2, use the COALESCE function to convert nulls, as in COALESCE(MANAGER_ID, ’No Manager’).

Both Oracle and DB2 support CONCAT and || (two vertical bars) as synonyms for the concatenation operator. Be aware of result data type and length differences between Oracle and DB2. For DB2, refer to the SQL Reference: in the section “[Expressions ...] With the Concatenation Operator” there is an extensive table showing the result data type and length for the various combinations of operands. For example, concatenating a CHAR(m) and a CHAR(n) produces a VARCHAR(m+n) in DB2 if (m+n) > 254, while Oracle produces a CHAR(m+n) for (m+n) up to 2000.
A difference in CONCAT behavior occurs when nulls are involved. In DB2, if either operand is NULL, the result is NULL. In Oracle a null is treated as a zero-length string, so it does not cause a concatenation of a null to produce null result. Here is an example that shows this:

```sql
create table xxx (c1 varchar(5));
insert into xxx values ('zyx');
insert into xxx values (null);
select 'abc' || 'def' || c1 from xxx;
```

In Oracle, the result of the above Select is:

```
'ABC'||'DEF
----------
abcdefzyx
abcdef
```

In DB2, the result is:

```
1
----------
abcdefzyx
- [ i.e., null ]
```

To have the DB2 result match Oracle’s, change the SELECT to either of these (which change nulls to empty strings):

```sql
select 'abc' || 'def' || coalesce(c1,'') from xxx;
select 'abc' || 'def' || value(c1,'')    from xxx;
```

In the output of SELECT statements in both Oracle and DB2, by default the null values of a column appear last when that column is ordered in ascending sequence, and first when ordered in descending sequence. Oracle’s SELECT statement allows NULLS FIRST or NULLS LAST to be specified as a way to override the default. DB2 does not provide this syntax, but a way to obtain a similar result is to use coalesce() to convert null to the empty string, which is at the opposite end of the collating sequence from null. For example:

(a) select LASTNAME from xxx ORDER BY LASTNAME ASC;   -- nulls last
(b) select coalesce(LASTNAME,'') from xxx ORDER BY 1 ASC;   -- nulls first
(c) select coalesce(LASTNAME,'') as LASTNAME from xxx ORDER BY LASTNAME ASC;   -- nulls first
(d) select LASTNAME from xxx ORDER BY LASTNAME DESC;   -- nulls first
(e) select coalesce(LASTNAME,'') from xxx ORDER BY 1 DESC; -- nulls last
(f) select coalesce(LASTNAME,'') as LASTNAME from xxx ORDER BY LASTNAME DESC;   -- nulls last

With the above approach you have to be careful because the application actually sees empty strings returned, not nulls. If that must be avoided, you can keep the original column in the select list and order on the coalesced form of it, as in this altered version of query (f) above:

```sql
(g) select LASTNAME, coalesce(LASTNAME,'') as LASTNAME_NO_NULLS  
    from xxx ORDER BY LASTNAME_NO_NULLS DESC;   -- nulls last
```

Set Operators: UNION, INTERSECT, and MINUS

Set operators can be used to combine result sets in both Oracle and DB2. The differences between the products:

DB2 supports the ALL option on each operator, allowing duplicates to be preserved. Oracle allows ALL only on UNION. MINUS is not supported by DB2. DB2’s equivalent is EXCEPT (without the ALL option).
Rename

Oracle supports rename table, view, and sequences. DB2 supports the renaming of tables and table spaces. Applications that rename views need to be changed to drop and recreate the views.

ALTER TABLE

While DB2 UDB and Oracle support an ALTER TABLE statement, there are differences in what can be changed. The following table gives a quick summary of these differences. (Not all of the DB2 features are shown.)

<table>
<thead>
<tr>
<th>Function</th>
<th>Oracle</th>
<th>DB2 UDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add column</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Increase column size</td>
<td>yes</td>
<td>VARCHAR only</td>
</tr>
<tr>
<td>Decrease column size</td>
<td>yes, if it contains only NULLs or no rows</td>
<td>no</td>
</tr>
<tr>
<td>Change column type</td>
<td>yes, if it contains only NULLs or no rows</td>
<td>no</td>
</tr>
<tr>
<td>Change VARCHAR2 to CHAR</td>
<td>yes</td>
<td>N/A</td>
</tr>
<tr>
<td>Add NOT NULL constraints</td>
<td>yes, only if there is no NULL in the column</td>
<td>no</td>
</tr>
<tr>
<td>Enable/disable constraints</td>
<td>yes</td>
<td>(similar function is provided by the SET INTEGRITY statement)</td>
</tr>
<tr>
<td>Add /drop constraints</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>NOT LOGGED INITIALLY</td>
<td>N/A</td>
<td>yes</td>
</tr>
<tr>
<td>VARCHAR2 to LONGVARCHAR</td>
<td>yes, if it contains only NULLs or no rows</td>
<td>N/A</td>
</tr>
<tr>
<td>Cache the table</td>
<td>yes</td>
<td>N/A (but the table’s table space can be assigned to its own buffer pool)</td>
</tr>
</tbody>
</table>

Array INSERTS / UPDATES / FETCHES

For embedded static or dynamic SQL, DB2 UDB does not support array inserts, updates, or fetches. When these facilities are used in Oracle, the best ways to map them to embedded SQL in DB2 are:

**INSERT:** Use a separate set of host variables for each row to be inserted, and include multiple rows in the Insert statement. Merging single-row inserts for even five rows into one Insert statement reduces the number of server communications by 80% for a given number of inserts. For example:

```sql
INSERT INTO T1 VALUES (:row1_col1, :row1_col2, :row1_col3, ..., :row1_colN),
(:row2_col1, :row2_col2, :row2_col3, ..., :row2_colN),
(:row3_col1, :row3_col2, :row3_col3, ..., :row3_colN),
(:row4_col1, :row4_col2, :row4_col3, ..., :row4_colN),
(:row5_col1, :row5_col2, :row5_col3, ..., :row5_colN);
```

Alternatively, compound SQL can be used. See “Named Savepoints and Compound SQL (Static and Dynamic)” on page 21.

**UPDATE:** Use compound SQL. See “Named Savepoints and Compound SQL (Static and Dynamic)” on page 21.

**FETCH:** In DB2, blocking of fetches normally occurs by default, so network traffic is minimized even if array fetch is not used. Fetching one row at a time in the application does not have a major impact on performance.

The DB2 Call Level Interface (CLI) does support array inserts and fetches, but not array updates, so compound SQL can be used to optimize mass updates in CLI.
Objects

Oracle8 supports the concept of objects. An object has associated attributes and methods. Attributes are static, similar to columns in a table. Methods are functions that act on the attributes. All attributes and methods in Oracle8 are public (that is, it has no private attributes or methods). You can modify the methods of an object, but you cannot alter the attributes of an object.

DB2 supports structured types, which are useful for modelling objects that have a well-defined structure consisting of attributes. Structured types feature the property of inheritance and the capability of storing instances of a structured type either as rows in a table, or as values in a column. Users can create methods to define the behavior of objects represented by structured types. DB2 also supports typed tables, where a column can have a composite type.

Oracle8 objects can be converted using these DB2 object-relational features to achieve the identical functionality. For more information on object support in DB2, see the topic “Object-Relational Programming” in the Application Development Guide.
Application Development

For any significant application to use an RDBMS, a programming language is required as a framework for the SQL language. The programming language controls the logic flow and handles tasks outside the scope of the database. It must also handle error and exception conditions. This section discusses the programming environments associated with Oracle and DB2 UDB.

Most Oracle applications are programmed in either PL/SQL, C, or C++.

PL/SQL is a proprietary language that is only available with Oracle. As mentioned earlier, the DB2 equivalent of PL/SQL is DB2’s SQL procedure language, which was enhanced and renamed to SQL Control Statements in V7.2. As of V7.2, SQL control statements can be used in stored procedures, functions, methods, and standalone SQL statements (through dynamic compound statements).

C and C++ are general-purpose programming languages that can be used by many applications, including those accessing databases. DB2 UDB supports all of the commonly used programming languages such as C, C++, REXX, COBOL, Fortran, and Java. DB2 UDB also supports stored procedures in each of these languages.

DB2 databases can be accessed using Embedded SQL applications (static or dynamic) and/or a callable SQL interface called DB2 Call Level Interface (CLI). CLI is a C/C++ application programming interface for dynamic SQL, and is based on the ISO standard for SQL/CLI and the Microsoft Open Database Connectivity (ODBC) specification.

The Oracle Call Interface (OCI) is a proprietary Oracle programming interface for the C language. DB2 UDB does not have a direct mapping for OCI applications, but when ported to DB2 they are usually converted to use CLI. This conversion is relatively straightforward and the flow of execution is the same, but the OCI call names differ from the CLI function names. To assist in the conversion, the most commonly used OCI calls are mapped to CLI in the table in “Appendix I - Mapping of Oracle8 OCI to DB2 CLI” on page 88.

With the advent of SQL control statements in DB2, there is much less need to convert PL/SQL code to C for DB2. Most of the sample conversions of PL/SQL to C have been removed from this document. If you need them, obtain an earlier version of this document.
## DB2 SQL Control Statements in Procedures

SQL control statements (new in V7.1, enhanced in V7.2) are the obvious vehicle to use when porting PL/SQL programs to DB2. The following table shows the mapping of some of the PL/SQL constructs to the SQL control statements that are available for use in procedures. Full examples of SQL Procedures are shipped with DB2 in sqllib/samples/sqlproc, and one of them is shown in “Appendix D - Sample DB2 SQL Procedure” on page 75.

<table>
<thead>
<tr>
<th>PL/SQL Statement</th>
<th>SQL Control Statement in a Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>DECLARE rowtype1 table1%ROWTYPE; [columns a,b,c]; BEGIN Variable9 datatype9 := value9; END; DECLARE variablea datatype1; DECLARE variableb datatype2; DECLARE variablec datatype3; DECLARE variable9 datatype9; DEFAULT value9; SELECT a, b, c INTO variablea, variableb, variablec FROM table1 WHERE c = xxxx; END;</td>
<td></td>
</tr>
<tr>
<td>DECLARE CURSOR mycursor IS SELECT a, b FROM table1 WHERE b=yyyy; BEGIN FOR table1_rec IN mycursor LOOP a_total := a_total + table1_rec.a; END LOOP; FOR my_loop AS SELECT a, b FROM table1 WHERE b=yyyy DO set a_total = a_total + a; END FOR;</td>
<td></td>
</tr>
<tr>
<td>DECLARE variablel := value1; set variablel = value1;</td>
<td></td>
</tr>
<tr>
<td>SELECT a INTO variablel FROM table1 WHERE b=yyyy; set variablel = (SELECT a FROM table1 WHERE b = yyyy);</td>
<td></td>
</tr>
<tr>
<td>IF condition1 THEN statement1; statement2; ELSEIF condition2 THEN statement3; statement4; ELSE statement5; END IF; IF condition1 THEN statement1; statement2; ELSEIF condition2 THEN statement3; statement4; ELSE statement5; END IF;</td>
<td></td>
</tr>
<tr>
<td>LOOP statement1; statement2; EXIT WHEN condition; [or: if condition then exit; end if;] END LOOP; LOOP statement1; statement2; END LOOP;</td>
<td></td>
</tr>
<tr>
<td>WHILE condition LOOP statement1; statement2; END LOOP; WHILE condition DO statement1; statement2; END WHILE;</td>
<td></td>
</tr>
<tr>
<td>FOR index IN lower_bound..upper_bound LOOP statement1; statement2; END LOOP; SET index = lower_bound; WHILE index &lt;= upper_bound DO statement1; statement2; SET index = index + 1; END WHILE;</td>
<td></td>
</tr>
</tbody>
</table>
C / C++

Other than SQL Procedures (starting in DB2 V7.1), C is the most commonly used programming language for DB2 UDB, with Java increasing in popularity. The Oracle Proc*C/C++ Precompiler conforms to the Entry SQL92 ANSI standard, and also provides a FIPS Flagger to identify ANSI extensions. It is recommended that the existing Oracle Embedded SQL applications be precompiled with the following options to facilitate migration to DB2:

```
FIPS=YES  MODE=ANSI  DBMS=V7  ORACA=NO
```

In “Data Types” on page 15 the table summarized the mapping of DB2 datatypes to C declarations. Here are some key points:

- In the DECLARE section, DB2 does not accept typedef types. The type declaration must be explicit.

- When declaring a character string type of size N in DB2, ensure that you declare it as size N+1 in C, as C requires a null terminator for its string. In CLI this can be overridden to avoid null termination on output by calling SQLSetEnvAttr with SQL_ATTR_OUTPUT_NTS set to SQL_FALSE.

- DB2 UDB does not support %type definition in C (or any language). The variable type needs to be declared explicitly when porting from Oracle to DB2.

Java

You can write DB2 client applications and stored procedures in Java. DB2 implements two standards-based Java programming APIs: Java Database Connectivity (JDBC) and embedded SQL for Java (SQLJ).

Because the JDBC and SQLJ implementations in DB2 are closely based on the standards, there should be very little to do in converting Java applications from Oracle to DB2. The topic “Programming in Java” in the Application Development Guide provides the DB2-specific aspects of Java programming for DB2. The key specific considerations for JDBC or SQLJ applications in DB2 are:

- The following class libraries are provided, and must be named in the CLASSPATH or included with your applets: `Db2java.zip` provides the JDBC driver and JDBC and SQLJ support classes, including stored procedure and UDF support.
Sqlj.zip provides the SQLJ translator class files. Runtime.zip provides Java run-time support for SQLJ applications and applets.

The following packages are provided and must be imported as shown at the top of Java source files for DB2:

```java
import java.sql.*; // required in every JDBC or SQLJ program
import sqlj.runtime.*; // required in every SQLJ program
import sqlj.runtime.ref.*; // required in every SQLJ program
```

A JDBC or SQLJ program must load one of the supplied JDBC drivers: COM.ibm.db2.jdbc.app.DB2Driver for applications (type 2 driver); or COM.ibm.db2.jdbc.net.DB2Driver for applets (type 3 driver).

The program connects to the database, specifying the location with a URL as defined in the JDBC specification and using the DB2 subprotocol; `jdbc:db2:sample` for example. Applets require you to provide the user ID, password, host name, and the port number for the applet server. Applications implicitly use the default value for user ID and password from the DB2 client catalog, unless you explicitly specify alternate values.

“Appendix F - Sample DB2 Java Stored Procedure” on page 78 contains a sample program of a Java client calling a Java stored procedure.

### CLI Dynamic SQL

CLI is a DB2 UDB application programming interface for dynamic SQL, and can also be used as an ODBC driver. CLI interfaces consist of a set of function calls which manage session-related information through handles. Instead of using host variables, CLI uses parameter markers in SQL statements, and program variables are bound to the markers. In C terminology, binding a variable means giving DB2 UDB a pointer to the location to be used.

CLI allows applications to build SQL statements and prepare them on the fly. The major benefit is that there is no precompilation or binding required prior to the execution of the application. Also, CLI is the only DB2 programming interface that currently supports updatable scrollable cursors. CLI also has many optimization techniques, such as Array Insert, that allow the application to minimize compilation and network traffic cost relative to embedded SQL applications.

Sample CLI programs are shipped with DB2 in `sqlib/samples/cli`. Use the Readme in that directory to help you find the files of particular interest to you.

### Embedded Static SQL

Both Oracle and DB2 UDB support embedded static SQL. Static SQL must be precompiled and bound prior to execution. Since compilation time can be significant, static SQL usually provides the best performance for running an application.

After compilation, DB2 UDB embedded SQL needs to be bound to a particular database, which creates a package in the database. Packages must be re-bound when table statistics are updated to ensure that the most appropriate access plan is always in use. Binding is an additional step when building DB2 UDB embedded applications. This is the stage where you can specify what type of isolation level the application requires. Isolation level and concurrency control are discussed in “Concurrency Control” on page 50. The user who executes the application must have EXECUTE privilege on the package.

Data is passed between the embedded SQL and the host language via host variables. All DB2 UDB embedded static SQL must start with the phrase EXEC SQL. DB2 UDB supports C, C++, COBOL, Fortran, and Java as host languages for embedded static SQL.
As shown by the table below, the syntax of the embedded static SQL of the two databases is almost identical:

<table>
<thead>
<tr>
<th>Declare</th>
<th>Pro C with Embedded Static Oracle SQL</th>
<th>C Function with Embedded static DB2 SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>EXEC SQL BEGIN DECLARE SECTION:</td>
<td>EXEC SQL BEGIN DECLARE SECTION:</td>
</tr>
<tr>
<td></td>
<td>long int host_var1;</td>
<td>long int host_var1;</td>
</tr>
<tr>
<td></td>
<td>long int arg2_out;</td>
<td>long int arg2_out;</td>
</tr>
<tr>
<td></td>
<td>EXEC SQL END DECLARE SECTION;</td>
<td>EXEC SQL END DECLARE SECTION;</td>
</tr>
<tr>
<td>SQL statement</td>
<td>EXEC SQL INSERT INTO table_name(col1) VALUES(:host_var1);</td>
<td>EXEC SQL INSERT INTO table_name(col1) VALUES(:host_var1);</td>
</tr>
<tr>
<td>Error handling</td>
<td>arg2_out = sqlca.sqlcode;</td>
<td>arg2_out = sqlca.sqlcode;</td>
</tr>
</tbody>
</table>

For a full example of a DB2 static SQL application, see sqllib/samples/c/openftch.sqc in a DB2 installation. This sample is also shown in “Appendix G - Sample DB2 Static SQL Program” on page 82. Note that the shipped sample programs also come with build scripts and makefiles, and for various programming languages.

**Embedded Dynamic SQL**

Both Oracle and DB2 UDB support dynamic SQL. Embedded dynamic SQL programs, like static applications, require precompile/bind, compile, and link steps in order to be made ready for execution. However, the preparation of individual statements, which includes access plan selection, is done at run time. (With a DB2 static application, this preparation is done when the application is bound to the database.) From a programming perspective, dynamic SQL allows you to construct the SQL at run time. If you do not know the exact SQL you want to issue, use dynamic instead of static SQL. Dynamic SQL also takes advantage of the latest table statistics for access plan selection, which sometimes allows more efficient plans to be chosen.

Oracle documentation classifies the different methods possible for implementing embedded dynamic SQL:

<table>
<thead>
<tr>
<th>Method</th>
<th>Kind of SQL Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>non-query without host variables</td>
</tr>
<tr>
<td>2</td>
<td>non-query with known number of input host variables</td>
</tr>
<tr>
<td>3</td>
<td>query with known number of select-list items and input host variables</td>
</tr>
<tr>
<td>4</td>
<td>query with unknown number of select-list items or input host variables</td>
</tr>
</tbody>
</table>

For methods 1 through 3, DB2 support is virtually identical to what is provided by Oracle. In method 1, EXECUTE IMMEDIATE is used to both prepare and execute the dynamic statement in one step. In method 2, PREPARE and EXECUTE statements are executed separately, and allow multiple executions of the same statement with different host variable values, after only one PREPARE. Method 3 is similar to method 2, but must be used for SELECT statements; following the PREPARE, instead of an EXECUTE, the DECLARE, OPEN, FETCH, and CLOSE statements are used to define and manipulate the cursor associated with the dynamic statement.

For method 4, Oracle supports two approaches, its own and the ANSI approach. DB2’s support is conceptually similar to both of these, but is implemented somewhat differently from both of them. In the Oracle approach, the basic steps are:

1. Allocate one or two SQLDAs.
2. PREPARE the statement.
3. DECLARE CURSOR; DESCRIBE BIND VARIABLES for the statement into the bind SQLDA.
4. OPEN the cursor.
5. DESCRIBE SELECT LIST for the statement into the select SQLDA.
6. FETCH from the cursor, using the select SQLDA.
7. CLOSE the cursor.

In the Oracle ANSI approach for method 4 the steps are similar, but there are special statements for setting up descriptors instead of SQLDAs. The sequence of statements is:

1. ALLOCATE DESCRIPTOR for input.
2. ALLOCATE DESRIPTOR for output.
3. PREPARE the statement.
4. DESCRIBE INPUT for the statement into the input descriptor.
5. SET DESCRIPTOR for input, repeat as appropriate.
6. DECLARE CURSOR.
7. OPEN the cursor using the input descriptor.
8. DESCRIBE OUTPUT for the statement into the output descriptor.
9. SET DESCRIPTOR for output, repeat as appropriate.
10. FETCH from the cursor into the output descriptor.
11. GET DESCRIPTOR, once for each output column.
12. CLOSE the cursor.
13. DEALLOCATE DESCRIPTOR for input and output.

In DB2, the basic steps for method 4 are:

1. Allocate an SQLDA.
2. PREPARE the statement.
3. DESCRIBE the statement into the SQLDA.
4. If necessary, allocate a new (larger) SQLDA and do another DESCRIBE into it.
5. DECLARE CURSOR.
6. OPEN the cursor.
7. FETCH from the cursor, using the SQLDA.
8. CLOSE the cursor.

The fundamental difference between corresponding Oracle and DB2 applications often relates to differences in the SQLDA structure. The differences and similarities of the SQLDA are discussed below (see “SQLDA Mappings” on page 45).

A sample DB2 Method 4 application is provided in “Appendix H - Sample DB2 Dynamic Embedded Application” on page 84. Other dynamic SQL samples are shipped with DB2 and can be identified by searching for dynamic in the Readme for the appropriate language (sqlib/samples/cobol, for example). The best description of implementing dynamic SQL in DB2 is in the Application Development Guide.

Cursors

Cursors in embedded SQL are very similar in Oracle and DB2 UDB, as demonstrated in the following table:

<table>
<thead>
<tr>
<th>Oracle Embedded SQL Cursor</th>
<th>DB2 UDB Embedded SQL Cursor</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXEC SQL DECLARE cursor_name CURSOR FOR SELECT col1 FROM table WHERE col2='123';</td>
<td>EXEC SQL DECLARE cursor_name CURSOR FOR SELECT col1 FROM table WHERE col2='123';</td>
</tr>
<tr>
<td>EXEC SQL OPEN cursor_name</td>
<td>EXEC SQL OPEN cursor_name;</td>
</tr>
<tr>
<td>EXEC SQL FETCH cursor_name INTO :host_var1;</td>
<td>EXEC SQL FETCH cursor_name INTO :host_var1;</td>
</tr>
<tr>
<td>EXEC SQL CLOSE cursor_name;</td>
<td>EXEC SQL CLOSE cursor_name;</td>
</tr>
</tbody>
</table>

Oracle includes the concept of cursor attributes, which can be mapped to DB2 return codes or other handling techniques. Note that DB2 UDB does not support %ROWCOUNT. The application must have its own explicit indexing or use other methods like FETCH FIRST N ROWS ONLY to control the number of rows returned. See “ROWNUM” on page 24 for details.
**Oracle Cursor Attribute** | **DB2 UDB Mapping** | **Description**
---|---|---
%ISOPEN | EXEC SQL OPEN CURSOR cursor_name; \nif (SQLCODE == -502) /* already open */ \nOR \nEXEC SQL FETCH cursor_name INTO var1; \nif (SQLCODE == -501) /* not open */ | Boolean attribute. Returns TRUE if the cursor is already open, but requires the (extra) OPEN to be issued. Or, you can assume the cursor is open, and deal with the error if it isn’t.
%NOTFOUND | if (SQLCODE == 100 ) | Boolean attribute. Returns TRUE if the fetch did not return a row.
%FOUND | if (SQLCODE == 0 ) | Boolean attribute. Returns TRUE if the fetch returned a row.
%ROWCOUNT | Use a counter variable or FETCH FIRST N ROWS ONLY | Numeric attribute. Returns the number of rows returned so far.

DB2 UDB and Oracle handle cursor exit conditions identically, except for the variable and #DEFINE name differences highlighted in the following:

### Oracle Embedded SQL return codes

```sql
status = SUCCESS;
while (status == SUCCESS) {
    EXEC SQL FETCH cursor_name
    INTO :var1;
    status = ORC_CODE;
    if (status == SUCCESS) {
        ...
    } else if (status==NO_DATA_FOUND) {
        EXEC SQL CLOSE cursor_name
    }
}
```

### DB2 UDB SQL Return Codes

```sql
status = SUCCESS;
while (status == SUCCESS) {
    EXEC SQL FETCH cursor_name
    INTO :var1;
    status = SQLCODE;
    if (status == SUCCESS) {
        ...
    } else if (status==NO_DATA_FOUND) {
        SQL_NO_DATA_FOUND
        EXEC SQL CLOSE
        cursor_name
    }
}
```

DB2 UDB supports scrollable cursors through the CLI interface. It allows the cursor to scroll forward and backward. Since CLI is a support layer for ODBC and JDBC, all of them support scrollable cursors. For more information, consult the [CLI Programming Guide and Reference](#).

### Null Indicators

Null indicators are defined as SMALLINT or short in C and C++. When a null value is fetched from a column, the host variable for the column remains unchanged, and the null indicator variable for the column is set to a negative value. If the data type can handle NULLs, the application must provide a NULL indicator, unlike in Oracle. If a NULL indicator is not provided in DB2, an SQLCODE -305 (SQLSTATE 22002) is returned.

When processing INSERT or UPDATE statements, DB2 checks the indicator variable if one exists. If the indicator variable is negative, the database manager sets the target column value to NULL if NULLs are allowed. If the indicator variable is zero or positive, the database manager uses the value of the associated host variable.

### SQLDA Mappings

The SQLDA structure is declared in the `sqlda.h` header file for both Oracle and DB2. These declarations are provided below. Note that although the SQLDA structures are quite different between Oracle and DB2, the mappings between the SQLDA components show many similarities. These mappings are shown with the Oracle SQLDA declaration below.
Oracle SQLDA Declaration and Mapping to DB2 SQLDA

```c
struct SQLDA /* Field meaning and its mapping to DB2 SQLDA */
{
  long N; /* Descriptor size in number of entries */
  char **V; /* Ptr to Arr of addresses of main variables */
  Long *L; /* Ptr to Arr of lengths of buffers */
  short *T; /* Ptr to Arr of types of buffers */
  short **I; /* Ptr to Arr of addresses of indicator vars */
  long F; /* Number of variables found by DESCRIBE */
  char **S; /* Ptr to Arr of variable name pointers */
  short *M; /* Ptr to Arr of max lengths of var. names */
  short *C; /* Ptr to Arr of current lengths of var. names */
  short *X; /* Ptr to Arr of max lengths of ind. var. names */
  short *Y; /* Ptr to Arr of cur lengths of ind. var. names */
};
```

DB2 SQLDA Declaration

```c
struct sqlname /* Variable Name */
{
  short length; /* Name length [1..30] */
  char data[30]; /* Variable or Column name */
};
```

```c
struct sqldistinct_type /* name of distinct type */
{
  short length; /* Name length [1..27] */
  char data[27]; /* Name of distinct type */
  char reserved1[3]; /* reserved */
};
```

```c
struct sqlvar2 /* Variable Description */
{
  union sql8bytelen len; /* Eight byte length. 4 bytes used now */
  char * sqldatalen; /* Pointer to four (4) byte length buffer. This may be used to hold the length for lob data types. */
  struct sqldistinct_type sqldatatype_name; /* distinct type name */
};
```

```c
struct sqlvar /* Variable Description */
{
  short sqltype; /* Variable data type */
  short sqllen; /* Variable data length */
  char * sqldata; /* Pointer to variable data value */
  short * sqlind; /* Pointer to Null indicator */
  struct sqlname sqlname; /* Variable name */
};
```

```c
struct sqlda
{
  char sqlda[8]; /* Eye catcher = 'SQLDA' */
  long sqldabc; /* SQLDA size in bytes=16+44*SQLN */
  short sqln; /* Number of SQLVAR elements */
  short sqld; /* # of columns or host vars. */
};
```
struct sqlvar sqlvar[1]; /* first SQLVAR element */

**SQLCA / ORACA**

SQLCA is the SQL Communications Area structure. The SQLCA structure is declared in the `sqlca.h` header file for both Oracle and DB2. These structures are similar for both Oracle and DB2. The table below shows how to retrieve error and warning messages from Oracle and DB2 UDB.

<table>
<thead>
<tr>
<th>Oracle Messages and Warnings via SQLCA</th>
<th>DB2 Messages and Warnings via SQLCA</th>
</tr>
</thead>
<tbody>
<tr>
<td>sqlca.sqlerrm.sqlerrml : contains the length of an error message.</td>
<td>sqlca.sqlerrm.sqlerrml: contains the length of sqlerrmc</td>
</tr>
<tr>
<td>sqlca.sqlerrm.sqlerrmc: contains the error message text</td>
<td>sqlca.sqlerrm.sqlerrmc: contains the token(s), separated by 'XFF', that are substituted for variables in the descriptions of error conditions</td>
</tr>
<tr>
<td>sqlerrm: returns the error message text that can be used for diagnostics</td>
<td>sqlaintp: decodes sqlerrmc into a descriptive message</td>
</tr>
<tr>
<td>errmsg := sqlerrm(errCode)</td>
<td>rc = sqlaintp(msgbuffer,1024,80,sqlca.sqlcode);</td>
</tr>
<tr>
<td>sqlca.sqlcode: error code</td>
<td>sqlca.sqlcode: error code</td>
</tr>
</tbody>
</table>

The ORACA structure (in `oraca.h`) provides additional diagnostic information, which is specific to Oracle and therefore not supported in DB2. The best way to map ORACA usage in DB2 is through the monitoring capabilities, as described in “Database Snapshot Monitor” on page 65 and “Database Event Monitor” on page 66.

**Error Handling**

The following table provides a list of frequently encountered error codes in DB2 and Oracle:

<table>
<thead>
<tr>
<th>Oracle</th>
<th>DB2 UDB</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-54</td>
<td>-911</td>
<td>Transaction rollback due to deadlock (or lock timeout in DB2)</td>
</tr>
<tr>
<td></td>
<td>(reason code 2 = deadlock; reason code 68 = lock timeout)</td>
<td></td>
</tr>
<tr>
<td>100 or -1,403</td>
<td>100</td>
<td>Not Found</td>
</tr>
<tr>
<td>-2, 291</td>
<td>-545</td>
<td>Integrity constraint violation</td>
</tr>
<tr>
<td>-1</td>
<td>-803</td>
<td>Duplicate record (primary key or unique constraint violation)</td>
</tr>
<tr>
<td>-1, 427</td>
<td>-811</td>
<td>More than one row returned</td>
</tr>
</tbody>
</table>

Part of porting from Oracle to DB2 involves mapping the error conditions. The mapping is not direct, so each application needs to map out the conversion from Oracle to DB2 UDB.

Oracle supports *user-defined errors* and an application can raise exceptions. DB2 UDB embedded SQL does not support user-defined errors in the SQL error handling structure. However, DB2 SQL Procedures can declare a condition as a user-defined name for an existing or user-defined SQLSTATE value, then use the SIGNAL statement to indicate to the application that the condition has occurred.

<table>
<thead>
<tr>
<th>Oracle Exception</th>
<th>Description</th>
<th>Directions for Handling</th>
<th>Example</th>
</tr>
</thead>
</table>
| Predefined Error | 1 of 20 errors that occur most often in PL/SQL; For example, | Do not declare, and allow the Oracle Server to raise them implicitly | SELECT...
...EXCEPTION WHEN NO_DATA_FOUND THEN ROLLBACK; |
DECLARE
  exception1 EXCEPTION;
...
BEGIN
  ...
  RAISE exception1;
...
EXCEPTION
  WHEN exception1 THEN
    DBMS_OUTPUT.PUT_LINE('Error.);

Declare within the declarative section and raise explicitly A condition that the developer determines is abnormal

DECLARE
  exception1 EXCEPTION;
BEGIN
  ...
  RAISE exception1;
...
EXCEPTION
  WHEN exception1 THEN
    DBMS_OUTPUT.PUT_LINE('Error.');

Declare within the declarative section, and allow the Oracle Server to raise them implicitly

DECLARE
  exception1 EXCEPTION;
PRAGMA
EXCEPTION_INIT(exception1,-2292)...
BEGIN
  ...
  EXCEPTION
    WHEN exception1 THEN
      DBMS_OUTPUT.PUT_LINE('Error.');

Declare within the declarative section, and raise explicitly

DB2 UDB exceptions are communicated mostly through the SQLCODE, although the SQLSTATE is a more platform-independent diagnostic vehicle. The EXEC SQL WHENEVER clause is used to indicate what action to take for each of the three supported conditions: SQLERROR, SQLWARNING, and NOT FOUND. Note that the syntax is the same as for Oracle embedded SQL. To map the handling of specific Oracle errors in DB2, you can use WHENEVER SQLERROR to pass control to an error routine which checks for and acts upon the specific errors.

<table>
<thead>
<tr>
<th>DB2 UDB Exception</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
</table>
| SQLERROR           | Identifies any condition where SQLCODE < 0 | EXEC SQL WHENEVER SQLERROR GO TO Error;
|                    |             | EXEC SQL UPDATE...
|                    |             | status = SQLCODE;
|                    |             | if ( status == SUCCESS)
|                    |             | ....
|                    |             | Error:
|                    |             |   status = SQLCODE;
|                    |             |   EXEC SQL ROLLBACK;
|                    |             | Return(status); |
| SQLWARNING         | Identifies any condition where SQLWARN(0)='W' or SQLCODE > 0 but not equal to 100 | EXEC SQL WHENEVER SQLWARNING CONTINUE;
|                    |             | EXEC SQL UPDATE...
|                    |             | status = SQLCODE;
|                    |             | Return(status); |
| NOT FOUND          | Identifies any condition where SQLCODE = 100 (SQLSTATE = '02000') | EXEC SQL WHENEVER NOT FOUND GO TO Error;
|                    |             | EXEC SQL UPDATE...
|                    |             | status = SQLCODE;
|                    |             | return(status); |
|                    |             | ....
|                    |             | Error:
|                    |             |   status = SQLCODE;
|                    |             |   EXEC SQL ROLLBACK;
|                    |             | Return(status); |
Constraints

Both DB2 UDB and Oracle support primary key, foreign key (referential integrity), unique, NOT NULL, and check constraints, although with minor differences. Primary key and check constraints are identical in both products. Foreign keys are discussed below.

Columns specified in a unique constraint must be defined as NOT NULL in DB2 UDB, while in Oracle a unique constraint allows NULL values, and even multiple rows with NULL values. This difference can make application changes necessary when porting to DB2.

The primary consideration for DB2 UDB is always to ensure full data integrity. In certain situations, however, preserving data integrity on a continuous basis can have a significant performance impact. For example, bulk loading of data via the Load command would be slower if for every row a detour is taken to ensure that all of the pertinent constraints are satisfied for that row.

The mechanism that DB2 uses to balance integrity and performance is to place a table in CHECK PENDING state when an operation is performed that can lead to integrity exposures (Load, for example). While the table is in that state, only limited operations can be performed against it (no Selects, for example). To make the table fully accessible again (take it out of CHECK PENDING state), DB2 must verify that all constraints are satisfied.

The SET INTEGRITY statement is used both to put a table into CHECK PENDING state and to take it out again after an operation is performed. SET INTEGRITY is the DB2 equivalent of Oracle’s “set constraint <name> deferred/immediate” statement. Here’s an example:

```sql
db2 set integrity for T1 off;
.... do Load, Alter Table to add constraints, etc.
[Load itself will place the table in CHECK PENDING state]
db2 set integrity for T1 immediate checked;
```

Referential Integrity

Both DB2 and Oracle support foreign keys where a parent table’s primary key is referenced by the child table. DB2 UDB, however, allows referential integrity (RI) to apply for any unique constraints, not just the primary key. (Note that the unique constraint can be a composite key.) This can reduce checking in the application code to enforce more complex dependencies between tables automatically.

Due to the late introduction of RI in Oracle, some Oracle applications check table dependencies by fetching from many dependent tables. They can be changed to having RI constraints in the table declarations. This can simplify applications as well as improve performance.

DB2 UDB provides the SET NULL and NO ACTION options on DELETE, in addition to the CASCADE and RESTRICT options which Oracle also provides. The default for Oracle is DELETE RESTRICT and the default for DB2 is DELETE NO ACTION; these are virtually identical in meaning. The approach to take in porting is to use the same option in DB2 as was used in Oracle.
Concurrency Control

One of the most significant differences users notice when they port applications from Oracle to DB2 is the difference in concurrency control between the two databases. This section addresses the locking behaviors of each database and explains how to map from Oracle to DB2 UDB.

Some Oracle applications, when ported to DB2, appear to behave identically, and the topic of concurrency can be ignored. However, if your applications involve frequent accesses to the same tables, you may find that your applications behave differently.

To get the best results, sometimes it is worth redesigning applications to achieve the best concurrency in DB2 UDB. Understanding concurrency control in DB2 UDB helps you know how to rework an application.

Concurrency and Locks

For both Oracle and DB2, a transaction is an atomic unit of work, in which all changes are either committed or rolled back. However, while both DB2 and Oracle have row-level locking (as opposed to page-level locking, as in SQL Server), there are differences in when the locks are acquired.

In general terms, there are share locks and exclusive locks. To update a row, the database server needs to acquire an exclusive lock on that row first. When a share lock is acquired for an object on behalf of an application, other applications can also acquire a share lock on the object, but requests for exclusive locks are denied. An exclusive lock, on the other hand, blocks other applications from acquiring locks, even share locks, on the object. The time an application is blocked because of the unavailability of a lock is called the lock wait time.

In Oracle, when an application requests a row to be fetched, no locks are acquired. In DB2 UDB, when an application issues a read request, a share lock is acquired on the row. A DB2 UDB application may acquire more share locks based on the isolation level of the application and the access plan of the query. For example, for a table scan, a share lock may be acquired for each row touched by the table scan, which may contain more rows than the result set. Alternatively, one table lock may be acquired.

Due to these differences, ported applications that have updaters and readers accessing the same data concurrently from the same table may experience more lock wait time in DB2, but with a more consistent view of the data.

Isolation Levels

Before looking at how to improve the concurrency of ported applications, it is useful to have a quick description of differences between DB2 UDB’s and Oracle’s implementation of concurrency control.

Oracle implements an optimistic view of locking. The Oracle assumption is that in most cases, the data fetched by an application is unlikely to be changed by another application. It is up to the application to take care of the situation in which the data is modified by another concurrent application.

For example, when an Oracle application starts an update transaction, the old version of the data is kept in the rollback segment. When any other application makes a read request for the data, it gets the version from the rollback segment. Once the update transaction commits, the rollback segment version is erased and all other applications see the new version of the data. Different readers of the data may hold a different value for the same row, depending on whether the data is fetched before or after the update commits. Hence, it is also called the Oracle versioning technique.

To ensure read consistency in Oracle, the application must issue SELECT FOR UPDATE. In this case, all other readers and updaters are blocked.

DB2 UDB has a suite of concurrency control schemes to suit the needs of applications. An application can set the level of isolation to provide the proper level of concurrency. One of these means can be used:
For CLI and ODBC applications, call the SQLSetConnectAttr() or SQLSetStmtAttr() function, or set the TXNISOLATION keyword in db2cli.ini.

For JDBC applications, invoke Connection.setTransactionIsolation().

For CLP scripts, issue “CHANGE ISOLATION TO <level>” before connecting to a database.

For static SQL programs, prepare or bind the program with the “ISOLATION <level>” clause.

Starting in DB2 V7.2, the isolation level can be set within the SQL statement itself, for Select, Insert, Update, and Declare Cursor statements. For example, “Select * from T1 WITH RS”.

Here is a brief description of each isolation level. For more details, consult the DB2 UDB Administration Guide.

DB2 UDB has four isolation levels: Repeatable Read, Read Stability, Cursor Stability and Uncommitted Read.

**Repeatable Read (RR)**
This is the highest level of isolation. It blocks other applications from changing data that has been read in the RR transaction, until the transaction commits or rolls back. If you fetch from a cursor twice in the same transaction, you are guaranteed to get the same answer set. Phantom rows cannot occur by showing up in the second execution of the query but not the first.

**Read Stability (RS)**
Like RR, this level of isolation guarantees that the rows read by the application remain unchanged in a transaction. However, it does not prevent the introduction of phantom rows.

**Cursor Stability (CS)**
This level guarantees only that a row of a table cannot be changed by another application while your cursor is positioned on that row. This means the application can trust the data it reads by fetching from the cursor and updating it. This is the default.

**Uncommitted Read (UR)**
This level is commonly known as a *dirty read*. When a UR transaction issues a read, it reads the only version of the data that DB2 UDB has, even though the data may have been read, inserted, or updated by another transaction. The data is labeled as *dirty* because if the updater were to roll back, the UR transaction would have read data that was never committed. Unlike Oracle, DB2 does not use the rollback segment to store the *old* version of the data. Access to the data is controlled using locks. This is the only type of read access where DB2 UDB does not acquire a share lock on the object; hence, it does not prevent the data from being updated by another application.

Oracle's implementation most resembles Read Stability (RS) in DB2 UDB for writers and most resembles Uncommitted Read (UR) for readers. Note that DB2 UDB UR is not the same as Oracle's versioning. UR reads uncommitted data if there is a transaction in progress, as opposed to the last version of the data read by Oracle. Suppose an application is reading a column from a row that has been modified by another transaction from a value of 3 to a value of 5. In Oracle, the application reads 3. In DB2 UDB, UR reads 5. If the update transaction commits, then DB2 has the right data. If the update transaction rolls back, then Oracle has the right data.

With the exception of UR transactions, DB2 UDB can guarantee that the data an application reads does not change under it. The application can trust the data it fetched. This behavior can simplify the application design.

On the other hand, because DB2 UDB requests an exclusive lock on behalf of the application during an update, no other applications can read the row (except when the UR isolation level is used). This can reduce concurrency in the system if there are a lot of applications attempting to access the same data at the same time.

To increase the concurrency of the system, commit your transactions often, including read-only transactions. If possible, reschedule the applications that compete for access to the same table. Also, use Uncommitted Read transactions where read consistency is not an issue. Use Cursor Stability whenever possible for other applications.
Some applications do cross-table consistency checking in the application code instead of using the referential integrity (RI) constraints on the tables. This approach can increase the number of locks acquired by DB2 because the applications need more cursors and fetch more data. Use of RI constraints can reduce the lock-wait time by reducing the number of cursors and the number of rows fetched in the application, and hence reduce the amount of locking done by the database.

Lock Escalation

A lock can be at the row level, the table level, or even the database level. If an application anticipates accessing a large number of rows in a table, it can request a table lock to reduce the locking overhead.

DB2 UDB may convert row locks to a table lock automatically without an explicit user request. If DB2 perceives that there are too many locks held at a given time, it performs lock escalation by acquiring a table lock and then releasing the row locks held on that table. The optimizer may also request a table lock if it thinks it is more efficient for a particular query. Oracle does not have the concept of lock escalation; the application is blocked if there is insufficient memory to acquire the lock. The application has to explicitly request a table lock in Oracle.

Lock escalation can have mixed results for the applications. Lock escalation can temporarily slow the system down because of the processing required to release all of the row locks. Once escalation is done, however, the application may run faster, because DB2 UDB no longer needs to acquire and release row locks on the table.

A more important side effect of lock escalation is the concurrency impact on other applications. For example, if an application holds a share table lock on table T1 due to a lock escalation (or other reason), other applications cannot update rows in T1 (but they can read rows in T1). Similarly, an exclusive table lock held by an application does not allow other applications to read or update rows in the table. Refer to the Administration Guide for details on locks and how they conflict.

Lock escalation is DB2 UDB’s way to handle exceptional situations where the number of lock requests is much higher than usual. Lock escalation should not be a frequent event in a production environment. The database configuration parameter that controls the amount of memory devoted to storing locks is LOCKLIST, the number of 4K blocks for recording the locks held by applications. To have predictable results, allocate sufficient memory for LOCKLIST, and acquire a table lock explicitly in your application when you feel it is appropriate. Read-only tables, and tables that are used by only one application at a time, can be altered so that table locks are always obtained, eliminating row locks and reducing CPU consumption. To do this, use:

```
ALTER TABLE <name> LOCKSIZE TABLE
```

Use DB2 database system monitor snapshots to determine if you have lock escalations in a given period and increase (or decrease) your LOCKLIST appropriately.

Deadlocks

As a result of different concurrency controls in Oracle and DB2 UDB, an application ported directly from Oracle to DB2 UDB may experience deadlocks that it did not have previously. As DB2 UDB acquires a share lock for readers, updaters may be blocked.

A deadlock occurs when a transaction cannot proceed because it is dependent on exclusive resources that are locked by some other transaction, which in turn is dependent on exclusive resources in use by the original transaction. The only way to resolve a deadlock is to roll back one of the applications.

DB2 UDB resolves deadlocks automatically, without user intervention. DB2 UDB has a background server process called the deadlock detector. It periodically examines the locks held in the system to determine if there is a deadlock. If so, it picks a victim and rolls back the transaction. Note that the victim can be a read-only transaction. If a transaction is rolled back due to a deadlock, DB2 UDB returns an error code (SQLCODE -901, SQLSTATE 40001) indicating a rollback due to deadlock.
Oracle does not have a deadlock detector to resolve deadlocks. Applications can use the NOWAIT clause or set a time-out for the session to avoid deadlocks.

You can change the frequency of the deadlock detector checks by setting the database parameter DLCHKTIME, which applies to all applications for that database.

You can monitor if a deadlock has occurred using the snapshot monitor. It shows the number of deadlocks that occurred in the system since the last system monitor reset or since the first connect to the database, whichever is later. To get detailed information on each deadlock, create an event monitor to monitor for deadlocks. This way, you can tell exactly which applications are involved in the deadlock and what is the point of contention. This information can help you reschedule or redesign your application programs.

**Optimistic Locking**

One locking situation that may require special treatment in DB2 is where an application reads a row from a table, presents data from the row to a user, waits for the user to change values, then updates the row. If two (or more) users try to process the same row at the same time, then either: (a) if the row is not read FOR UPDATE, both users can read the row in share mode, and a deadlock results when either tries to update it; or (b) if the row is read FOR UPDATE, one user’s read waits until the other user commits. If this type of application must have minimal lock wait time and the same locking behavior when ported from Oracle, it can use the optimistic locking scheme.

This technique requires the table to have a column that can identify when the row’s last change took place. A timestamp is the most typical example. Read the row and commit right away to release the lock. This allows other applications to have write access to the row (as in Oracle). When it is time to update the row, retrieve the row again to see if the row has changed. If the timestamp column has not changed since the last fetch, the application can update the row. If the timestamp has changed, which is relatively rare, the application must start the process again at the beginning, re-reading the row and presenting the latest version of its contents to the user.

```
Fetch row, including the timestamp column
commit; (release the lock)

... think time while the user makes changes ...

[start transaction]
fetch the row again and compare its timestamp with the timestamp in the previous fetch
if timestamps match
    do the work; (update the row, including the timestamp column)
else
    handle the exception situation; (usually, go back to the start)
commit;
```

**How to Improve Concurrency**

To maximize concurrency in DB2 UDB, keep the following in mind:

- Periodically commit your read-only transactions to release the share locks acquired by DB2. In Oracle, this is not required since locks are not acquired for (read-only) queries.
- Use Uncommitted Read (UR) and Cursor Stability (CS) whenever possible.
- Check return codes for rollbacks due to deadlocks.
- Use optimistic locking (described above) to reduce lock waits in the system.
- Use RI to reduce the number of cursors used in the application.
If required, reschedule your applications so the updating of a table occurs at a different time from querying the same table. To see if applications are blocking each other, observe the cumulative count of lock waits from the Database Monitor snapshots.

Use database system monitor snapshots to find out how much of the LOCKLIST is used for your environment. Increase the LOCKLIST value if your applications are encountering lock escalations. Decrease the LOCKLIST value if you do not need the space and you can make better use of the memory elsewhere.

Use database system monitor snapshots to determine if you are encountering deadlocks in your system. It provides detailed information such as which applications are involved in the deadlocks. Use the Event Monitor to determine why the deadlocks occurred. You can eliminate deadlocks by reworking the applications, committing your read transactions more often, or using another isolation level.

Logging

In “Database Recovery Logs” on page 12, we briefly discussed the differences in how DB2 UDB and Oracle log changes to a database. In particular, recall that DB2 UDB does not have rollback segments.

DB2 UDB stores changes to the database in log files. It stores log files in the directory specified by the database configuration parameter NEWLOGPATH. Log files can be system files or raw devices.

You can control the number and size of the DB2 log files via the LOGPRIMARY, LOGSECOND, and LOGFILSZ database parameters. Consult the DB2 UDB Administration Guide for details. You can reduce the amount of information logged by DB2 by using the NOT LOGGED INITIALLY clause in the Create Table and Alter Table statements. This can significantly improve the speed of updates. However, keep in mind that the data updated during the NOT LOGGED INITIALLY period is not recoverable. A typical time to use NOT LOGGED INITIALLY is when an application first populates a table using the IMPORT command or INSERT ... SELECT FROM.

Archive Logging

DB2 allows three types of logging options, which are controlled by the two database configuration parameters, LOGRETAIN and USEREXIT.

If LOGRETAIN is OFF, a situation commonly known as circular logging, DB2 UDB reuses existing log files once all the log files are full, assuming those files are inactive. A log file is inactive when all of the data pages associated with the log records in that log file have been written to permanent storage; so the log file is no longer required for recovery. Circular logging is not recommended for production systems. However, it is very useful for sandbox and development systems since one does not have to worry about allocating extra storage for archiving old log files, so very little user intervention is required.

Setting LOGRETAIN to ON and USEREXIT to OFF tells DB2 UDB to create new log files when the old ones are full, leaving the old files in the log directory.

Setting LOGRETAIN and USEREXIT to ON tells DB2 to invoke a USEREXIT program with the name db2uext2 when a log file becomes full. The user exit program can perform a variety of activities, but typically it moves the full log files to an archiving directory, which can be off-site for safety. This is the typical operational database behavior for a production system. A sample user exit is shipped in sqlib/samples/c/db2uext2.cdisk.

LOGRETAIN ON is required for DB2 to rollforward the database from a previous backup, applying the changes in the archived log files. Oracle logging can be best mapped to LOGRETAIN ON with a USEREXIT defined.
Database Administration

While the DB2 and Oracle SQL languages are mostly similar, the database administration tools are quite different. However, they are meant to accomplish similar tasks: control database layouts, control user access, monitor database activity, back up and restore the database, and so on. In this section, we describe the tools in DB2, the purpose of each, and how administration tasks should be mapped from Oracle tools. All the tools mentioned here are included with DB2 UDB EE.

The Control Center is DB2 UDB's graphical interface that shows database administrators and application programmers database objects and their relationship to each other. It is the rough equivalent to the Enterprise Manager in Oracle.

From the Control Center, you can perform almost all administrative duties by invoking different subcomponents such as Command Center, Journal, Script Center, Alert Center, and Monitors. For example, you can:

- develop and schedule scripts using the Script Center
- administer a particular DB2 instance using the Control Center
- look at database snapshots and monitor events with the monitors

This section gives a quick overview of the capabilities of some of the components. Note that some tools are invoked directly via icons, while others are invoked by right-clicking on an object and selecting the desired tool.
Control Center

In addition to acting as the central point of access for various other GUI tools, the Control Center is itself a vital tool. It lets you avoid looking up the syntax of many DB2 UDB commands and reduces the learning curve for getting started with DB2 UDB. The Control Center provides drill-down capability for many database objects. You can generate database snapshots, create event monitors, change configuration parameters, and create and manage objects in the database. Right-click on an object to find out the different functions you can perform on that database object.

You can save the commands generated by the Control Center in a script, using the Show SQL buttons. The scripts can be used later for inspection as well as subsequent non-interactive administrative tasks.

The Control Center also provides the ability to administer remote machines. A discovery feature in the Control Center allows you to add new servers and databases to the various DB2 directories that act as the connectivity mechanisms.

You should use the Control Center to change the database configuration parameters after creating a new database, as the default configuration parameters are likely to be inappropriate for your database. To do this, right-click on the database name and select Configure Performance Using Wizard.

Sample Screen of the Control Center
Command Center

The Command Center is an interactive interface for issuing DB2 commands and SQL statements. It can be used to test out statements, save them in scripts, execute existing scripts, and show the access plans of statements. It also includes the SQL Assist feature, which guides the user in writing SQL statements by presenting lists of available tables and columns and allowing statement options to be specified through a graphic interface.
Script Center

Complicated Oracle SQL*Plus scripts should be ported into the DB2 UDB Script Center. The Script Center can be invoked from the Control Center or the Command Center. The Script Center provides the ability to have scripts that contain SQL statements and DB2 commands, operating system commands, and other executables. The script can be scheduled for future activation and the status of the scheduled events can be monitored using the Journal. The Script Center cannot process query result sets into tailored reports, so if this capability is required, a separate product may be called for, such as QMF for Windows.

Journal

The Journal is used to monitor database activities. It can be used to monitor the status of scheduled jobs, review the status of completed jobs, and look at the Alerts and Message log. The Message log is particularly useful for user support because it records all messages seen by the DB2 GUI user.

Command Line Processor

The Command Line Processor (CLP) is the command line interface to DB2 UDB. Almost all commands from the Control Center can be issued in the CLP. Enter `db2` to start a CLP interactive session, or just type `db2 <command>` to enter a CLP command. CLP scripts can invoke operating system commands and other executables (prefix them with `!`). CLP is similar to SQL*Plus in Oracle. However, some of the SQL*Plus scripts can be best ported to Command Center scripts instead of to CLP scripts.

CLP can describe the syntax of a DB2 command. To do this, type a question mark before the command name. For example, typing `? backup` returns the syntax of the Backup command. (Note that this does not work for SQL statements. Refer to the SQL Reference for their syntax.) Another use for CLP is to obtain the explanation of an SQLCODE or SQLSTATE. For example, `db2 ? SQL8001` returns the explanation of SQLCODE -8001 or message SQL8001N, and `db2 ? 42968` returns the explanation of SQLSTATE 42968.

You can create a script containing a set of CLP commands and invoke the script using the `db2` command. Assuming you use a semicolon (`;`) to terminate each command, invoke the script using the `-tvf` flags [“t” means the default terminator (;) is used; “v” (verbose) means that command text is echoed to standard output; “f” means that the script filename follows], as in:

```
db2 -tvf runstuff.clp
```

Some complex SQL statements, such as trigger and procedure definitions, require semicolons to terminate their individual substatements. When these complex statements are placed in a CLP script, a different character, such as `@`, must be used to terminate the statements. To execute such a script, use the “-td” flag to designate the statement terminator, as in:

```
db2 -td@ -vf runstuff.clp
```

Authentication

DB2 utilizes the operating system for the authentication of connecting users. That is, when authentication takes place (see next paragraph), DB2 verifies that the user ID and password are valid in the appropriate repository outside of DB2 (such as `/etc/passwd` on UNIX), not through a user definition inside DB2.

The value of the database manager parameter AUTHENTICATION determines how and where the authentication of users takes place. The default value, SERVER, means that a user ID and password must be passed to the server for that user to be authenticated. Other values allow authentication to be performed on the client or at a DCE or Kerberos server, or cause flowed passwords to be encrypted between client and server.
Privileges and Authorities

After being authenticated by DB2, a user can perform tasks according to the privileges and authorities they have. Privileges are single permissions, such as the ability to connect to databases or to delete rows from a specific table. Authorities are groupings of privileges needed by a system administrator, for example. You can name one system user group for each of these authorities at the database manager level: SYSADM, SYSCTL, and SYSMAINT. When a user is a member of one of those groups, they have the corresponding privileges. For example, if you define a group called ADMINGRP on the system and set database manager configuration parameter SYSADM_GROUP to be ADMINGRP, all users belonging to that group have SYSADM privileges. At the database level, DBADM authority can be granted to specific users to allow them to access any data and run various utilities.

Oracle has the concept of roles, and a set of privileges can be granted to a role. DB2 does not support the concept of roles. To grant privileges to a database user, it must be done explicitly. To grant a privilege to all users, grant it to PUBLIC.

For more information on authentication, privileges, and authorities, see the Administration Guide, and the GRANT and REVOKE statements in the SQL Reference.

Data Management

DB2 UDB supports system-managed space (SMS) and database-managed space (DMS) table spaces. SMS uses system files to store data and DMS can use system files as well as raw devices to store data. The Oracle storage model is most similar to the DB2 UDB DMS with raw devices. The following sections give you some guidance on how to migrate Oracle table spaces to DB2 UDB table spaces. Refer to the Administration Guide topic “Table Space Design Considerations” for more information.

If you have table spaces with different page sizes, make sure that you create one temporary table space for each page size. This allows any table to be reorganized using a temporary table space, instead of requiring work space in the table’s permanent table space.

SMS Table spaces

System-managed space (SMS) uses operating system files as containers to store the data. It is the easiest way of creating and managing a database, and is the default. The limitations of SMS are often the limits of the operating system. For instance, some operating systems do not allow the size of a file to exceed 2GB.

Since the concept of SMS does not exist in Oracle, the question is: when should SMS be used when porting to DB2? SMS containers are not preallocated, and grow as the data is entered into the database. This means you need not worry about preallocating too much space for a table space. You can use the extent size of the table space to control how much the container grows each time. A disadvantage of SMS is that you cannot add containers to an SMS table space once it is created.

From a performance perspective, SMS may be less efficient than DMS because it depends on the operating system to do the file manipulation. Containers have to be extended at run time and the data is written to a file rather than to a raw device. The performance difference is usually 10% or less. It is not an issue, however, if most of the data in the table space is buffered by the buffer pool.

DB2 UDB recommends using an SMS table space for the system catalog (table space SYSCATSPACE) and all temporary table spaces (TEMPSPACE1). One reason for doing this is because in most cases, the system catalog tables are buffered in memory (buffer pool) as they are frequently accessed. Using SMS can reduce the amount of administration required without much performance cost. SMS is appropriate for temporary table spaces for the same basic reasons.

For user data, applications can use SMS for ease of maintenance and to avoid having to precommit storage for the data. These can be compelling reasons for certain environments. For example, if you just want to get the database
up and running in a sandbox environment, SMS can save a lot of startup time. Also, if the database is not expected
to grow quickly, SMS may be more than adequate.

DMS Table spaces

Database-managed space (DMS) is designed for its flexibility, extensibility, and performance. For Oracle users,
managing DMS table spaces should be familiar. Containers can be added to the table space and enlarged as the
database grows. You can specify the containers that belong to a table space when creating the table space, or you
can do it later. The containers are preallocated when they are assigned to a table space, which means there is no
overhead to extend the storage at run time.

Oracle's table space concept maps well to a DB2 UDB DMS table space with raw devices. In addition to raw
devices, DB2 UDB also allows operating files to be containers. When more than one container is defined for a table
space, DB2 UDB stripes the data across the containers. The stripe size is the extent size defined for that table
space. When a new container is added to the table space, DB2 UDB rebalances the containers automatically by
spreading the data across all of the containers. This is the reason we recommend having all containers the same
size in a table space. The applications can benefit from a maximum degree of I/O parallelism. Autorebalancing
makes DMS table spaces much easier to manage than Oracle table spaces.

Since DMS is not the default, the user must explicitly define DMS table spaces and assign them to tables, indexes
or long fields. DMS table spaces can bypass the file system, DB2 UDB has complete control of the storage and
allows more efficient access to the storage. For this reason, DMS table spaces may have better performance than
SMS table spaces.

Multiple Buffer Pools

Oracle (starting in Oracle8) supports up to three buffer pools: the keep pool, the recycle pool, and the default pool.
They are all assigned from the buffer cache in the SGA. The keep pool is used for buffers of objects that you want to
remain in the cache. The recycle pool is used for buffers that you want Oracle to discard quickly. The default pool is
used for all buffers not explicitly assigned to the keep or recycle pool.

DB2 UDB also supports multiple buffer pools, but the number of them is not limited and the names are user-defined.

The frequency of data access satisfied by the buffer pool (the hit ratio) is the single most important indicator of the
performance of the database. Hence, the ability to improve the hit ratio by allocating different amounts of memory to
buffer the pages from different table spaces can yield a dramatic performance improvement. For example, you can
assign a table's data to one table space and its indexes to another table space. If the access to the table is almost
always through index probes, one can allocate a large buffer pool for the index table space and a very small buffer
pool for the data table space. This can greatly improve the hit ratio for the table while using the same amount of
memory.

As you migrate your applications from Oracle to DB2 UDB, spend some effort considering how the database should
be laid out in the physical system and, at the same time, consider which table spaces should be buffered in which
buffer pools to get the best hit ratio for your data. It is recommended that you start with a configuration similar to
Oracle, using the same buffer pools with the same table space assignments. Since you can alter the buffer
pool-table space assignments as the system develops and the data access pattern becomes more clear, you can
tune buffer pool performance over time, possibly adding or removing buffer pools.

Cached tables in Oracle should be mapped to tables (and indexes) in dedicated buffer pools large enough to house
the data in memory.

The default configuration established by the CREATE DATABASE command has one buffer pool
(IBMDEFAULTBP) used by all table spaces in the system. This buffer pool should always be expanded before any
serious work is done with the database. Buffer pools can be created, dropped, and changed by using the CREATE
BUFFERPOOL, DROP BUFFERPOOL, and ALTER BUFFERPOOL statements. Table spaces are assigned to
buffer pools by using the ALTER TABLESPACE statement. A buffer pool can have a 4KB (default), 8KB, 16KB, or
32KB page size. The size should be set to match the page size of the table spaces assigned to the buffer pool. If the size of a buffer pool is set to -1 (pages), this indicates that the value of the database configuration parameter BUFFPAGE is to be used for the size of the buffer pool.

```
db2 update db configuration for <dbname> using buffpage 20000
```

```db2 connect to <dbname>
```

```db2 alter bufferpool ibmdefaultbp size -1     -- set IBMDEFAULTBP size to 20000 pages
```

Large objects are not buffered in buffer pools. If the application fetches or updates a LOB column, it always triggers an I/O. Hence, it is important for performance to use a LOB only when the maximum length of a VARCHAR column is not long enough. This maximum is based on the page size; 32762 characters for a 32K page size, for example.

**Import**

Import inserts rows into a table from a file. The following is a typical example of an Import command where the columns are delimited by commas (,), strings are in single quotes (‘), carriage returns are allowed in strings (delprioritychar option), and the table is replaced with the contents of the import file (replace option) dat.tmp.

```
import from dat.tmp
of del
  modified by coldel,
    chardel''
  delprioritychar
usedefaults
commitcount 1000
messages my_table.msg
replace into my_table
;
```

Since the default DATE formats are not the same in Oracle and DB2 (see “DATE and TIME” on page 18), DATE columns exported from Oracle need to be dealt with in a special manner. Here are some approaches:

- Use the Oracle TO_CHAR function to convert the dates to the DB2 default format when writing them to a file.
- Use the MODIFIED BY DATEFORMAT clause to specify a non-standard input format. (You can also use MODIFIED BY TIMEFORMAT and MODIFIED BY TIMESTAMPFORMAT to override the regular formats of those datatypes.)
- Since the Oracle DATE type contains portions of DB2 data types DATE, TIME, and TIMESTAMP, you can use a TIMESTAMP column in DB2. The Import utility then picks up the relevant parts from the input file.

**Load**

Load is the most efficient way of populating the database, generally at least ten times faster than Import or Insert statements, because data pages are formatted directly, without an SQL interface, logging, and the locking of each row. Load can receive data either from a file or a named pipe. The only type of integrity checking performed by Load is for duplicate keys; other types of checking must be performed after the load, by executing the SET INTEGRITY statement, as described in “Constraints” on page 49.

Many recent enhancements have been made to Load; possibly the most relevant to this document is the ability to load non-standard date, time, and timestamp formats, as discussed above for Import. Refer to the Data Movement Utilities Guide and Reference for more information on Load, Import, and other utilities.

For best performance, have DB2 build the indexes during the load (create them before running Load). You should also use one of the variations of the STATISTICS YES option in the Load command. These two techniques allow DB2 to load, index, and gather statistics with only one pass through the data.

Note: While a table is being loaded its tablespace is in “load pending” state, and other tables in the tablespace cannot be accessed. If such table access is needed, one approach is to place the tables in separate tablespaces.
Another approach is to load into a staging table and then insert rows into the real table via “Insert ... Select * from staging_table”.

Redirected Restore

One of the more common problems in database administration is how to reorganize database storage space. In DB2 you cannot drop containers or reduce their size, and for SMS table spaces you cannot add or enlarge containers. (For DMS table spaces you can add and enlarge containers. Both of these are done through ALTER TABLESPACE.) To circumvent any of these restrictions, you can use Redirected Restore to redefine a container’s definitions in a table space during the restore process.

If you specify redirect during the restore process, you can redefine the physical allocation of the table space. For example, you can add a container in a SMS table space, or you can change a DMS table space from file containers to device containers. You can also use redirected restore to shrink your table space by dropping or shrinking containers during the restore process.
Problem Determination

The life of a database administrator gets exciting when there is a problem in the system! Like the regular maintenance tasks of database administration, the diagnostic tools for each database can be quite different. In this section we give a quick outline of DB2 UDB's diagnostic tools and what type of problems each is designed to address. Please consult the DB2 UDB documentation for the full syntax and functionality of each tool. The Control Center allows you to access most of the tools described in this section.

Explain

*Explain* is a database facility for recording and presenting the access plans of queries. In both DB2 UDB and Oracle, the process to generate the access plan is similar. The major difference is that Oracle records the actual cost of the statement at run time, while DB2 UDB records the estimated cost of the SQL statement at compile time.

To enable explain, you first need to create the explain tables (see below), then populate them as you prepare statements. In addition to the explain facility, DB2 UDB provides a graphical tool to view the access plan. The tool is called Visual Explain.

Oracle used to offer only a rule-based optimizer. The cost-based optimizer was introduced in Oracle7. DB2 UDB has always had a cost-based optimizer. What is unique about the DB2 UDB optimizer is its query rewrite capability. Each user SQL statement is rewritten internally before it is passed to the optimization process. The rewrite phase modifies the SQL so it is easier to optimize and can result in superior plans. The user no longer has to worry about how to write the best possible SQL statement. The optimizer shoulders much of the responsibility.

In Oracle, the PLAN_TABLE can be created using the `utlxplan_sql` script and when EXPLAIN <SQL> statement is issued, the table is populated with the database access plan. The cost information is not included. If the server SQL trace is turned on, the tool `TKPROF` can retrieve the costs of the statement:

```sql
(tkprof trace_file username/password explain=yes).
<per session>
  set sql_trace=TRUE;
  ..submit job..
  set sql_trace=false;
  tkprof ... 
```

To view the Oracle access plan, issue the SQL statement:

```sql
select LPAD(' ',2*LEVEL || operation ||
  DECODE(id,0,' Cost = '||position) "Operation",
  options "Options", object_name "object"
  from plan_table where statement_id = 'example'
  connect by prior id = parent_id start with id = 0 
```

DB2 UDB records the estimated cost of a query at preparation time, prior to the actual execution. In fact, you can get the estimated cost without ever executing a query by setting the SNAPSHOT mode to EXPLAIN (see below).

To create the explain tables, use the CLP script `EXPLAIN.DDL` in the `sqlib/misc` directory:

```sql
db2 -tvf EXPLAIN.DDL
```

Once the explain tables are created, access plans for SQL statements can be captured in those tables. These methods are available to capture plans:

Run the EXPLAIN statement. The WITH SNAPSHOT option generates an explain snapshot for use with Visual Explain (see below).

When precompiling or binding a static SQL application, use EXPLAIN ALL (or EXPLSNAP ALL to store an explain snapshot).
In a dynamic SQL application, or in CLP or CLI, issue SET CURRENT EXPLAIN MODE YES before preparing the statement(s) of interest. Or use SET CURRENT EXPLAIN SNAPSHOT YES to store an explain snapshot.

To generate a report from the contents of the explain tables, use the `db2exfmt` command.

Another form of explain output, which does not involve the explain tables, is provided by the `db2expln` and `dynexpln` commands. `db2expln` dumps the access plan for an existing static SQL package, while `dynexpln` takes a statement, or a file of statements as input, binds the statement(s) to the database, then invokes `db2expln`. These commands are useful for quick analysis of statements and do not provide as much information as `db2exfmt`. Both work in interactive mode as well as taking input parameters.

From an ease of use perspective, Visual Explain is far superior to the command line tools `db2exfmt`, `db2expln`, and `dynexpln`. However, the command line tools are valuable when used in scripts.

**Visual Explain**

Visual Explain provides the same information about an access plan as `db2exfmt`, but graphically. It allows you to drill down and see detailed information about each step of the plan, such as the cumulative estimated cost and the estimated number of rows to be processed at that step. It provides the same information as `db2exfmt`.

Visual Explain can be invoked via the Control Center, through these methods and others:

Right-click on a database name, then choose **Explain SQL**. This lets you enter an SQL statement or get statements from a file, then shows the access plan graph.

Right-click on a database name, then choose **Show Explained Statements History**. This shows a list of all statements that have explain information in the explain tables. For those shown as having an explain snapshot, you can choose to display the access plan graph.

To generate the information for Visual Explain, an explain snapshot must be created, using one of the methods listed above in “Explain” on page 63.

After a query goes through the query rewrite phase, its syntax may bear little resemblance to that of the original query. Visual Explain allows you to see the original SQL and the rewritten SQL. For each step in the access plan, the cumulative cost of the plan up to that step is presented, so analysis should begin by looking for the step that has the largest increase in cumulative cost. (Costs are given in units called *timerons*, which reflect the combined CPU, I/O, and communication overhead and not any real unit such as microseconds.) From studying the access plan and its costs you can decide if there are ways to either improve the cost of an operation or remove it. For example, you could add an index to improve the search, or return the result set in the order of an index to possibly eliminate a sort from the access plan.
Database Snapshot Monitor

The snapshot monitor provides the ability to look into the current state of the database. One can capture snapshots for the database manager, database, applications, buffer pools, table spaces, tables, and locks, as well as dynamic SQL. The monitor can be invoked from the Control Center (see the end of this topic), or with the command get snapshot.

The command db2 get snapshot for <...> is often the starting point of an investigation. It gives a quick way of evaluating what is going on in DB2. Some examples:

- db2 get snapshot for locks on <database> tells you which applications are holding which locks, and which applications are currently waiting for a lock to be freed.
- db2 get snapshot for database on <database> gives you a list of indicators on overall database behaviour.
- db2 get snapshot for applications on <dbname> gives detailed status on each connection to the database.

Snapshots for the buffer pools and table spaces are particularly useful in analyzing the I/O hit ratio.

Be sure to reset the snapshot counters at the start of the period for which you want to investigate DB2 activity. To reset counters for the monitor, issue the db2 reset monitor all command. It clears all the counters, and subsequent snapshots return counter values accumulated since the most recent reset or connection.

The amount of data gathered by the snapshot monitor is determined by the switches that have been set. The switches can be set at the instance level by updating the DBM configuration, or at the application level with the update monitor switches command. The switches are: sort, lock, table, buffer pool, unit of work and SQL statement. To turn switches on or off, use the update monitor switches command like this:

```
db2 UPDATE MONITOR SWITCHES USING LOCKS ON
```

To find out which switches are turned on, type:

```
db2 GET MONITOR SWITCHES
```

To find out the default setting of your database instance, type:

```
db2 GET DBM MONITOR SWITCHES
```

The decision on what to gather is based on the overhead of gathering the data and what type of information is required. With all switches turned off, the snapshot monitor still gathers a lot of useful information, enough to diagnose most problems. With all switches turned on, the overhead can be as high as 10 percent. A common practice in production environments is to turn on one switch at a time as you narrow down the problem.

The simplest format for invoking the snapshot monitor with all switches turned off is:

```
db2 GET SNAPSHOT FOR ALL ON <DBNAME>
```

One very common DBA task is to report the slowest performing SQL statements. The DB2 snapshot monitor provides this ability through the DYNAMIC SQL option. When the command is issued with the WRITE TO FILE clause, DB2 UDB writes all the SQL statements in the dynamic statement cache to a file on the server, as well as to the caller. After that, you can use the table function SQLCACHE_SNAPSHOT to identify the problem SQL statements. Reset monitor does not clear the statement cache, only the related counters. Consult the System Monitor Guide and Reference for more details.
% the following command writes the contents of the statement cache to a file
\texttt{db2 get snapshot for dynamic sql on sample write to file}

% the following command calls the UDF \texttt{sysfun.SQLCACHE_SNAPSHOT}. \texttt{Table_name} is an
% arbitrary valid SQL identifier
\begin{verbatim}
  db2 select table_name.db_name, substr(stmt_text,1,200),table_name.num_executions
  from table(sysfun.SQLCACHE_SNAPSHOT()) table_name
  where table_name.commit_sql_stmts > 100
\end{verbatim}

The Control Center provides a graphical snapshot monitoring capability that is quite different from the command-line
based facilities just described. First of all, a number of pre-defined performance monitors are shipped with DB2.
They can be listed by right-clicking on \textbf{Systems} and choosing \textbf{List Monitors}. Here, you can also create your own
monitors. Each monitor includes a number of performance variables (Applications in Lock Wait, for example) that
are each represented as a line on a graph. To control monitoring through the Control Center, right-click on either an
instance or database name, then choose \textbf{Performance Monitoring}.

\section*{Database Event Monitor}

One disadvantage of the snapshot monitor is that it does not capture events that happen before the snapshot is
issued. For example, \texttt{db2 get snapshot for database} may show you that you had three deadlocks since the last
monitor reset, but it does not show more information on the deadlocks unless you issued the snapshot request in the
brief period while the related locks were held but before the deadlock was resolved.

\textit{Event monitors} allow you to monitor and collect data on database activities over a period of time. For example, you
can create an event monitor to capture information on deadlocks, including which applications and which tables are
involved. \textit{An event monitor for statements can also be very useful to identify errors or poorly performing statements,
or simply to find what statements are being executed} (SQL may be generated by a middleware product).

To create an event monitor called \texttt{evmon1} for deadlocks, use a command similar to the following:
\begin{verbatim}
  db2 CREATE EVENT MONITOR evmon1 FOR DEADLOCKS
  WRITE TO FILE ' /EVENTMONITORS/DEADLOCK/EVMON1' MAXFILES 3 MAXFILESIZE 1000
\end{verbatim}

To turn on the event monitor, type:
\begin{verbatim}
  db2 SET EVENT MONITOR evmon1 STATE = 1
\end{verbatim}

To turn off the event monitor, type:
\begin{verbatim}
  db2 SET EVENT MONITOR evmon1 STATE = 0
\end{verbatim}

For each event of the appropriate type(s) that occurs while the event monitor is active, DB2 writes unreadable
binary records to a file in the specified path. To view the information, you can either use the \texttt{db2eva} command
(which brings up a display window but is not available on all platforms) or use the formatting command \texttt{db2evmon}
to write readable records to a file:
\begin{verbatim}
  db2evmon -path /eventmonitors/deadlock/evmon1 > my_evmon1.out  or  
  db2evmon -db db2cert -evm evmon1 > my_evmon1.out
\end{verbatim}
Error Conditions

DB2 UDB stores important messages and error logs in the following places: the `db2diag.log` file, alerts, and the message log. Alerts and the message log were discussed earlier (see “Journal” on page 58) and can be accessed from the Control Center (Journal icon). There is also a user-invoked trace facility. `db2diag.log` and trace cannot be accessed via the Control Center.

**DB2DIAG.LOG**

The `db2diag.log` file is created by DB2 under `sqllib/db2dump` on UNIX, and under `sqllib\db2` on Windows platforms. It contains messages and trace information from DB2 internal functions. Sometimes the amount of information in `db2diag.log` can be overwhelming for DB2 UDB support staff as well as for DB2 users. You can change the amount of information logged in `db2diag.log` by changing the database manager parameter `DIAGLEVEL`, and override the location of `db2diag.log` by changing the database manager parameter `DIAGPATH`. Although the `db2diag.log` contents are difficult to understand, it is definitely the most comprehensive resource when you are trying to debug a non-trivial problem.

The Journal has replaced much of the need to read the `db2diag.log` file, because problems that occur when running the Control Center are typically logged in the Journal in a much more readable format than the corresponding content of `db2diag.log`.

**Trace Facility**

DB2 trace (`db2trc`) is used to diagnose an isolated problem. Once you have isolated an error scenario, you can trace the DB2 UDB logic paths. `db2trc` comes with a heavy overhead because it traces every function call in DB2 UDB. It is best to isolate the scenario as narrowly as possible to reduce the amount of trace information generated by the tool.

The output from `db2trc` is difficult to read unless you know the DB2 function calls. Since it contains the return code from each function, one of the most useful kinds of information you can get quickly is the SQLCODE number in the file, which can help you to find out at what stage the scenario failed. Our experience has been that this information can lead you quickly to a solution.

Because it has heavy overhead, `db2trc` is not suitable for tracing timing problems. There are better tools for this, such as the event monitor.

The default trace buffer is too small for any serious debugging. Always allow at least 1 MB of buffer when you turn on the trace. Run `db2trc` from the command line. `db2trc` produces a binary output, which can be formatted in two ways, ordered by time or by process ID.

```
db2trc on -l 2000000     % turn on the trace with 2 MB buffer
... run the scenario
db2trc dmp tracefile    % capture the trace binary output
db2trc off               % turn off the trace
db2trc fmt tracefile trace.fmt  % sort the formatted output in time order
db2trc flw   tracefile trace.flw % sort formatted trace output sorted by process ID
```
Performance Tuning

A question frequently asked of DB2 support staff is how to make DB2 UDB run faster. Benchmark results have demonstrated that DB2 UDB outperforms Oracle on many workloads, but to tap all the potential of DB2 UDB, a certain amount of tuning might be required to tailor it to your workload and environment.

Before you start working with DB2 UDB tools, take some time to find out whether the performance problem is indeed inside the database. Each operating system provides a set of tools to determine which program is using up system resources (vmstat, iostat, other profiling tools on UNIX, and the Performance Monitor on NT, for example). The information gathered from operating system monitoring tools can give you important clues on where to start looking for the problem.

Why is the database taking so long to process a workload? Is it taking too much CPU time? Is it taking too much I/O time while the system sits idle (I/O wait time)? Is only one of the processors busy, while others sit idle? The following four cases demonstrate how the DB2 diagnostic tools can be used in combination to help you answer such questions.

Case 1

Suppose you find the database is taking a long time to process a query. The operating system monitor tools indicate that the CPU is working overtime. The snapshot monitor for dynamic SQL confirms that it is indeed the bad query that is taking up all the CPU cycles. With this information, the next step is to use explain or Visual Explain to determine if you have a bad access plan. If you verify that the optimizer is picking a poor plan, you may want to do a runstats on the tables accessed, so that the statistics are up-to-date, and perhaps create a new index. In fact, you may wish to use the Index Advisor (“Create Index Using Wizard” in the Control Center) to analyze the query or your entire set of queries and have it recommend indexes that should be created.

Case 2

Again, the database is taking a long time to process a query. The operating system monitor tools indicate that the system is almost idle. The snapshot monitor indicates the application is spending a lot of time waiting for locks. With this information, the next step should be to use the snapshot monitor for locks to determine which locks are under contention and track down the SQL statements competing for the locks. Minor application changes (such as more frequent commits), a change of isolation level, and application redesign are among the possible solutions to lock contention issues.

Case 3

Suppose that for the same symptom, the operating system monitoring tools indicate that the CPU is relatively idle, but there is continual reading from the disks. The snapshot monitor for the table indicates the application is reading most of the table from the disk. This behavior may be caused by one of a few reasons: bad physical layout of the database, a buffer pool that is too small for the table space, or a bad access plan. You can use explain or Visual Explain to verify if you have a bad plan. You can use the snapshot monitor for table spaces to verify if your I/O hit ratio is low. To find out if you have a bad database physical layout, compare the access plan to your physical layout. You may be limiting the amount of I/O parallelism due to the placement of your tables and their indexes.

Case 4

Again, for the same symptom, the operating system monitoring tools indicate that only one of the CPUs in an SMP system is busy. The access plan may indicate SMP parallelism is not turned on. Change the configuration parameters to turn on SMP parallelism and expand the application heap (APPLHEAP) and ASL heap (ASLHEAPSZ) to improve SMP parallelism for the query.

The above four scenarios are used to illustrate how the same symptom can be caused by very different problems, and how DB2 UDB tools can help you to diagnose the problems.
Conclusion

The purpose of this document is to highlight potential issues in porting databases and applications from Oracle to DB2 UDB. It is a living document. As we become more experienced with different porting situations, the document will grow accordingly. If the reader encounters an issue that needs to be addressed but is not covered in this document, please contact the owner, Bill Wilkins, at wilkins@ca.ibm.com. Future readers will benefit from the new information. Thank you.

DB2 Resources

DB2 Universal Database Home Page
http://www.software.ibm.com/data/db2

DB2 Developer Domain
http://www.ibm.com/software/data/developer

Porting from Oracle (or Sybase, or DB2 for Windows, OS/2 and Unix) to DB2 UDB for iSeries
See the porting guides available at http://www.iseries.ibm.com/developer/db2/porting.html

DB2 NewsGroup
news.comp.databases.ibm-db2

IBM PartnerWorld for Developers
http://www.developer.ibm.com

IBM Data Management: Events (particularly Webcasts on migration to DB2 from other RDBMS's)
http://www-3.ibm.com/software/data/events/

Software Migration Project Office -- DB2 Team

Recommended Publication
Using the New DB2: IBM's Object-Relational Database System

DB2 Magazine
http://www.db2mag.com/
Appendix A - DATE data type

ORACLE table HOLIDAY

CREATE TABLE HOLIDAY
(
    Holiday    VARCHAR(25),
    ActualDate DATE,
    CelebratedDate DATE
);

insert into HOLIDAY VALUES ('NEW YEARS DAY',
    TO_DATE('01-JAN-1995','DD-MON-YYYY'),
    TO_DATE('01-JAN-1995','DD-MON-YYYY'))
...;

SELECT * from HOLIDAY
HOLIDAY                     ACTUALDATE      CELEBRATEDDATE
------------------------- ----------      --------------
NEW YEARS DAY 01-JAN-95      01-JAN-95
MARTIN LUTHER KING, JR. 15-JAN-95      16-JAN-95
LINCOLNS BIRTHDAY 12-FEB-95      20-FEB-95
WASHINGTONS BIRTHDAY 22-FEB-95      20-FEB-95
FAST DAY, NEW HAMPSHIRE 22-FEB-95      22-FEB-95
MEMORIAL DAY 30-MAY-95  29-MAY-95
INDEPENDENCE DAY 04-JUL-95 04-JUL-95
LABOR DAY 04-SEP-95 04-SEP-95
COLUMBUS DAY 08-OCT-95 09-OCT-95
THANKSGIVING 23-NOV-95 23-NOV-95

10 record(s) selected.

select Holiday, ActualDate, CelebratedDate
from Holiday
where CelebratedDate - ActualDate != 0
HOLIDAY                          ACTUALDATE CELEBRATEDDATE
-------------------------        ---------- --------------
MARTIN LUTHER KING, JR. 15-JAN-95 16-JAN-95
LINCOLNS BIRTHDAY 12-FEB-95 20-FEB-95
WASHINGTONS BIRTHDAY 22-FEB-95 20-FEB-95
MEMORIAL DAY 30-MAY-95 29-MAY-95
COLUMBUS DAY 08-OCT-95 09-OCT-95

5 record(s) selected.

DB2 UDB Table HOLIDAY

CREATE TABLE HOLIDAY
(
    Holiday    CHAR(25),
    ActualDate DATE,
    CelebratedDate DATE
);

insert into HOLIDAY VALUES ('NEW YEARS DAY', DATE('1995-01-01'), DATE('1995-01-01'))
insert into HOLIDAY VALUES ('MARTIN LUTHER KING, JR.', DATE('1995-01-15'), DATE('1995-01-16'))
insert into HOLIDAY VALUES ('LINCOLNS BIRTHDAY', DATE('1995-02-15'), DATE('1995-02-20'))
insert into HOLIDAY VALUES ('WASHINGTONS BIRTHDAY', DATE('1995-02-22'), DATE('1995-02-20'));
SELECT * from holiday

<table>
<thead>
<tr>
<th>HOLIDAY</th>
<th>ACTUALDATE</th>
<th>CELEBRATEDDATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEW YEARS DAY</td>
<td>01/01/1995</td>
<td>01/01/1995</td>
</tr>
<tr>
<td>MARTIN LUTHER KING, JR.</td>
<td>01/15/1995</td>
<td>01/16/1995</td>
</tr>
<tr>
<td>LINCOLNS BIRTHDAY</td>
<td>02/15/1995</td>
<td>02/20/1995</td>
</tr>
<tr>
<td>WASHINGTONS BIRTHDAY</td>
<td>02/22/1995</td>
<td>02/20/1995</td>
</tr>
<tr>
<td>FAST DAY, NEW HAMPSHIRE</td>
<td>02/22/1995</td>
<td>02/22/1995</td>
</tr>
<tr>
<td>MEMORIAL DAY</td>
<td>05/30/1995</td>
<td>05/29/1995</td>
</tr>
<tr>
<td>INDEPENDENCE DAY</td>
<td>07/04/1995</td>
<td>07/04/1995</td>
</tr>
<tr>
<td>LABOR DAY</td>
<td>09/04/1995</td>
<td>09/04/1995</td>
</tr>
<tr>
<td>COLUMBUS DAY</td>
<td>10/08/1995</td>
<td>10/09/1995</td>
</tr>
</tbody>
</table>

10 record(s) selected.

select Holiday, ActualDate, CelebratedDate
from Holiday
where CelebratedDate - ActualDate != 0;

<table>
<thead>
<tr>
<th>HOLIDAY</th>
<th>ACTUALDATE</th>
<th>CELEBRATEDDATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MARTIN LUTHER KING, JR.</td>
<td>01/15/1995</td>
<td>01/16/1995</td>
</tr>
<tr>
<td>LINCOLNS BIRTHDAY</td>
<td>02/15/1995</td>
<td>02/20/1995</td>
</tr>
<tr>
<td>WASHINGTONS BIRTHDAY</td>
<td>02/22/1995</td>
<td>02/20/1995</td>
</tr>
<tr>
<td>MEMORIAL DAY</td>
<td>05/30/1995</td>
<td>05/29/1995</td>
</tr>
<tr>
<td>COLUMBUS DAY</td>
<td>10/08/1995</td>
<td>10/09/1995</td>
</tr>
</tbody>
</table>

5 record(s) selected.

Here are some errors that may occur in DB2 if the use of Oracle DATE columns is not converted properly.

1. Oracle DATE column is changed to a DB2 TIMESTAMP, but the Insert value (timestamp) does not include the time component:
   db2 insert into mytable values ('2000-08-16') returns an error:
   SQL0180N The syntax of the string representation of a datetime value is incorrect. SQLSTATE=22007
   db2 insert into mytable values ('2000-08-16-13.25.30') is valid.

2. Oracle DATE column becomes a DB2 DATE, but is joined to a TIMESTAMP column in a different table:
   db2 select * from t1, t2 where t1.datecolumn = t2.timestampcolumn returns an error:
   SQL0401N The data types of the operands for the operation "=" are not compatible.

In general, DB2 truncates valid literal timestamps and treats them as valid dates, so this example is OK:

   db2 select * from t1 where datecolumn = '2000-08-16-13.25.30' (returns rows with datecolumn='2000-08-16')
Appendix B - Oracle Identifier Names

As referenced in “Length of DB2 Identifier Names” on page 20, a sample SQL*Plus script is included below to help gather information from the Oracle Data Dictionary and identify all Oracle object names that exceed the 18 character limit imposed by DB2.

```sql
rem ---------------------------------------------------------------------------
rem - Invoke via "sqlplus uid/pwd @filename"
set echo off
set linesize 256
set pagesize 1024
show user
rem ---------------------------------------------------------------------------

prompt Data Dictionary!
rem desc dict;
rem select table_name, substr(comments,1,64) from dict;

prompt Version / Environment!
select * from sm$version;
select * from v$version;
select * from global_name;
select substr(parameter,1,32), substr(value,1,32) from v$nls_parameters;

prompt USER / SCHEMA!
rem get user/schema info
select * from all_users;

prompt INDEXES!
rem get (total #) AND (# objects with names > 18 chars) group by schema
rem + get non-system indexes
select count(*), owner from all_indexes group by owner;
select count(*), owner from all_indexes
   where owner not in('SYS','SYSTEM') and length(index_name) > 18
      group by owner;
select owner, index_name, table_owner, table_name, table_type, uniqueness,
   tablespace_name    from all_indexes
   where owner not in ('SYS','SYSTEM')
      order by owner, table_name;

prompt CONSTRAINTS!
rem get (total #) AND (# objects with names > 18 chars) group by schema
select count(*), owner from all_constraints group by owner;
select count(*), owner from all_constraints
   where owner not in('SYS','SYSTEM') and length(constraint_name) > 18
      group by owner;

prompt TRIGGERS!
rem get (total #) AND (# objects with names > 18 chars) group by schema
select count(*), owner from all_triggers group by owner;
select count(*), owner from all_triggers
   where owner not in('SYS','SYSTEM') and length(trigger_name) > 18
      group by owner;

prompt SEQUENCES!
rem get (total #) group by schema
select count(*), sequence_owner from all_sequences
   group by sequence_owner;
```
Appendix C - ROWID Column / Generate_unique

The Oracle ROWID pseudo-column has no equivalent in DB2. Each use of Oracle ROWID must be evaluated and the DB2 UDB application programs altered based on the use of Oracle ROWID. Two solutions to convert Oracle ROWID to DB2 UDB could be GENERATE_UNIQUE or the primary key. Here is sample code for converting an Oracle ROWID to DB2 GENERATE_UNIQUE with a trigger.

The DB2 GENERATE_UNIQUE function returns a bit data character string that is unique compared to the output of any other execution of the same function. The result of the function is a value that includes a timestamp (the internal form of the Universal Time) and the partition number where the function was processed. The latter part guarantees that a table partitioned across multiple partitions also has unique values.

If the application design and database permit, a DB2 UDB primary key can be used to replace the Oracle ROWID. Analysis and testing should be done to verify that the primary key solution is valid across tables and the database. See “Sequence Objects” on page 22 for information on using sequence objects to generate primary key values.

Oracle Sample Code:

```sql
CREATE TABLE EMP_UPDATE
(
    EMPNO             CHAR(6),
    LASTNAME CHAR(15),
    PRIMARY KEY (EMPNO)
);

INSERT INTO EMP_UPDATE (EMPNO, LASTNAME)
VALUES ('000001','Smith');

INSERT INTO EMP_UPDATE (EMPNO, LASTNAME)
VALUES ('000002','Jones');

SELECT ROWID, EMPNO, LASTNAME
FROM  EMP_UPDATE;

ROWID    EMPNO     LASTNAME
------------------   ------    ---------------
AAAAfDAAEAAAAAOvAAA   000001    Smith
AAAAfDAAEAAAAAOvAAB   000002    Jones

SELECT LASTNAME
FROM EMP_UPDATE
WHERE ROWID = 'AAAAfDAAEAAAAAOvAAA';

LASTNAME
----------
Smith
```
DB2 UDB Sample Code:

CREATE TABLE EMP_UPDATE
(
    UNIQUE_ID CHAR(13) FOR BIT DATA,
    EMPNO CHAR(6) NOT NULL,
    LASTNAME CHAR(15),
    PRIMARY KEY(EMPNO)
);

CREATE TRIGGER EMP_UPDATE_UNIQUE
    NO CASCADE BEFORE INSERT ON EMP_UPDATE
    REFERENCING NEW AS NEW_UPD
    FOR EACH ROW MODE DB2SQL
    SET NEW_UPD.UNIQUE_ID = GENERATE_UNIQUE();

INSERT INTO EMP_UPDATE (EMPNO, LASTNAME)
VALUES ('000001', 'Smith'),
       ('000002', 'Jones');

SELECT * FROM EMP_UPDATE;

<table>
<thead>
<tr>
<th>UNIQUE_ID</th>
<th>EMPNO</th>
<th>LASTNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>X'19990114195748114225000000'</td>
<td>000001</td>
<td>Smith</td>
</tr>
<tr>
<td>X'19990114195748124402000000'</td>
<td>000002</td>
<td>Jones</td>
</tr>
</tbody>
</table>

2 record(s) selected.

SELECT LASTNAME
FROM EMP_UPDATE
WHERE UNIQUE_ID = X'19990114195748114225000000';

LASTNAME
---------------
Smith

1 record(s) selected.

The following is sample code that demonstrates the use of a DB2 primary key to retrieve a row in a Select statement:

SELECT LASTNAME
FROM EMP_UPDATE
WHERE EMPNO = '000001'

LASTNAME
---------------
Smith

1 record(s) selected.
Appendix D - Sample DB2 SQL Procedure

This sample (nestcase.db2) is taken from the DB2 SQL Procedure sample directory (sqllib/samples/sqlproc). It is a CLP script that creates an SQL procedure.

To create the SQL procedure using this script, perform the following steps:
1. Connect to the database.
2. Issue the command `db2 -td@ -vf nestcase.db2`.

To call this SQL procedure from the command line, perform the following steps:
1. Connect to the database.
2. Issue the command `db2 "CALL bump_salary (51)"`.

The sample `nestcase.sqc` demonstrates how to call this SQL procedure using an embedded C client application.

```sql
CREATE PROCEDURE bump_salary (IN deptnumber SMALLINT)
LANGUAGE SQL
BEGIN
    DECLARE SQLSTATE CHAR(5);
    DECLARE v_salary DOUBLE;
    DECLARE v_id SMALLINT;
    DECLARE v_years SMALLINT;
    DECLARE at_end INT DEFAULT 0;
    DECLARE not_found CONDITION FOR SQLSTATE '02000';

    DECLARE C1 CURSOR FOR
        SELECT id, CAST(salary AS DOUBLE), years
        FROM staff
        WHERE dept = deptnumber;
    DECLARE CONTINUE HANDLER FOR not_found
        SET at_end = 1;
    OPEN C1;
    FETCH C1 INTO v_id, v_salary, v_years;
    WHILE at_end = 0 DO
        CASE
            WHEN (v_salary < 2000 * v_years) THEN UPDATE staff
                SET salary = 2150 * v_years
                WHERE id = v_id;
            WHEN (v_salary < 5000 * v_years) THEN CASE
                WHEN (v_salary < 3000 * v_years) THEN UPDATE staff
                    SET salary = 3000 * v_years
                    WHERE id = v_id;
                ELSE UPDATE staff
                    SET salary = v_salary * 1.10
                    WHERE id = v_id;
                END CASE;
            ELSE UPDATE staff
                SET job = 'PREZ'
                WHERE id = v_id;
        END CASE;
        FETCH C1 INTO v_id, v_salary, v_years;
    END WHILE;
    CLOSE C1;
END @
```
Appendix E - Sample DB2 User-Defined Functions (UDFs)

Here are some sample DB2 user-defined SQL scalar functions.

The body of the next_day function is a fullselect. It implements the Oracle next_day function, which takes a date and a day of the week in character form, and returns as a date the first occurrence of the specified day of the week that is later than the input date. The to_date function definition also contains a fullselect, and is a partial implementation of the functionality of the Oracle to_date function.

The body of the sum_range function is a dynamic compound SQL statement, which is a new feature in V7.2.

Create a function for converting a day name to a day code as returned by the DB2 dayofweek function. We make use of this day code in the next_day function below.

```sql
CREATE FUNCTION day_text_to_int (D VARCHAR(9))
RETURNS INTEGER
LANGUAGE SQL
CONTAINS SQL
NO EXTERNAL ACTION
DETERMINISTIC
RETURN CASE UCASE(D)
WHEN 'SUNDAY' THEN 1
WHEN 'MONDAY' THEN 2
WHEN 'TUESDAY' THEN 3
WHEN 'WEDNESDAY' THEN 4
WHEN 'THURSDAY' THEN 5
WHEN 'FRIDAY' THEN 6
WHEN 'SATURDAY' THEN 7
END;
```

Next, create the next_day function itself.

```sql
CREATE FUNCTION next_day (D DATE, Y VARCHAR(9))
RETURNS DATE
LANGUAGE SQL
CONTAINS SQL
NO EXTERNAL ACTION
DETERMINISTIC
RETURN CASE
WHEN ((dayofweek(D) - day_text_to_int(Y)) >= 0)
THEN (D + (7 - (dayofweek(D) - day_text_to_int(Y))) DAYS)
ELSE (D + (day_text_to_int(Y) - dayofweek(D)) DAYS)
END;
```

The functions can be tested by running examples such as the following using the Command Line Processor:

```sql
db2 values (next_day(current date, 'Tuesday'))
db2 values (next_day(date('11/01/2000'), 'Monday')) [ output is 11/06/2000 ]
```

Create a DB2 user-defined function to perform a subset of the Oracle to_date function. This function ONLY supports a pattern of the form 'yyyy-mm-dd hh24:mi:ss', as in “to_date( '2001-08-24 15:25:40', 'yyyy-mm-dd hh24:mi:ss' )”. Also, it assumes that the input is valid.

```sql
create function to_date(date_time char(19), pattern char(21)) returns timestamp
Deterministic no external action contains sql
return
  case when upper(pattern) = 'YYYY-MM-DD HH24:MI:SS' then
    timestamp( substr(date_time,1,10), substr(date_time,12,8) )
  end;
```
Sample command line invocation:

```
db2 values (to_date(char('2001-08-24 15:25:40'), char('yyyy-mm-dd hh24:mi:ss')))  
(output: 2001-08-24-15.25.40.000000)
```

**************************************************************************************************************************************

Create a function that sums all of the integers between the two integer input parameter values (inclusive)

```sql
CREATE FUNCTION sumrange (X INT, Y INT) RETURNS BIGINT 
    LANGUAGE SQL 
    CONTAINS SQL 
    NO EXTERNAL ACTION 
    DETERMINISTIC 
BEGIN ATOMIC 
    DECLARE SUM, COUNTER BIGINT DEFAULT 0; 
    IF X < Y THEN SET COUNTER = X; 
    WHILE COUNTER <= Y DO 
        SET SUM = SUM + COUNTER; 
        SET COUNTER = COUNTER + 1; 
    END WHILE; 
    ELSEIF X > Y THEN SET COUNTER = Y; 
    WHILE COUNTER <= X DO 
        SET SUM = SUM + COUNTER; 
        SET COUNTER = COUNTER + 1; 
    END WHILE; 
    ELSE -- X=Y 
        SET SUM = X; 
    END IF; 
    RETURN SUM; 
END @
```

The function can be tested by running examples such as the following using the Command Line Processor:

```
db2 values sumrange(5,7)  [output is 18] 
db2 values sumrange(2,-4)  [output is -7] 
db2 values sumrange(12,12) [output is 12] 
```
Appendix F - Sample DB2 Java Stored Procedure

This sample is taken from the DB2 java sample directory (sqllib/samples/java). It illustrates how a java stored procedure can be used to retrieve a multiple-row result set.

Stored procedure itself: MRSPsrv.java

```java
import java.lang.*;
import java.util.*;
import java.io.*;
import java.sql.*;              // JDBC classes
import java.math.*;             // BigDecimal
import COM.ibm.db2.jdbc.app.*;  // DB2 UDB JDBC classes
import COM.ibm.db2.app.*;       // StoredProc and associated classes

////////////////////////////////////////////////////////////////////////
// Java stored procedure which takes 3 input String/Character parameters
// Each parameter is the SQL statement to be executed.
// ////////////////////////////////////////////////////////////////////////
class MRSPsrv extends StoredProc {
    public void MRSPsrv (String q1, String q2, String q3)
        throws Exception
    {
        try
        {
            // get caller's connection to the database; inherited from StoredProc
            Connection con1 = getConnection();

            // create and execute statement based on 1st parameter passed in
            Statement stmt1 = con1.createStatement();
            stmt1.execute(q1);

            // create and execute statement based on 2nd parameter passed in
            Statement stmt2 = con1.createStatement();
            stmt2.execute(q2);

            // create and execute statement based on 3rd parameter passed in
            Statement stmt3 = con1.createStatement();
            stmt3.execute(q3);

            // explicitly close the Connection object; otherwise, if
            // the database manager configuration variable KEEPARI = YES,
            // DB2 may not clean up Connection resources
            con1.close();
        }
        catch (Exception e)
        {
            throw e;
        }
    }
}
```
Java client code to invoke stored procedure: MRSPcli.java

```java
import java.lang.*;
import java.util.*;
import java.io.*;
import java.sql.*;              // JDBC classes
import COM.ibm.db2.jdbc.app.*;  // DB2 UDB JDBC classes

class MRSPcli
{
  static
  {
    try
    {
      Class.forName("COM.ibm.db2.jdbc.app.DB2Driver").newInstance();
    }
    catch (Exception e)
    {
      System.out.println("\n Error loading DB2 Driver...\n");
      System.out.println(e);
      System.exit(1);
    }
  }

  // main application: .connect to the database
  //                   .register the stored procedure
  //                   .call the stored procedure
  public static void main (String argv[])
  {
    Connection con = null;
    try
    {
      System.out.println("Java Multiple Resultsets Stored Procedure Sample");
      // Connect to Sample database
      // URL is jdbc:db2:dbname
      String url = "jdbc:db2:sample";

      if (argv.length == 0)
      {
        // connect with default id/password
        con = DriverManager.getConnection(url);
      }
      else if (argv.length == 2)
      {
        String userid = argv[0];
        String passwd = argv[1];
        // connect with user-provided username and password
        con = DriverManager.getConnection(url, userid, passwd);
      }
      else
      {
        throw new Exception("Usage: java MRSPcli [username password]\n");
      }

      // Set AutoCommit
      con.setAutoCommit(true);

      Statement stmt = con.createStatement();

      String callName = "MRSPsrv";
      String storedProcName = "MRSPsrv!MRSPsrv";
      String mode = "fenced";

      try
      {
        // drop the stored procedure if it exists
        stmt.executeUpdate("DROP PROCEDURE " + callName);
      }
      catch (SQLException e)
      {
        // ignore this error
      }
    }
  }
}
```
try
{
    // define the parameters for the Stored Procedure
    String parameterList =
        "(in q1 varchar(200), in q2 varchar(200), in q3 varchar(200))";

    // construct the create procedure statement
    String cpStr =  "CREATE PROCEDURE " + callName + " " + parameterList
                    + " LANGUAGE JAVA " +
                    " PARAMETER STYLE DB2GENERAL " + mode +
                    " EXTERNAL NAME " + storedProcName + "";
    System.out.println("\nRegistering Java Stored Procedure... \n");
    // execute the create statement
    stmt.executeUpdate (cpStr);
}
catch (SQLException e)
{
    System.out.println ("\nError received registering stored proc");
    throw e;
}

// create the 3 sql statements to be executed in the Java Stored Proc
String q1 = "select ID, NAME, DEPT, JOB from STAFF where SALARY > 20000.00";
String q2 = "select EMPNO, FIRSTNME from EMPLOYEE where SALARY > 20000.00";
String q3 = "select DEPTNO, DEPTNAME from DEPARTMENT";

// create callable statement to call the Stored Procedure
String callSql = "Call " + callName + "(?, ?, ?)";
System.out.println("Creating CallableStatement = " + callSql);
CallableStatement callStmt = con.prepareCall(callSql);
// set the parameters required for the Stored Procedure
System.out.println(" Param.1 = " +q1);
callStmt.setString (1, q1);
System.out.println(" Param.2 = " +q2);
callStmt.setString (2, q2);
System.out.println(" Param.3 = " +q3);
callStmt.setString (3, q3);

ResultSet rs = null;
System.out.println("\nExecuting the Java Stored Procedure now...\n");
callStmt.execute();
int rsCount = 0;

while( true )
{
    int rowCount = callStmt.getUpdateCount();
    if( rowCount > 0 )
    {
        System.out.println("================================================================");
        System.out.println("Rows changed = " + rowCount);
        callStmt.getMoreResults();
        System.out.println();
        continue;
    }
    if( rowCount == 0 )
    {
        System.out.println("================================================================");
        System.out.println("No rows changed or sql was DDL command");
        callStmt.getMoreResults();
        System.out.println();
        continue;
    }
}
rs = callStmt.getResultSet();
if( rs != null )
{
    rsCount++;
    System.out.println("Fetching all the rows from the result set "+rsCount);
    fetchAll(rs);
    callStmt.getMoreResults();
    System.out.println();
    continue;
}

break;

// close off everything before we leave
System.out.println("Closing statements and connection.");
callStmt.close();
stmt.close();
con.close();
}
catch (Exception e)
{
    try
    {
        if( con != null )
        {
            con.close();
        }
    }
    catch (Exception x)
    {
        //ignore this exception
    }
    System.out.println (e);
}

// =====================================================
// Method: fetchAll
// ==============================================================
static public void fetchAll( ResultSet rs )
{
    try
    {
        System.out.println("=================================================================");
        ResultSetMetaData stmtInfo = rs.getMetaData();
        int numOfColumns = stmtInfo.getColumnCount();
        int r = 0;
        while( rs.next() )
        {
            r++;
            System.out.print("Row: " + r + ": ");
            for( int i=1; i <= numOfColumns; i++ )
            {
                System.out.print(rs.getString(i));
                if( i != numOfColumns ) System.out.print(", ");
            }
            System.out.println("");
        }
    }
    catch (SQLException e)
    {
        System.out.println("Error: fetchALL: exception");
        System.out.println (e);
    }
}
Appendix G - Sample DB2 Static SQL Program

As mentioned in "Embedded Static SQL" on page 42, a sample DB2 embedded static SQL application (taken from sqllib/samples/c/openftch.sqc) is included here.

****************************************************************************
**
** Source File Name = openftch.sqc
**
** Licensed Materials - Property of IBM
**
** (C) COPYRIGHT International Business Machines Corp. 1995, 2000
** All Rights Reserved.
**
** US Government Users Restricted Rights - Use, duplication or
** disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
**
** PURPOSE: This sample program demonstrates the use of a CURSOR.
** The CURSOR is processed using static SQL. This program
** obtains all managers in the STAFF table of the
** SAMPLE database and change their job from "Mgr" to
** "Clerk" or DELETE the row dependent upon the "dept" value.
**
** A ROLLBACK is done so that the SAMPLE database remains
** unchanged.
**
** An external function "check_error" is contained in the file "util.c"
** which must be compiled along with this file.
**
** EXTERNAL DEPENDENCIES :
**
** - Ensure existence of database for precompile purposes.
** - Precompile with the SQL precompiler (PREP in DB2)
** - Bind to a database (BIND in DB2)
** - Compile and link with the IBM Cset++ compiler (AIX and OS/2)
** or the Microsoft Visual C++ compiler (Windows)
** or the compiler supported on your platform.
**
** For more information about these samples see the README file.
**
** For more information on programming in C, see the:
** - "Programming in C and C++" section of the Application Development Guide
** For more information on building C applications, see the:
** - "Building C Applications" section of the Application Building Guide.
**
** For more information on the SQL language see the SQL Reference.
**
*******************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "utilemb.h"

EXEC SQL INCLUDE SQLCA;

int main(int argc, char *argv[]) {

    EXEC SQL BEGIN DECLARE SECTION;
    char   pname[10];
    short  dept;
    char userid[9];

}
char passwd[19];
EXEC SQL END DECLARE SECTION;

printf( "Sample C program: OPENFTCH\n" );

if (argc == 1) {
    EXEC SQL CONNECT TO sample;
    EMB_SQL_CHECK("CONNECT TO SAMPLE");
} else if (argc == 3) {
    strcpy (userid, argv[1]);
    strcpy (passwd, argv[2]);
    EXEC SQL CONNECT TO sample USER :userid USING :passwd;
    EMB_SQL_CHECK("CONNECT TO SAMPLE");
} else {
    printf ("\nUSAGE: openftch [userid passwd]\n\n" );
    return 1;
} /* endif */

EXEC SQL DECLARE c1 CURSOR FOR /* :rk.1:erk. */
SELECT name, dept FROM staff WHERE job='Mgr'
FOR UPDATE OF job;

EXEC SQL OPEN c1; /* :rk.2:erk. */
EMB_SQL_CHECK("OPEN CURSOR");

do {
    EXEC SQL FETCH c1 INTO :pname, :dept; /* :rk.3:erk. */
    if (SQLCODE != 0) break;
    if (dept > 40) {
        printf( "%-10.10s in dept. %2d will be demoted to Clerk\n", 
            pname, dept );
        EXEC SQL UPDATE staff SET job = 'Clerk' /* :rk.4:erk. */
            WHERE CURRENT OF c1;
        EMB_SQL_CHECK("UPDATE STAFF");
    } else {
        printf("%-10.10s in dept. %2d will be DELETED!\n", 
            pname, dept);
        EXEC SQL DELETE FROM staff WHERE CURRENT OF c1;
        EMB_SQL_CHECK("DELETE");
    } /* endif */
} while ( 1 );

EXEC SQL CLOSE c1; /* :rk.5:erk. */
EMB_SQL_CHECK("CLOSE CURSOR");

EXEC SQL ROLLBACK;
EMB_SQL_CHECK("ROLLBACK");
printf("\nOn second thought -- changes rolled back.\n" );

EXEC SQL CONNECT RESET;
EMB_SQL_CHECK("CONNECT RESET");
return 0;
} /* end of program : OPENFTCH.SQC */
Appendix H - Sample DB2 Dynamic Embedded Application

As mentioned in “Embedded Dynamic SQL” on page 43, a sample DB2 embedded dynamic SQL application is included below. The C application accepts any SQL statement. If the statement is not a query, an attempt is made to EXECUTE IMMEDIATE the statement. If the statement is a query, then the query is executed with the appropriate cursor declarations; the SQLDA is used to determine the number of items in the select-list. For simplicity, only integer data types are allowed in the select-list in this example.

```c
/* #include / #define */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sqlenv.h>

EXEC SQL INCLUDE SQLCA;
EXEC SQL INCLUDE SQLDA;
#define SQLSTATE           sqlca.sqlstate
#define CHECKERR(CE_STR)   if ( check_error(CE_STR, &sqlca) != 0)  \
                         exit(1);

/* global variables */
EXEC SQL BEGIN DECLARE SECTION;
    char dbname[9];
    char stmt[80];
EXEC SQL END DECLARE SECTION;

/* function prototypes */
void main(int argc, char * argv[]);
int check_error(char eString[], struct sqlca *caPointer);
void method4();

/* function main */
void main(int argc, char * argv[]) {
    if (argc != 3) {
        printf("USAGE:  %s <database> <sql stmt> \n", argv[0]);
        exit(99);
    }
    strcpy (dbname, argv[1]);
    strcpy (stmt, argv[2]);
    EXEC SQL CONNECT TO :dbname;
    CHECKERR("connect");

    method4();

    EXEC SQL CONNECT RESET;
    CHECKERR("connect reset");
}

/* function method4 */
void method4() {

    struct sqlda * sqldaptr;    /* ptr to sqlda structure */
    short  numcols;             /* temp var to hold # of cols in */
                                 /* select-list after 1st describe */

    short  i;                   /* counter var */
    int    rownum;              /* counter to denote row # being printed */
```
EXEC SQL PREPARE s4 FROM :stmt;
CHECKERR("prepare s4 from :stmt");

/* allocate "minimal" sqlda AND describe stmt */
if ((sqldaptr = (struct sqlda *)malloc(SQLDASIZE(1))) == NULL) {
    printf("Could not allocate SQLDA of 1 \n");
    exit (99);
}

sqldaptr->sqln=1;
EXEC SQL DESCRIBE s4 INTO :*sqldaptr;
printf("describe #1 SQLCODE = %d \t sqln = %d \t sqld = %d \n",
        SQLCODE, sqldaptr->sqln, sqldaptr->sqld);
if ((SQLCODE != 0) && (SQLCODE != 236))
    CHECKERR("describe s4 into :*sqldaptr");

/* if sqlda NOT large enough, allocate new sqlda AND describe stmt */
/* sqlcode 236 = sqlstate 01005 */
if (SQLCODE == 236) {
    numcols=sqldaptr->sqld;
    free(sqldaptr);
    if ((sqldaptr = (struct sqlda *)malloc(SQLDASIZE(numcols))) == NULL) {
        printf("Could not allocate SQLDA of %d \n", sqldaptr->sqld);
        exit (99);
    }
    sqldaptr->sqln=numcols;
    EXEC SQL DESCRIBE s4 INTO :*sqldaptr;
    printf("describe #2 SQLCODE = %d \t sqln = %d \t sqld = %d \n",
            SQLCODE, sqldaptr->sqln, sqldaptr->sqld);
    if ((SQLCODE != 0) && (SQLCODE != 236))
        { CHECKERR("describe s4 into :*sqldaptr"); }
}

/* if NON-select stmt, execute and exit */
if (sqldaptr->sqld == 0) {
    printf("NOT a select stmt - executing! \n");
    EXEC SQL EXECUTE IMMEDIATE :stmt;
    CHECKERR("execute immediate :stmt");
    EXEC SQL COMMIT WORK;
    CHECKERR("commit work (NON-select)");
    free(sqldaptr);
    return;
}

/* sqldaptr contains type/length info for each column in select-list - cycle */
/* through info on each column to allocate storage for sqldata/sqlind pointers */
/* Note that switch stmt only handles integer columns - this can be expanded to */
/* include additional SQL data types */
for (i=0; i<sqldaptr->sqld; i++) {
    printf("sqldaptr->sqlvar[%d].sqltype= %d \t sqldaptr->sqlvar[%d].sqllen = %d \n", i, sqldaptr->sqlvar[i].sqltype, i, sqldaptr->sqlvar[i].sqllen);
    switch(sqldaptr->sqlvar[i].sqltype) {
        case SQL_TYP_INTEGER:
        case SQL_TYP_NINTEGER:
            sqldaptr->sqlvar[i].sqldata=(char *)malloc(sqldaptr->sqlvar[i].sqllen);
            break;
        default:
            break;
    } /* switch */
    if (sqldaptr->sqlvar[i].sqldata == NULL) {
        printf("Could not allocate sqlvar[%d].sqldata mem \n", i);
        exit (-1);
memset (sqldaptr->sqlvar[i].sqldata, '\0', sizeof(short));
/* an odd value for sqltype implies that the datatype is */
/* nullable - if so allocate storage for sqlind pointer */
if (sqldaptr->sqlvar[i].sqltype & 1) {
    sqldaptr->sqlvar[i].sqlind = (char *)malloc(sizeof(short));
    if (sqldaptr->sqlvar[i].sqlind == NULL) {
        printf(" Could not allocate sqlvar[%d].sqlind mem \n", i);
        exit (-1);
    } else
        memset (sqldaptr->sqlvar[i].sqlind, '\0', sizeof(short));
} /* if */
} /* for */

EXEC SQL DECLARE c4 CURSOR FOR s4;
EXEC SQL OPEN c4;
CHECKERR("open cursor c4");
rownum=1;
do {
    EXEC SQL FETCH c4 USING DESCRIPTOR :*sqldaptr;
    CHECKERR("fetch c4");
    if (SQLCODE != 100) {
        printf(" # %3d >> ", rownum);
        for (i=0; i<sqldaptr->sqld; i++) {
            printf(" %2d \t", *(int *)sqldaptr->sqlvar[i].sqldata);
        }
        printf(" \n");
        rownum++;
    }
} /* do while SQLCODE != 100 */
while (SQLCODE != 100);
EXEC SQL CLOSE c4;
CHECKERR("close cursor c4");
printf("##########################################################################\n");
EXEC SQL COMMIT WORK;
CHECKERR("commit work");
free(sqldaptr);
return 0;

/* function check_error */
int check_error (char eString[], struct sqlca *caPointer)
{
    char eBuffer[1024];
    char sBuffer[1024];
    short rc, Erc;

    if (((caPointer->sqlcode == 0) || (caPointer->sqlcode == 100))
        return 0;
    else { /* SQLERROR or SQLWARNING */
        printf("--- error report ---\n");
    }
}
printf("ERROR occurred : \%s\nSQLCODE : \%ld\n", eString, caPointer->sqlcode);

Erc = sqlaintp(eBuffer, 1024, 80, caPointer);
if (Erc > 0) printf("\%s", eBuffer);
else printf("sqlaintp rc = \%d\n", Erc);

rc = sqlogstt(sBuffer, 1024, 80, caPointer->sqlstate);
if (rc > 0) printf("\n\%s", sBuffer);
else printf("sqlogstt rc = \%d\n", rc);

if (caPointer->sqlcode < 0) {
    printf("--- end error report ---\n");
    return 1;
} else {
    printf("--- end error report ---\n");
    printf("WARNING - CONTINUING PROGRAM WITH WARNINGS!\n");
    return 0;
}

} /* else - SQLERROR or SQLWARNING */

} /* fn check_error */
Appendix I - Mapping of Oracle8 OCI to DB2 CLI

The purpose of this appendix is to give a mapping of the most important Oracle8 OCI (Oracle Call Interface) calls to the closest DB2 CLI (Call Level Interface) equivalents. Refer to the DB2 Call Level Interface Guide and Reference for details on the CLI calls. Numbers in parentheses refer to the corresponding notes below the tables. “N/A” means that the OCI call has no equivalent in CLI.

<table>
<thead>
<tr>
<th>Connect / Initialize / Authorize</th>
<th>Bind / Define / Describe</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCIInitialize</td>
<td>OCIBindDynamic</td>
</tr>
<tr>
<td>OCIEnvInit</td>
<td>SQL__Data (11)</td>
</tr>
<tr>
<td>OCIServerAttach</td>
<td>OCIBindByObject</td>
</tr>
<tr>
<td>OCIEnvAttach</td>
<td>N/A (12)</td>
</tr>
<tr>
<td>OCIEnvBegin</td>
<td>OCIBindArrayOfStruct</td>
</tr>
<tr>
<td>OCIEnvEnd</td>
<td>SQLBindParameter (13)</td>
</tr>
<tr>
<td>OCILogon</td>
<td>OCIBindArrayOfStruct</td>
</tr>
<tr>
<td>OCILogoff</td>
<td>SQLBindCol (15)</td>
</tr>
<tr>
<td>OCILogoff</td>
<td>OCIBindByPos</td>
</tr>
<tr>
<td>OCILogoff</td>
<td>N/A (12)</td>
</tr>
<tr>
<td>OCILogoff</td>
<td>OCIDescribeAny</td>
</tr>
<tr>
<td>OCILogoff</td>
<td>(many) (16)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Handles / Descriptors</th>
<th>Prepare / Execute / Fetch</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCIRawHandleAlloc</td>
<td>OCIStmtPrepare</td>
</tr>
<tr>
<td>OCIRawHandleFree</td>
<td>SQLPrepare</td>
</tr>
<tr>
<td>OCIRawHandleFree</td>
<td>OCIStmtExecute</td>
</tr>
<tr>
<td>OCIRawHandleFree</td>
<td>SQLExecute</td>
</tr>
<tr>
<td>OCIRawHandleFree</td>
<td>OCIStmtFetch</td>
</tr>
<tr>
<td>OCIRawHandleFree</td>
<td>SQLFetch (17)</td>
</tr>
<tr>
<td>OCIRawHandleFree</td>
<td></td>
</tr>
<tr>
<td>OCIRawHandleFree</td>
<td></td>
</tr>
<tr>
<td>OCIRawHandleFree</td>
<td></td>
</tr>
<tr>
<td>OCIRawHandleFree</td>
<td></td>
</tr>
<tr>
<td>OCIRawHandleFree</td>
<td></td>
</tr>
<tr>
<td>OCIRawHandleFree</td>
<td></td>
</tr>
<tr>
<td>OCIRawHandleFree</td>
<td></td>
</tr>
<tr>
<td>OCIRawHandleFree</td>
<td></td>
</tr>
<tr>
<td>OCIRawHandleFree</td>
<td></td>
</tr>
<tr>
<td>OCIRawHandleFree</td>
<td></td>
</tr>
<tr>
<td>OCIRawHandleFree</td>
<td></td>
</tr>
<tr>
<td>OCIRawHandleFree</td>
<td></td>
</tr>
<tr>
<td>OCIRawHandleFree</td>
<td></td>
</tr>
<tr>
<td>OCIRawHandleFree</td>
<td></td>
</tr>
<tr>
<td>OCIRawHandleFree</td>
<td></td>
</tr>
<tr>
<td>OCIRawHandleFree</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Transaction Management</th>
<th>Miscellaneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCITransCommit</td>
<td>OCIBreak</td>
</tr>
<tr>
<td>OCITransCommit</td>
<td>SQLCancel</td>
</tr>
<tr>
<td>OCITransCommit</td>
<td>OCIServerVersion</td>
</tr>
<tr>
<td>OCITransCommit</td>
<td>SQLGetInfo (18)</td>
</tr>
<tr>
<td>OCITransCommit</td>
<td>OCIPasswordChange</td>
</tr>
<tr>
<td>OCITransCommit</td>
<td>N/A</td>
</tr>
<tr>
<td>OCITransCommit</td>
<td>OCIErrorGet</td>
</tr>
<tr>
<td>OCITransCommit</td>
<td>SQLGetDiagRec</td>
</tr>
<tr>
<td>OCITransCommit</td>
<td>OCISqlStmtGetProc</td>
</tr>
<tr>
<td>OCITransCommit</td>
<td>SQLGetData (11)</td>
</tr>
<tr>
<td>OCITransCommit</td>
<td>OCISqlStmtSetProc</td>
</tr>
<tr>
<td>OCITransCommit</td>
<td>SQL_Data (11)</td>
</tr>
<tr>
<td>OCITransCommit</td>
<td>OCILdaToSvCxt</td>
</tr>
<tr>
<td>OCITransCommit</td>
<td>N/A</td>
</tr>
<tr>
<td>OCITransCommit</td>
<td>OCISvcCxtToLda</td>
</tr>
<tr>
<td>OCITransCommit</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Notes for the tables above:

1. SQLAllocHandle is passed the desired handle type: environment, connection, statement, or descriptor.
2. SQLDriverConnect is an alternative to SQLConnect, providing additional parameters.
3. OCISessionBegin has no CLI equivalent. To establish multiple database connections in CLI, multiple connection handles must be allocated and OCIServerAttach calls replaced by SQLConnect calls.
4. OCIAttrGet can be replaced by SQLGetConnectAttr, SQLGetEnvAttr, or SQLGetStmtAttr, depending on the type of handle that the attribute value is wanted for.
5. `SQLSetStmtAttr` must be called with an Attribute value of `SQL_ATTR_APP_PARAM_DESC` or `SQL_ATTR_APP_ROW_DESC`. However, descriptors can be allocated implicitly instead. The function of `OCIParamGet` is performed by `SQLSetStmtAttr` (or implicitly).

6. CLI does not have complex object retrieval (COR) descriptors or handles.

7. `OCIAttrSet` can be replaced by `SQLSetConnectAttr`, `SQLSetEnvAttr`, or `SQLSetStmtAttr`, depending on the type of handle for which an attribute value is to be set.

8. `SQLFreeHandle` must be called with a HandleType of `SQL_HANDLE_DESC` (or the connection can be freed).

9. `SQLEndTran` does either a Commit or a Rollback, depending on the completion type parameter value.

10. There are no CLI calls specifically for two-phase Commit, but it is supported (see CLI Guide and Reference).

11. For piecewise operations there is no direct replacement for `OCIBindDynamic`, but for inserts, `SQLParamData` and `SQLPutData` must be called, and for selects, `SQLGetData`.

12. CLI has no special support for user-defined types. CLI treats one as it does the underlying built-in type. In SQL statements the CAST function must be used to convert a parameter with a user-defined type to the corresponding built-in type (or vice-versa). For more information, see the topic “User Defined Types in Predicates” in the CLI Guide and Reference.

13. To use array inserts, `SQLBindParameter` must be called. In addition, calls to `SQLSetStmtAttr` are needed to set attributes `SQL_ATTR_PARAMSET_SIZE` (array size) and `SQL_ATTR_PARAM_BIND_TYPE` (row-wise or column-wise binding of parameters). `OCIDefineArrayOfStruct` calls can be ignored.

14. No direct equivalent, but `SQLDescribeParam` is the closest.

15. To use array fetches, `SQLBindCol` must be called. In addition, calls to `SQLSetStmtAttr` are needed to set attributes `SQL_ATTR_ROW_ARRAY_SIZE` (array size) and `SQL_ATTR_ROW_BIND_TYPE` (row-wise or column-wise array retrieval).

16. An `OCIDescribeAny` call should be replaced by the appropriate call from among `SQLColAttribute`, `SQLColumns`, `SQLDescribeCol`, `SQLForeignKeys`, `SQLGetFunctions`, `SQLPrimaryKeys`, `SQLProcedures`, `SQLProcedureColumns`, `SQLSpecialColumns`, `SQLStatistics`, `SQLTablePrivileges`, and `SQLTables`.

17. `SQLFetch` fetches a single row. Use `SQLFetchScroll` to return a rowset; the simplest type of usage is a basic array fetch.

18. `SQLGetInfo` provides much more than the server version. One call is needed per type of information wanted.

Other notes:

Most OCI calls include an error handle, which does not exist in CLI. When porting OCI calls with an error handle, the error handle parameter must be removed from the call, and a check of the return code from the CLI call must be added. When a return code other than SQL_SUCCESS occurs, a call to `SQLGetDiagRec` should be made.

The following classes of OCI functions have no equivalents in DB2 CLI. The functionality must be implemented either in SQL or in C (or C++) directly.

Navigational functions: `OCIObject__` and `OCICache__`

Datatype mapping and manipulation functions: `OCIColl__`, `OCIDate__`, `OCINumber__`, `OCIString__`, etc. However, CLI performs conversion of data between data types wherever possible.

External procedure functions: `OCIExtProc__`