# Table of Contents

**ACKNOWLEDGMENTS** .......................................................................................................................... 1  
**INTRODUCTION** ................................................................................................................................. 2  
**WHO SHOULD READ THIS DOCUMENT?** ......................................................................................... 3  
**WHY MIGRATE TO DB2 UDB?** ........................................................................................................... 4  
  - INTEGRATED SUPPORT FOR WINDOWS NT .................................................................................... 4  
  - INTEGRATED SYSTEM MANAGEMENT TOOLS ............................................................................... 4  
  - DATA REPLICATION ........................................................................................................................... 6  
  - INTEGRATED WEB ACCESS ................................................................................................................. 6  
  - INTEGRATED SUPPORT FOR COMPLEX DATA ............................................................................... 6  
  - INTEGRATED SUPPORT FOR DEVELOPMENT ENVIRONMENTS .................................................... 7  
  - IBM SOLUTION DEVELOPER PROGRAM ......................................................................................... 7  
**DB2 UNIVERSAL DATABASE PRODUCT FAMILY** .................................................................................. 8  
  - DB2 UNIVERSAL DATABASE PERSONAL EDITION ......................................................................... 9  
  - DB2 UNIVERSAL DATABASE WORKGROUP EDITION ....................................................................... 9  
  - DB2 UNIVERSAL DATABASE ENTERPRISE EDITION ....................................................................... 9  
  - DB2 UNIVERSAL DATABASE EXTENDED ENTERPRISE EDITION ...................................................... 9  
  - DB2 SOFTWARE DEVELOPER’S KIT (DB2 SDK) ............................................................................. 10  
**ARCHITECTURE** .................................................................................................................................... 11  
  - CLIENT-SERVER VIEW ....................................................................................................................... 11  
  - SERVER ARCHITECTURE .................................................................................................................... 11  
  - META DATA ....................................................................................................................................... 13  
  - PHYSICAL DATABASE ....................................................................................................................... 13  
  - MEMORY MODEL .............................................................................................................................. 16  
  - DATABASE RECOVERY LOGS ............................................................................................................. 17  
  - MAPPING OF ORACLE AND DB2 UDB TERMINOLOGY .................................................................... 18  
**DATA TYPES** ......................................................................................................................................... 19  
  - MAPPING ORACLE DATA TYPES TO DB2 UDB DATA TYPES .......................................................... 23  
  - NULL INDICATOR ............................................................................................................................... 24  
  - DATE ................................................................................................................................................ 24  
  - VARCHAR2 ....................................................................................................................................... 24  
  - NUMBER ........................................................................................................................................... 25  
  - DECIMAL ......................................................................................................................................... 25  
  - RAW ................................................................................................................................................ 26  
  - DB2 MAXIMUMS ............................................................................................................................... 26  
  - LENGTH OF DB2 IDENTIFIERS ............................................................................................................ 26  
  - DB2 LIMITS ...................................................................................................................................... 27  
**SQL LANGUAGE ELEMENTS** .............................................................................................................. 28  
  - NAMES SAVEPOINT AND COMPOUND SQL .................................................................................... 28  
  - SEQUENCE OBJECT ......................................................................................................................... 29  
  - OPTIMIZER HINTS .............................................................................................................................. 29  
  - ROWNUM ....................................................................................................................................... 30  

© IBM Corporation
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPLICATION DEVELOPMENT</td>
<td>42</td>
</tr>
<tr>
<td>C / C++</td>
<td>42</td>
</tr>
<tr>
<td>JAVA</td>
<td>44</td>
</tr>
<tr>
<td>STATIC SQL</td>
<td>44</td>
</tr>
<tr>
<td>EMBEDDED DYNAMIC SQL</td>
<td>45</td>
</tr>
<tr>
<td>CLI DYNAMIC SQL</td>
<td>45</td>
</tr>
<tr>
<td>OUTER JOIN</td>
<td>46</td>
</tr>
<tr>
<td>CURSORS</td>
<td>46</td>
</tr>
<tr>
<td>SQLDA MAPPINGS</td>
<td>48</td>
</tr>
<tr>
<td>Oracle SQLDA Declaration and Mapping to DB2 SQLDA</td>
<td>48</td>
</tr>
<tr>
<td>DB2 SQLDA Declaration</td>
<td>49</td>
</tr>
<tr>
<td>SQLCA / ORACA</td>
<td>49</td>
</tr>
<tr>
<td>ERROR HANDLING</td>
<td>50</td>
</tr>
<tr>
<td>CONSTRAINTS</td>
<td>53</td>
</tr>
<tr>
<td>DEFERRED UNIQUE CONSTRAINTS</td>
<td>53</td>
</tr>
<tr>
<td>REFERENTIAL INTEGRITY</td>
<td>53</td>
</tr>
<tr>
<td>CONCURRENCY CONTROL</td>
<td>54</td>
</tr>
<tr>
<td>CONCURRENCY AND LOCKS</td>
<td>54</td>
</tr>
<tr>
<td>ISOLATION LEVELS</td>
<td>55</td>
</tr>
<tr>
<td>OPTIMISTIC LOCKING</td>
<td>56</td>
</tr>
<tr>
<td>LOCK ESCALATION</td>
<td>57</td>
</tr>
<tr>
<td>DEADLOCKS</td>
<td>58</td>
</tr>
<tr>
<td>HOW TO IMPROVE CONCURRENCY</td>
<td>59</td>
</tr>
<tr>
<td>LOGGING</td>
<td>59</td>
</tr>
<tr>
<td>ARCHIVING</td>
<td>60</td>
</tr>
<tr>
<td>DATABASE ADMINISTRATION</td>
<td>61</td>
</tr>
<tr>
<td>COMMAND CENTER</td>
<td>61</td>
</tr>
<tr>
<td>CONTROL CENTER</td>
<td>62</td>
</tr>
<tr>
<td>SCRIPT CENTER</td>
<td>63</td>
</tr>
<tr>
<td>JOURNAL</td>
<td>63</td>
</tr>
<tr>
<td>COMMAND LINE PROCESSOR</td>
<td>63</td>
</tr>
<tr>
<td>AUTHENTICATION</td>
<td>64</td>
</tr>
<tr>
<td>ACCESS PRIVILEGES</td>
<td>64</td>
</tr>
<tr>
<td>DATA MANAGEMENT</td>
<td>64</td>
</tr>
</tbody>
</table>
Acknowledgments

This document was originally authored by Ming Wu, DB2 Technical Enablement Services, Toronto Lab.

The author would like to thank the following people for the invaluable advice and guidance provided in the production of this document:

- Dennis Bockus, Editor. IBM Canada
- Robert Newman, Contributor of many sample programs.
- Grant Hutchison, Reviewer.
- The members of the DM Services team in Toronto who provide daily input and guidance.
- The members of the Software Migration Project Office (SMPO) DB2 Migration Team including Debra Eaton who reviewed and contributed of many sample programs, as well as Kevin Decker and Marina Greenstein.
Introduction

DB2® Universal Database (DB2 UDB) can help improve the performance of database applications. Many solution developers have already chosen DB2 UDB as their primary development database environment, and have ported and continue to enable applications to it to take advantage of its unique features.

DB2 UDB is a true cross-platform DBMS, running on a wide variety of systems including Windows NT and 95, Solaris, HP-UX, AIX®, SCO UnixWare and OS/2®. It scales from single-processor workstations or servers, to symmetrical multiprocessor (SMP) servers, and on up to massively parallel processing (MPP) computers.

A real database leader in several technologies, it provides integrated support for complex data such as text documents; images; video and audio clips; integrated Web access through native support for Java, JDBC, SQLJ and Net.Data; integrated system management tools; and data replication service.

The current trend is to develop database applications that run on multiple database servers. The objective of this document is to help IBM business partners and customers to port Oracle applications to IBM’s DB2 Universal Database (DB2 UDB).

There are many motivations to enable an application to run on DB2 UDB. First, our partners and customers prefer to develop applications that are as database independent as possible. They may also want to deploy DB2 UDB technologies to achieve superior performance and scalability. And, they want to use DB2 UDB advanced features to simplify the application development. Last but not least, they want to run on DB2 UDB, the most reliable database server in the workstation market. The purpose of this paper is to assist them to achieve all of these goals.

This is a working document; new topics will be added as they appear in more and more porting situations. Meanwhile, depending on the type of application, not all topics discussed in this paper are relevant to a particular situation.
Who Should Read This Document?

This document is written for the application developers and database administrators who want to convert their applications from Oracle to DB2 UDB. We assume you are currently working with Oracle Release 7 or Oracle 8 and are porting to DB2 UDB Enterprise Edition (EE) Version 6.1. We do not cover the details of the DB2 UDB Extended Enterprise Edition (EEE) because it can run on hardware not supported by Oracle 7. However, applications developed for the EE edition are completely portable to the EEE edition without modifications. This document covers topics that are most often encountered by SQL developers porting from Oracle to DB2 UDB V6. It also covers topics that are relevant to database administrators.

We are referring to DB2 UDB V6 EE when we use the term DB2 UDB in this document. Unless otherwise specified, Oracle implies Oracle 7.

We assume that readers are familiar with the concepts of RDMS and with Oracle SQL and PL/SQL. We also assume readers have easy access to DB2 UDB V6.1 documentation. Refer to that documentation for detailed information about the actual SQL statement syntax.

This document takes a top-down approach, starting with the architectures of Oracle and DB2 UDB and ending with bits and bytes data-type conversions. Although both Oracle and DB2 UDB are essentially platform independent, we will point out the differences between Windows operating systems and UNIX operating systems when those differences matter.
Why Migrate to DB2 UDB?

DB2 UDB is a database leader in several technologies, and offers true multi-platform support and scalability. The same database is able to mix workloads on a single server. The DB2 UDB design handles workloads from high-volume online transaction processing (OLTP) to complex multi-user queries while maintaining excellent performance.

On December 1, 1998, the IBM Personal Systems Group and the IBM Software Group published a 100GB TPC-D benchmark (Transaction Processing Performance Council Benchmark D) using DB2 UDB 5.2.0 under Windows NT on an IBM Netfinity 7000 M10 server with four Pentium II Xeon 400 MHz processors. This benchmark achieved a multi-user throughput result of 831.2 QphD@100GB, a price/performance result of $130/QphD@100GB, and a power result of 3450 QppD@100GB. It has also since been published by TPC.

In addition to affordability and performance, DB2 UDB offers the following advantages:

- Integrated support for Windows NT
- Integrated system management tools
- Data replication service
- Integrated Web access
- Integrated support for complex data
- Integrated support for development environments
- IBM solution developer program

Each of these is described in detail below.

Integrated support for Windows NT

DB2 UDB conforms to Windows NT standards. It maps closely onto Windows NT internals for performance, and scales across all Windows NT hardware. It uses native Windows NT threads, and its architecture relies on Windows NT for task dispatching and other internal operating system functions. All these considerations make it more reliable and more tightly integrated to the operating system.

Integrated System Management Tools

DB2 administration tools allow users to perform database administration tasks for DB2 UDB servers that are available locally or remotely. The Control Center (Figure 1) is a graphical interface that can be used to perform server administrative tasks such as configuring, backing up and recovering data, managing directories, scheduling jobs and managing media, as well as accessing and manipulating databases. This tool can be installed on OS/2, Windows NT, or Windows 95/98 workstations.
The Control Center provides the following additional facilities to manage DB2 UDB servers:

- **Command Center**, to enter DB2 commands and SQL statements in an interactive window and see the execution output in a result window.
- **Script Center**, to create scripts, which can be stored and invoked at a later time. These scripts can contain DB2 commands, SQL statements and operating system commands. Scripts can be scheduled to run unattended, in which case they are called *jobs*. Jobs can be scheduled to run once only, at a later date or at regular intervals (for tasks such as backups).
- **Journal**, to view all available information in the history and alter a message in the log files about jobs that are pending, being executed or completed. It can also be used to review the results of unattended jobs.
- **Alert Center**, to monitor the system for early warnings of potential problems or to automate actions to correct problems that are discovered.
- **DB2 Performance Monitor**, to monitor the performance of a DB2 system and to create snapshots of data over a period of time or data for a particular event, which can be used to monitor activities.
- **Visual Explain**, to graphically analyze and tune SQL statements, as well as analyze query access plans.
- **SmartGuides**, to help perform administration tasks. For example, a SmartGuide is available to help tune the performance of the database server.
- **The Web Control Center**, the Java version of the DB2 UDB Control Center. The Web Control Center is implemented as a Java applet that uses DB2 JDBC support. Currently, the Web Control Center requires Netscape Navigator 4.04 for Windows 95 or Windows NT, and the JDK 1.1.4 patch for Navigator 4.04.

![Figure 1 Control Center Main Window](http://example.com/image.png)
Data Replication

DB2 UDB includes a complete data replication solution by supporting sources and targets that include the DB2 family, IMS, VSAM, Oracle, Sybase, Microsoft, Lotus Notes, and others to ensure timely, reliable, and consistent data across an enterprise. IBM offers the following data replication tools:

- IBM Replication – the DataPropagator® Relational Version 1 (DPROPR V1) products have been updated for Version 5 (V5) of the DB2 database.
- DB2 Universal Database V5 replication tools – the Control Center replication administration features and the Capture and Apply programs.
- IBM Capture and Apply for MVS V5.1
- Capture for VSE and VM V5.1 – integrated with IBM DB2 Server for VSE and VM V5.1.
- DPROPR V1 for support of DataJoiner
- DataPropagator Relational Capture and Apply for OS/400 V3.1
- IBM DataPropagator NonRelational
- IBM DataJoiner®
- Lotus NotesPump

Integrated Web Access

DB2 UDB provides web access to enterprise data on DB2 databases through native support for Java, Java Database Connectivity (JDBC), Embedded SQL for Java (SQLJ) and Net.Data®.

JDBC can be used to create applications or applets that access data in DB2 databases. These applets can be run inside HTML web pages on any system with a Java-enabled browser, independent of the client’s platform. The processing of JDBC applets is shared between the client and the server.

DB2 SQLJ support facilitates the creation, building and running of SQLJ programs against DB2 UDB databases.

DB2 Net.Data enables application developers to create Internet applications that access data from DB2 databases, are stored on a web server and are viewable from any web browser. While viewing these documents, users can either select automated queries or define new ones that retrieve the specified information directly from a DB2 UDB database.

Integrated Support for Complex Data

DB2 Universal Database Extenders allow storage and manipulation in the database of nontraditional data such as images, video, voice, complex documents, spatial objects and more. All these data types can be brought together in one SQL query and can then be manipulated with powerful built-in functions.
Each extender defines a new type of data in DB2 UDB by using built-in support for user-defined types and user-defined functions. Each extender also exploits DB2 Version 5 support for large objects of up to 2 gigabytes, and uses DB2 triggers to ensure referential integrity of image data.

The DB2 Extenders exploit the DB2 client/server model. Supported platforms are AIX, OS/2, Windows NT, HP-UX and Solaris Operating Environment.

**Integrated Support for Development Environments**

DB2 provides a Software Developer's Kit (SDK) that contains a collection of tools specially designed for database application developers. The DB2 SDK includes libraries, header files, documented Application Programming Interfaces (APIs) and sample programs to build database applications.

**IBM Solution Developer Program**

The IBM Solution Developer Program provides business, technical and marketing services to partners to help them develop and market applications. The strategic focuses of this program are network computing and e-business.

Benefits offered by this program include the following:

- Hardware and software discounts, equipment lease and loaner programs and business discounts to reduce development costs
- On-site and remote access to fully equipped testing and porting facilities at full-service IBM Solution Partnership Center (SPC) locations around the world
- Exclusive, focused development support
- Examples of how to interface with and exploit the newest technologies
- Technical information based on actual development experience in the form of “Frequently Asked Questions” and “Hints And Tips”
- The IBM Developer Connection, which is loaded with development tools, software and late-breaking news from IBM
- The *Global Software Solutions Guide*, an online catalog providing worldwide exposure to new customers for partners’ solutions
- In-depth technical seminars and hands-on workshops
- Partners In Development specialty areas, with specialized technical, business, marketing and information services for areas of expertise
- A technical library, complete with the latest white papers, road maps and a calendar of upcoming events
- The IBM Solution Developer Program worldwide web site, [http://www.developer.ibm.com/](http://www.developer.ibm.com/), which is a dynamic, 24-hour, 7-day-a-week service that provides information about all services.
DB2 Universal Database Product Family

The DB2 product family scales through a variety of platforms: AS/400® systems, RISC System/6000® hardware, IBM S390 systems, Intel systems and non-IBM machines from Hewlett-Packard and Sun Microsystems. Figure 2 provides a pictorial representation of the DB2 UDB connectivity.

DB2 UDB version 5.2 database software servers run on the following software environments: AIX, HP-UX, OS/2, SCO UnixWare, SVR4, Linux, Sun Solaris, Windows NT, Windows 98 and Windows 95.

Client access is provided for all these platforms, as well as for DOS, Apple MacOS, and Silicon Graphics IRIX. In addition, web access is provided with popular browsers and Java applications using DB2’s native Java/JDBC support and Net.Data.

Figure 2 DB2 UDB Connectivity
The DB2 UDB products and components include:

- DB2 Personal Edition
- DB2 Workgroup Edition
- DB2 Enterprise Edition
- DB2 Extended Enterprise Edition
- DB2 Software Developer’s Kit

**DB2 Universal Database Personal Edition**

DB2 Universal Database Personal Edition allows for the creation and use of local databases. It also allows access to remote relational databases when they are available. This product is available for the OS/2, Windows NT, and Windows 95 operating systems.

**DB2 Universal Database Workgroup Edition**

The DB2 Universal Database Workgroup Edition server enables local clients, remote clients and applications to create, update, control and manage relational databases using Structured Query Language (SQL), ODBC, or CLI. It contains all the latest DB2® Client Application Enablers, which enable client workstations to access the DB2 UDB server and all supported DB2 Net.Data products.

**DB2 Universal Database Enterprise Edition**

The DB2 Enterprise Edition includes all functions provided in the DB2 Workgroup Edition, plus DB2® Connect Enterprise Edition to allow support for host connectivity. This provides multi-user access to DB2 databases residing on host systems such as MVS/ESA, OS/390, AS/400, VM, and VSE. The DB2 Enterprise Edition supports unlimited LAN database access.

**DB2 Universal Database Extended Enterprise Edition**

DB2 Universal Database Extended Enterprise Edition (formerly known as DB2 Parallel Edition) enables a database to be partitioned across multiple independent computers of a common platform. SQL operations and utilities can operate in parallel on the individual database partitions. Performance is enhanced by speeding up the execution time of a single query or utility.
DB2 Software Developer's Kit (DB2 SDK)

DB2 SDK is a collection of tools that enable database application developers to build character-based, multimedia or object-oriented applications. It includes libraries, header files, documented APIs and sample programs.

The DB2 SDK can be used to develop applications that use the following interfaces:

- Embedded SQL, both static and dynamic
- Call Level Interface (CLI) development environment (compatible with ODBC from Microsoft)
- Java Database Connectivity (JDBC)
- Application programming interfaces to access database utilities

DB2 SDK supports several programming languages (including COBOL, FORTRAN, Java, C, and C++) for application development, and provides precompilers for the supported languages. It is available on all DB2 UDB-supported platforms. The same application does not require any source changes to run against DB2 UDB on any Intel or UNIX platform; therefore, only one base code allows support for several platforms.

DB2 SDK also supports SQLJ. Along with DB2 JDBC support provided by the DB2 Client Application Enabler (DB2 CAE), DB2 SQLJ support allows for the creation, build, and run of embedded SQL for Java applications, applets, stored procedures and user-defined functions (UDFs). These contain static SQL and use embedded SQL statements that are bound to a DB2 database.

Other important DB2 products are:

- DB2® Connect Personal Edition, which provides access from a single workstation to DB2 databases residing on host systems such as MVS/ESA, OS/390, OS/400, VM and VSE, as well as access to DB2 Universal Databases. This product is available for the OS/2, Windows 3.1x, Windows NT, and Windows 95 operating systems. DB2 Connect Enterprise Edition provides similar capabilities in a multi-user environment including UNIX systems, OS/2, and Windows NT.
- DB2® OLAP (online analytical processing) Server, which is designed for multidimensional planning, analysis, and reporting applications.
- DB2 for Domino, which extends the capabilities of DB2 Universal Database to Lotus Notes and Domino users.
- DB2 DataJoiner 2.1.2, which provides a single interface to heterogeneous databases. It provides global query optimization, and supports most relational and non-relational database systems.
Architecture

In this section, we try to 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 in different address spaces. The application code runs in the client process while the server code runs on separate processes. The client processes can run on the same or different machine from the database server, accessing the database server via 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 does not have such an extension. Instead, users can use any programming languages supported by the pre-compiler such as C, C++, Java, Cobol, REXX and Fortran to construct the 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, CLP does not have all of the features of 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 dedicated process architecture. For Window 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 db2agent) serving that connection. There may be more associated agent processes (db2agentp) 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 Extended Enterprise Edition (EEE) extends the architecture to environments with multiple nodes. For each node in a Nodegroup, a db2agent (or more than one agent for an environment with SMP node) is created for the connection. While the application communicates only with the coordinating node, DB2 UDB EEE will harness the resources on all of the nodes, totally transparent to the application. However, this document does not discuss DB2 UDB EEE in detail.
Like DB2 UDB, Oracle implements a similar architecture called the 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 the multithreaded server architecture, which implements a pooled-agents architecture with a number of server processes serving the applications. The requests from the applications are placed on a requested queue and are dispatched to one of the 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; note that DB2 UDB does not have a similar pool process model.

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

Metadata provide the roadmap to interpreting the information stored in the database. In Oracle, the metadata are stored in the *Data Dictionary* and in DB2 UDB, the metadata are stored in the *System Catalog*. The table names for Oracle and DB2 UDB are very different. You need to be familiar with the table names in DB2 UDB to map the system information from Oracle to DB2 UDB.

**Physical Database**

DB2 UDB and Oracle share the concept of the physical storage model. In this section we try to map the terminology between the two databases.

Both databases store the 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*).
DB2 UDB containers can be **system managed** (SMS) or **database managed** (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. Typically, only one type of container is used in a table space.

SMS is simple to manage. It grows automatically and is managed by the operating system. However, there may be more overhead incurred by the operating system. DMS is 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 will show you the best method for mapping Oracle table spaces to DB2 UDB table spaces.

Oracle data are stored in *data blocks*. The size of the data block is defined when a database is created (2K or 4K). An *extent* is a specific number of contiguous data blocks, obtained in a single allocation. *Segments* are made up of extends 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 are stored in *pages*. The size of the *page* (4K, 8K, 16K or 32K) is defined for each table space. In DB2 UDB, an *extent* is a specific number of contiguous pages. 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 pre-allocated. 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. However, the users can change the SMS container definition so the space is allocated one *extent* at a time. The default extent is 32 pages, while up to 256 pages are allowed.

DB2 UDB SMS data storage is completely page-based storage. In other words, DB2 SMS does not require a contiguous set of data blocks to store data as in Oracle. Hence, there is no fragmentation problem in DB2 UDB SMS tablespaces.
Oracle Database

System Table Space

User Table Space

Default Oracle Database Layout
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 used to share information between an application and the Oracle Database Server is called the User Global Area or the UGA. The UGA contains information such as user context and row-cache-cursors buffers.

DB2 UDB stores similar information to Oracle’s database. DB2 UDB uses different shared memory areas to store different types of information.

- **Database Manager Shared Memory Set** stores all the relevant information regarding a particular instance. Examples include lists of all active connections and security information.

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

- **Application Shared Memory Set** stores information that is relevant to a particular application. An example is the data passed from the application process to the dedicated Server process. This maps directly to the Oracle UGA. The sample contents are sort heaps, cursor information and session contexts.

*Database Manager Shared Memory Set* and *Database Shared Memory Set* can be loosely mapped to Oracle’s *SGA*. *Application Shared Memory Set* can be loosely mapped to the *UGA*.

*Data buffer cache,* or *bufferpools* (in DB2 UDB terminology), are used to buffer data in memory to reduce the amount of I/O operations to the physical database. Database performance can dramatically improved if the requested data are 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 *bufferpools* and you can assign multiple table spaces to a particular bufferpool. If the database has only one *bufferpool* for all the table spaces, then
it is equivalent to the Oracle implementation. A default bufferpool (IBMDEFAULTBP) is created when a database is first created. New bufferpools are created using the DDL statement CREATE BUFFERPOOL. Use the ALTER BUFFERPOOL statement to change the size of the bufferpool.

DB2 UDB can yield significant performance improvement by providing the ability to allocate different amounts of memory to cache data in each table space. Oracle cached tables are similar to bufferpools that are as large as the tables.

Since the hit-ratio (percentage of database access that is required to perform a physical I/O) can impact the performance of the database, tuning the bufferpools for each table space can significantly improve database performance. An example is to put indices of a particular table in a separate table space and assign a dedicated bufferpool to it. Assuming most of the table access is via index access, this can significantly improve database performance.

Database Recovery logs

All RDMS require the changes to be logged to the database 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 while a table is being updated or mutated.

DB2 UDB implements the write-ahead logging where the changed data is always written in the log files before the change is committed. DB2 UDB logs all of its changes in its log files, including the old version of the data. It does not have rollback segments. Also, DB2 UDB logs read-only as well as update transactions in the log files. Due to the differences in the database recovery implementations, the two databases have noticeable differences in controlling concurrency access of the data. These differences are discussed in more detail in the Concurrency Control section.
### Mapping of Oracle and DB2 UDB terminology

This section gives readers who are familiar with Oracle an overview of DB2 UDB terminology. The following tables provides a quick (and simple) mapping of DB2 UDB and Oracle jargon.

<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>DDCS access to hosts</td>
</tr>
<tr>
<td>N/A</td>
<td>PL/SQL</td>
<td>Proprietary programming language extension to SQL. DB2 UDB stored procedures can be programmed in Java, C, C++, Cobol, Fortran, and REXX</td>
</tr>
<tr>
<td>DB2 CLP</td>
<td>SQL Plus</td>
<td>Command line interface to the server</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 container/data files</td>
</tr>
<tr>
<td>Extends</td>
<td>Extends</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>
<tr>
<td>System catalog</td>
<td>Data dictionary</td>
<td>Meta data of the database</td>
</tr>
<tr>
<td>SMS</td>
<td>N/A</td>
<td>System managed tablespace</td>
</tr>
<tr>
<td>DMS</td>
<td>Data files</td>
<td>Database managed table space</td>
</tr>
<tr>
<td>Bufferpools</td>
<td>Data cache</td>
<td>Buffers data in the tablespaces to reduce disk I/O</td>
</tr>
<tr>
<td>Packages cache</td>
<td>Statement cache</td>
<td>Cache 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</td>
</tr>
<tr>
<td>see comments</td>
<td>SGA</td>
<td>Database Manager Shared Segments and Database shared segments</td>
</tr>
<tr>
<td>Application shared segments</td>
<td>UGA</td>
<td>Shared memory areas 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 pre-compiled 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, C/C++ data type mapping, and a quick description of each. Note that DB2 UDB has multiple definitions for DATE and multiple types for NUMBER.
<table>
<thead>
<tr>
<th>SQL Data Types / Decl/ sqldata</th>
<th>C/C++ Data Types / Decl</th>
<th>sqllen</th>
<th>Description</th>
</tr>
</thead>
</table>
| integer                       |                         |        | - 16-bit signed integer  
|                               | SMALLINT age;           |        | - range between (-32,768 and 32,767)  
|                               | (500 or 501)            |        | - precision of 5 digits  
|                               | short age = 32;         | 2      |             |
|                               | short int year;         |        |             |
|                               | INT salary;             |        | - 32-bit signed integer  
|                               | (496 or 497)            |        | - range between (-2,147,483,648 and 2,147,483,647)  
|                               | long salary;            | 4      | - precision of 10 digits  
|                               | long int deptno;        |        |             |
|                               | BIGINT serial_num;      |        | - 64-bit signed integer |
|                               | (492 or 493)            | 8      |             |
|                               | long long serial;       |        |             |
|                               | _int64 serial;          |        |             |
|                               | sqllint64 serial        |        |             |
| floating point                |                         |        | - single precision floating point  
|                               | REAL bonus;             |        | - 32-bit approximation of a real number  
|                               | FLOAT(n);               | 4      | - FLOAT(n) can be synonym for REAL if 0 < n < 25  
|                               | (480 or 481)            |        |             |
|                               | float bonus;            |        |             |
|                               | DOUBLE wage;            |        | - double precision floating point  
|                               | (480 or 481)            | 8      | - 64-bit approximation of a real number  
|                               | DOUBLE PRECISION wage;  |        | - range in (0, -1.79769E+308 to -2.225E-307, 2.225E-307 to 1.79769E+308)  
|                               |                           |        | - FLOAT(n) can be synonym for DOUBLE if 24 < n < 54  
|                               |                           |        |             |
| Decimal                       |                         |        | - packed decimal |
|                               | DECIMAL(4,2) price;     |        | - no exact equivalent for sql decimal type - use C  
|                               | DEC(4,2) price;         |        | double data type  
|                               | (484 or 485)            |        | - if precision/scale not specified, default is (5,0)  
|                               | NUMERIC (4,2) price;    |        | - max precision is 31 digits, and max range  
|                               | NUM (4,2) price;        |        | between (-10**31 + 1 ... 10**31 -1)  
|                               |                           |        | - consider using char / decimal func to manipul  
|                               |                           |        | packed decimal fields as char data  
| Date/Time                     |                         |        | - null-terminated character form (11 characters)  
|                               | DATE dt;                | 10     | - or- varchar struct form (10 characters)  
|                               | (384 or 385)            |        | - eg. ‘11/02/1997’  
|                               | char dt[11];            |        | - stored internally as a packed string of 4 bytes  
|                               | struct (                |        |             |
|                               |     short len;          |        |             |
|                               |     char data[10];      |        |             |
|                               | ) dt;                   |        |             |
|                               | TIME tm;                | 8      | - null-terminated character form (9 characters)  
|                               | (388 or 389)            |        | - or- varchar struct form (8 characters)  
|                               | char tm[9];             |        | - eg. ‘19:21:39’  
|                               |                        |        | - stored internally as a packed string of 3 bytes  
|                               | TIMESTAMP ts;           | 26     | - null-terminated character form (27 characters)  
|                               | (392 or 393)            |        | - or- varchar struct form (26 characters)  
|                               | char ts[27];            |        | - e.g. ‘1997-10-24-01.43.00.000000’  
|                               |                        |        | - stored internally as a packed string of 10 bytes  
<p>|</p>
<table>
<thead>
<tr>
<th>SQL Data Types / Decl</th>
<th>C/C++ Data Types / Decl</th>
<th>sqlen</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Character</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHAR</td>
<td>char;</td>
<td>n</td>
<td>- fixed length character string consisting of n bytes</td>
</tr>
<tr>
<td></td>
<td>char[5] zip;</td>
<td></td>
<td>- use char[n+1] where 1 &lt;= n &lt;= 254</td>
</tr>
<tr>
<td></td>
<td>(452 or 453)</td>
<td></td>
<td>- if length not specified, defaults to 1</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>char address[41];</td>
<td>n</td>
<td>- null-terminated variable length character string</td>
</tr>
<tr>
<td></td>
<td>(460 or 461)</td>
<td></td>
<td>- use char[n+1] where 1 &lt;= n &lt;= 32672</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>struct { short len; char data[40]; } address;</td>
<td>len</td>
<td>- non null-terminated varying character string with 2-byte string length indicator</td>
</tr>
<tr>
<td></td>
<td>(448 or 449)</td>
<td></td>
<td>- use char[n] in struct form where 1 &lt;= n &lt;= 32672</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- default sql type</td>
</tr>
<tr>
<td>LONG VARCHAR</td>
<td>struct { short len; char data[n]; } voice;</td>
<td>len</td>
<td>- non null-terminated varying character string with 2-byte string length indicator</td>
</tr>
<tr>
<td></td>
<td>(456 or 457)</td>
<td></td>
<td>- use char[n] in struct form where 32673 &lt;= n &lt;= 32700</td>
</tr>
<tr>
<td>CLOB(n)</td>
<td>sql type is clob(1m) chapter;</td>
<td>n</td>
<td>- non null-terminated varying character string with 4-byte string length indicator</td>
</tr>
<tr>
<td></td>
<td>(408 or 409)</td>
<td></td>
<td>- use char[n] in struct form where 1 &lt;= n &lt;= 2 147 483 647</td>
</tr>
<tr>
<td>CLOB locator variable</td>
<td>sql type is clob_locator cref;</td>
<td></td>
<td>- Identifies CLOB entities residing on the server</td>
</tr>
<tr>
<td></td>
<td>(964 or 965)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLOB file reference variable</td>
<td>sql type is clob_file cFile;</td>
<td></td>
<td>Descriptor for file containing CLOB data</td>
</tr>
<tr>
<td></td>
<td>(808 or 809)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Binary</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BLOB(n)</td>
<td>sql type is blob(1m) video;</td>
<td>n</td>
<td>- non null-terminated varying binary string with 4-byte string length indicator</td>
</tr>
<tr>
<td></td>
<td>(404 or 405)</td>
<td></td>
<td>- use char[n] in struct form where 1 &lt;= n &lt;= 2 147 483 647</td>
</tr>
<tr>
<td>BLOB locator variable</td>
<td>sql type is blob_locator bref;</td>
<td></td>
<td>- Identifies BLOB entities on the server</td>
</tr>
<tr>
<td></td>
<td>(960 or 961)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BLOB file reference variable</td>
<td>sql type is blob_file bFile;</td>
<td></td>
<td>Descriptor for file containing BLOB data</td>
</tr>
<tr>
<td></td>
<td>(804 or 805)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### SQL Data Types / Decl
<table>
<thead>
<tr>
<th>sqltype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double-Byte</td>
<td></td>
</tr>
<tr>
<td>GRAPHIC(1)</td>
<td>sqldbchar dbyte;</td>
</tr>
<tr>
<td>GRAPHIC(n)</td>
<td>sqldbchar graphic1[n+1]; wchar_t graphic2[100];</td>
</tr>
<tr>
<td></td>
<td>- sqldbchar is a single double-byte character string</td>
</tr>
<tr>
<td></td>
<td>- For a fixed-length graphic string of length integer which may range from 1 to 127. If the length specification is omitted, a length of 1 is assumed.</td>
</tr>
<tr>
<td></td>
<td>- pre-compiled with WCHARTYPE NOCONVERT option</td>
</tr>
<tr>
<td>VARGRAPHIC(n)</td>
<td>struct tag {</td>
</tr>
<tr>
<td></td>
<td>short int;</td>
</tr>
<tr>
<td></td>
<td>sqldbchar[n];</td>
</tr>
<tr>
<td></td>
<td>vargraphic1;</td>
</tr>
<tr>
<td></td>
<td>sqldbchar[n+1];</td>
</tr>
<tr>
<td></td>
<td>n*2+4 For a varying-length graphic string of maximum length integer, which may range from 1 to 16 336.</td>
</tr>
<tr>
<td></td>
<td>- pre-compiled with WCHARTYPE NOCONVERT option.</td>
</tr>
<tr>
<td></td>
<td>- null terminated variable-length</td>
</tr>
<tr>
<td>LONGVARGRAPHIC(n)</td>
<td>struct tag {</td>
</tr>
<tr>
<td></td>
<td>short int;</td>
</tr>
<tr>
<td></td>
<td>sqldbchar[n];</td>
</tr>
<tr>
<td></td>
<td>long_vargraphic1;</td>
</tr>
<tr>
<td></td>
<td>n*2 For a varying-length graphic string with a maximum length of 16 350 and a 2-byte string length indicator 16337&lt;=n &lt;=16350.</td>
</tr>
<tr>
<td></td>
<td>- pre-compiled with WCHARTYPE NOCONVERT option.</td>
</tr>
<tr>
<td>DBCLOB(n)</td>
<td>sql type is dbclob(1m) tokyo_phone_dir;</td>
</tr>
<tr>
<td></td>
<td>- For Non null-terminated varying double-byte character large object string of the specified maximum length in double-byte characters.</td>
</tr>
<tr>
<td></td>
<td>- 4 bytes string length indicator</td>
</tr>
<tr>
<td></td>
<td>- use dbclob(n) where 1&lt;=n &lt;= 1 073 741 823 double-byte characters.</td>
</tr>
<tr>
<td></td>
<td>- pre-compiled with WCHARTYPE NOCONVERT option.</td>
</tr>
<tr>
<td>DBCLOB locator variable</td>
<td>struct tag {</td>
</tr>
<tr>
<td></td>
<td>short int;</td>
</tr>
<tr>
<td></td>
<td>sqldbchar[n];</td>
</tr>
<tr>
<td></td>
<td>long_vargraphic1;</td>
</tr>
<tr>
<td></td>
<td>sql type is dbclob_locator tokyo_phone_loc; Identifies DBCLOB entities residing on the server.</td>
</tr>
<tr>
<td></td>
<td>- pre-compiled with WCHARTYPE NOCONVERT option.</td>
</tr>
<tr>
<td>DBCLOB file reference variable</td>
<td>struct tag {</td>
</tr>
<tr>
<td></td>
<td>short int;</td>
</tr>
<tr>
<td></td>
<td>sqldbchar[n];</td>
</tr>
<tr>
<td></td>
<td>long_vargraphic1;</td>
</tr>
<tr>
<td></td>
<td>sql type is dbclob_file tokyo_phone_ref; Descriptor for file containing DBCLOB data.</td>
</tr>
<tr>
<td></td>
<td>- pre-compiled with WCHARTYPE NOCONVERT option.</td>
</tr>
<tr>
<td>External Data</td>
<td>Datalink(n);</td>
</tr>
<tr>
<td></td>
<td>n+54 -The length of a DATALINK column is 200 bytes.</td>
</tr>
</tbody>
</table>

1 sqltype column denotes nullability and data type of a column within an SQL Descriptor.  1.sqltype column denotes nullability and data type of a column within an SQL Descriptor Area (SQLDA) - even numbers indicate data types declared as NOT NULL

2 sqllen column denotes the length attribute of the column

3 sqllen field for a decimal data type contains the precision in byte 1 and the scale in byte 2. Stored internally in packed decimal format (BCD notation) with an implicit decimal point whose position is determined by the precision and scale of the number (precision is the total # of bits or digits excluding the sign, scale is the # of digits to the right of the decimal point). Scale cannot be negative or greater than the precision.

4 FOR BIT DATA clause optional for CHAR, VARCHAR and LONG VARCHAR data types

5 can declare additional host variable types to denote locator and/or file reference variables for CLOB data types

6 can declare additional host variable types to denote locator and/or file reference variables for CLOB data types

7 double-byte character string data types include GRAPHIC, VARGRAPHIC, LONG VARGRAPHIC and DBCLOB

---

© IBM Corporation
## 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. Note that 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>
<tbody>
<tr>
<td>DATE</td>
<td></td>
<td>DATE</td>
<td>- if MM/DD/YYYY req'd, use DATE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TIME</td>
<td>- if HH:MM:SS req'd, use TIME</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TIMESTAMP</td>
<td>- if MM/DD/YYYY-HH:MM:SS.000000 req'd, use TIMESTAMP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Use Oracle TO_CHAR() fn to extract for subsequent DB2 load. Note Oracle default format is DD-MON-YY</td>
</tr>
<tr>
<td>VARCHAR2(n)</td>
<td>n &lt;=4000</td>
<td>VARCHAR(n)</td>
<td>n &lt;= 32762</td>
</tr>
<tr>
<td>LONG</td>
<td>n &lt;= 2GB</td>
<td>LONG VARCHAR(n)</td>
<td>- if n &lt;= 32700 bytes, use LONG VARCHAR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CLOB(n)</td>
<td>- if n &lt;= 2 GB, use CLOB</td>
</tr>
<tr>
<td>RAW(n)</td>
<td>n &lt;= 255</td>
<td>CHAR(n) FOR BIT DATA</td>
<td>- if n &lt;= 254, use CHAR(n) FOR BIT DATA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VARCHAR(n) FOR BIT DATA</td>
<td>- if n &lt;= 32672, use VARCHAR(n) FOR BIT DATA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BLOB(n)</td>
<td>- if n &lt;= 2 GB, use BLOB(n)</td>
</tr>
<tr>
<td>LONG RAW</td>
<td>n &lt;= 2 GB</td>
<td>LONG VARCHAR FOR BIT DATA(n)</td>
<td>- if n &lt;= 32700 bytes, use LONG VARCHAR FOR BIT DATA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BLOB(n)</td>
<td>- if n &lt;= 2 GB, use BLOB(n)</td>
</tr>
<tr>
<td>BLOB</td>
<td>n&lt;= 4 GB</td>
<td>BLOB(n)</td>
<td>- if n&lt;= 2 GB use BLOB(n)</td>
</tr>
<tr>
<td>CLOB</td>
<td>n&lt;= 4GB</td>
<td>CLOB(n)</td>
<td>- if n&lt;=2GB use CLOB(n)</td>
</tr>
<tr>
<td>NCLOB</td>
<td>n&lt;4GB</td>
<td>DBCLOB(n)</td>
<td>- if n&lt;2GB use NCLOB(n/2)</td>
</tr>
<tr>
<td>NUMBER</td>
<td></td>
<td>INTEGER</td>
<td>- if Oracle decl is NUMBER(p), use SMALLINT / INTEGER</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DECIMAL(p,s)</td>
<td>- if Oracle decl is NUMBER(p,s), use DECIMAL(p,s)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FLOAT(n) / REAL / DOUBLE</td>
<td>- if Oracle decl is NUMBER, use FLOAT(n) / REAL / DOUBLE</td>
</tr>
</tbody>
</table>
**Null Indicator**

Null indicators are defined as SMALLINT or short in C and C++.

**DATE**

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 UDB contains only the year, month and day. Data type `TIME` contains only the HH:MM:SS information. In UDB, the data type `TIMESTAMP` contains all the information from the year through to the seconds and fractions of seconds.

Because the default formats are also different (Oracle is DD-MON-YY and DB2 UDB is MM/DD/YYYY), you need to map from one to another using formatting functions. For example in order to map Oracle `DATE` to DB2 `DATE` for import, use the `TO_CHAR(ActualDate,”M/DD/YYYY”)`.

Oracle built-in functions such as `NEXT_MONTH`, `ADD_MONTH`, `NEXT_DAY` can be translated to DB2 UDB `user-defined functions` (UDF).

`TIMESTAMP` gives the most complete set of timing information. However, it also takes the most amount of storage space. Hence, only use it if you need the full resolution it provides. Code fragments in Appendix A illustrate the differences in the two databases.

**Varchar2**

Many Oracle applications use `VARCHAR2` for very small character strings, for example `VARCHAR2(2)`. 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.
The Oracle data type NUMBER can be mapped to many DB2 types. The type of mapping depends on what the NUMBER is really used for. Is the field used to store an integer or a real number with floating points? Also, the precision digits are required for the field. Another consideration is the space usage. The space usage for each DB2 type can vary depending on the type declared: SMALLINT uses 2 bytes and INTEGER uses 4 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 SMALLEST, for example, can save you 18 bytes per column.

**DECIMAL**

An Oracle NUMBER with non-zero precision should be mapped to DB2 data type DECIMAL. DECIMAL is stored packed in DB2 and here is an example of how the column can be inserted and retrieved from the database using the CLI interface. Make sure to retrieve the column as type SQL_C_CHAR.

```c
char limit_balance(14) = "9999999.99";
short limit_balance_ind = 10;
char balance(14);                /* for decimal (9,2) */
short balance_ind;               /* balance indicator */
.....
/*get a CLI statement handle and update the
decimal column using the parameter marker */
rc=SQLBindParameter(hStmtHandle,1,SQL_PARAM_INPUT,SQL_C_CHAR,SQL_DECIMAL,13,2,limit_balance,14,&(limit_balance_ind));
...
/* get a CLI statement handle for select and retrieve the decimal column using the parameter marker */
rc = SQLBindCol (hStmtHandle, 2,SQL_C_CHAR, balance,14, &(balance_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()`.

Oracle 8, 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 V6.1 support 4K, 8K, 16K and 32K page sizes. Depending on the page size, the maximum of columns and row length varies.

<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 tablespace</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>Most index key length</td>
<td>1,024</td>
<td>1,024</td>
<td>1,024</td>
<td>1,024</td>
</tr>
<tr>
<td>Most number of columns in an index key</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
</tbody>
</table>

**Length of DB2 Identifiers**

Some Identifier names in DB2 UDB have a maximum length of 18 bytes, including indexes, constraints, and triggers. However, identifier names in Oracle can be up to 30 characters in length. The conversion of the data types from Oracle to DB2 UDB will thus require that some Oracle identifiers exceeding 18 characters be compacted to fall within the 18-characters limit of DB2 UDB. Appendix B contains the script to extract Oracle identifiers greater than 18 characters that need to be modified when ported to DB2.
DB2 Limits

<table>
<thead>
<tr>
<th>Database Manager Limits</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Most SQL statement length in bytes</td>
<td>65,535</td>
</tr>
<tr>
<td>Identifier Length Limits (limit in Bytes)</td>
<td></td>
</tr>
<tr>
<td>Longest authorization name (can only be single-byte characters)</td>
<td>8</td>
</tr>
<tr>
<td>Longest constraint name</td>
<td>18</td>
</tr>
<tr>
<td>Longest correlation name</td>
<td>18</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>8</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 table name</td>
<td>128</td>
</tr>
<tr>
<td>Longest view name</td>
<td>128</td>
</tr>
<tr>
<td>Longest alias name</td>
<td>128</td>
</tr>
<tr>
<td>Longest unqualified package name</td>
<td>8</td>
</tr>
<tr>
<td>Longest unqualified user-defined type, user-defined function, stored procedure, buffer pool, table space, nodegroup, trigger or index name</td>
<td>18</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Numerical Limits</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Smallest INTEGER value</td>
<td>-2 147 483 648</td>
</tr>
<tr>
<td>Largest INTEGER value</td>
<td>+2 147 483 647</td>
</tr>
<tr>
<td>Smallest SMALLINT value</td>
<td>-32 768</td>
</tr>
<tr>
<td>Largest SMALLINT value</td>
<td>+32 767</td>
</tr>
<tr>
<td>Largest decimal precision</td>
<td>31</td>
</tr>
<tr>
<td>Smallest DOUBLE value</td>
<td>-1.79769E+308</td>
</tr>
<tr>
<td>Largest DOUBLE value</td>
<td>1.79769E+308</td>
</tr>
<tr>
<td>Smallest positive DOUBLE value</td>
<td>2.225E-307</td>
</tr>
<tr>
<td>Largest negative DOUBLE value</td>
<td>-2.225E-307</td>
</tr>
<tr>
<td>Smallest REAL value</td>
<td>-3.402E+38</td>
</tr>
<tr>
<td>Largest REAL value</td>
<td>3.402E+38</td>
</tr>
<tr>
<td>Smallest positive REAL value</td>
<td>1.175E-37</td>
</tr>
<tr>
<td>Largest negative REAL value</td>
<td>-1.175E-37</td>
</tr>
</tbody>
</table>
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 using migration tools such as ManTech’s SQL Conversion Workbench. Appendix A contains a C static SQL driver to insert, delete and query data from DB2 UDB tables in the SAMPLE database. However, the topics discussed here may require more planning and time to redesign the applications in order to achieve the same or similar results.

Names Savepoint and Compound SQL

Oracle savepoints are logical points in a transaction where the application can choose to roll back. An application can choose to roll back a partial transaction to a particular savepoint. Named savepoints are not currently supported by DB2 UDB but will be supported in a later release.

Instead, DB2 UDB includes the concept of a compound SQL. A compound SQL is a group of SQL statements that are wrapped between the BEGIN COMPOUND AUTOMATIC and END COMPOUND statements. DB2 UDB treats them as atomic and will roll back or commit all of them together. Some Oracle transactions with named savepoints can be mapped to DB2 UDB transactions with compound SQL statements.

If compound SQL is not sufficient, it may be better to group those statements that cause changes to the database from a specific savepoint and then re-execute if there is a need to do a rollback from the transaction.

One of the major advantages of compound SQL, and a feature not related to the topic of savepoint, 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 the transaction.

<table>
<thead>
<tr>
<th>DB2 Compound SQL</th>
<th>Oracle Savepoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXEC INSERT..</td>
<td>EXEC INSERT..</td>
</tr>
<tr>
<td>EXEC DELETE...</td>
<td>EXEC DELETE...</td>
</tr>
<tr>
<td>EXEC SQL BEGIN COMPOUND AUTOMATIC</td>
<td>SAVEPOINT A:</td>
</tr>
<tr>
<td>UPDATE ...</td>
<td>EXEC UPDATE..</td>
</tr>
<tr>
<td>INSERT...</td>
<td>IF SQL%NOTFOUND THEN</td>
</tr>
<tr>
<td>ELSE INSERT ..</td>
<td>ROLLBACK TO SAVEPOINT A;</td>
</tr>
<tr>
<td>END COMPOUND;</td>
<td>COMMIT;</td>
</tr>
<tr>
<td>COMMIT;</td>
<td></td>
</tr>
</tbody>
</table>
**Sequence Object**

The Oracle *sequence* object generates an ordered unique number. Typically sequence objects are used as the unique number for the primary key on insert. For example, a sequence object can be used to generate the next employee number or the next invoice order number. DB2 does not currently support *sequence* objects but will support the auto increment object in a later release.

Applications can use the *RANDOM* built-in function if the order does not matter, or create a table (for example, `next_number`) to generate the next number in the sequence by using triggers and `MAX(column) + 1`. Or use `GENERATE-UNIQUE UDF` to create a list of increasing numbers. Or, define a *UDF* to generate an ordered sequence. Appendix C includes the sample code for both trigger and UDF solutions.

**Optimizer Hints**

The Oracle optimizer used to be *ruled based*. The new *cost-based* Oracle optimizer is now available. Oracle applications sometimes use hints to force the optimizer to pick a particular access plan. DB2 UDB has only the *cost based* optimizer and does not support hints. Instead, DB2 UDB optimizer relies on detailed statistics to determine the proper access plan. Thus, it is important to keep the statistics up-to-date by running `runstat` frequently.

If a table grows and shrinks in size quickly, and it is difficult to update the statistics, it may be worthwhile to change to a lower optimization level where the number of rows does not affect the access plan. The default optimization level is 5. You can change the optimization level to 0 during the pre-compile phase.

```
    db2 prep filename.sqc OPTLEVEL 0
```

DB2 UDB V6 has the concept of *VOLATILE* tables. Volatile tables do not depend on the statistics stored in the System Catalog and reassess the access plan every time the table is accessed. For example, consider a table that is emptied at the beginning of the day and grows by thousands of rows each hour. If the statistics are updated at the start of the day, the optimizer may always think that the table is quite empty and will table scan when searching for a row. This can prove disastrous as the day goes on. Using Volatile table can prevent the performance degradation. However, there is an overhead for not caching the access plan when using the Volatile table.

Another solution is to update the statistics when the table is at its typical size (say, in the middle of the day).
**ROWNUM**

The ROWNUM cursor attribute is an Oracle built-in to identify a row number for a cursor. ROWNUM can be used as a counter or to limit the result set. DB2 UDB does not support ROWNUM. Instead, the application can use counters in its application to keep track of the row number. However, if ROWNUM is used to limit the result set, using the **FETCH FOR N ROWS** clause in the DECLARE CURSOR will return only N rows from the server. **FETCH FOR N ROWS** has the added benefit that N rows will be passed in one network crossing and buffered on the client side.

DB2 UDB has another clause, **OPTIMIZE FOR N ROWS in DECLARE CURSOR**. **OPTIMIZE FOR N ROWS** does not limit the result set. Instead, it returns N rows at a time to the client and buffers at the client machine. The application can request beyond N rows, but it requires more data to be passed from the server to the client.

**ROWID**

The Oracle ROWID data type is used to identify a row. It is stored in the index to reference the row in the table. DB2 does not support the ROWID datatype. If the ROWID is used to identify the rows in a particular table, perhaps the primary key of the table can be used instead. However, if an application requires an 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 13 bytes bit data (CHAR(13) FOR BIT DATA) that are unique across the system. The **GENERATE_UNIQUE** function is persistent across database reorganization and migrations. Below is an example of how it can be used in a insert statement. When it is used as the primary key.

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

Appendix D provides sample code for converting ROWID to **GENERATE_UNIQUE** and triggers. Please note that if the column is to be generated in a trigger, the column cannot be the primary key of the table since a primary key has to be defined in the insert statement.
**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 acquiring a shared lock 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 interval before the deadlock detector wakes up to check for deadlocks in the system. By doing so, DB2 UDB returns immediately to the application with an sqlcode -911 and sqlstate 40001 if the locking resource is blocked. Note that LOCKTIMEOUT affects all applications connected to the database.

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

**Temporary Tables**

DB2 UDB does not support temporary tables. The applications currently using the Oracle temporary table may consider using runtime non-persistent data structures, common table expression(CTE), or real tables to achieve equivalent functionality. It will be supported in a future release.

*The Common table expression* is similar to an in-line view in a statement. It can be used in query and insert statements. Temporary tables can be mapped into common table expressions as illustrated in the following example. PAYLEVEL and PAYBYED are common table expressions that are subsequently used in the actual SELECT statement. The Common table expression is only persistent for the duration of the statement.

```sql
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 HIREEYEAR = YEAR_OF_HIRE
    AND TOTAL_PAY < AVG_TOTAL_PAY;

One can replace temporary tables with data structures in the application, as an efficient and clean way of storing staging information. Like temporary tables, they are cleaned up when the applications exit.

The applications can also simulate the Oracle temporary tables with permanent tables in DB2 UDB using the LOGGED INITIALLY clause on the CREATE TABLE STATEMENT. The clause causes DB2 UDB to not log the changes to the table, hence removing the cost of logging. However, the application does remove the table and the table is not recoverable in the case of system crash.

All meta data in DB2 UDB are stored in the System Catalog. To avoid contentions in the System Catalog, try not to start all tables with the same prefix like TEMP. Instead, try to scatter the table names around the alphabet so they do not cause contentions in the system catalog when all of the meta data tables are on the same page.

**DECODE and CASE expression**

DB2 does not have decode statement. Instead, it has the case expression. The mapping of DECODE and CASE expressions is very direct:

\[
\text{DECODE (condition, case1, assign1, case2, assign2, \ldots, default)} \\
\text{CASE condition} \\
\quad \text{WHEN case1 THEN assign 1} \\
\quad \text{WHEN case2 THEN assign 2} \\
\quad \ldots \\
\quad \text{ELSE default} \\
\text{END}
\]

The CASE expression can be used for more than just mapping decode statements. Since DB2 UDB triggers can contain only SQL statements, the CASE expression can be used to translate logic in Oracle triggers.
**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. Note `TRUNCATE TABLE` is a DDL command and it cannot be rolled back.

DB2 UDB does not support `TRUNCATE TABLE`. The solution is to delete all rows on the table with the `DELETE` statement. If the table is very large, the application can use the `NOT LOGGED INITIALLY` option for performance reasons. The resource used by the table is not reclaimed when the rows are deleted. The container spaces used by the table will stay assigned to the table.

If reclaiming space is important, or the table is too large to delete row by row, the application can simulate `TRUNCATE TABLE` by importing an empty file to replace the table content. This will reclaim the space used by the table, leaving associated information (such as an index or authorizations) intact. The following is a sample of the UNIX script.

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

**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. BEFORE ROW triggers can only include select, set, and signal statements. AFTER TRIGGERS can include set, signal, inserts, updates, and deletes.

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

Another semantic difference between Oracle and DB2 triggers is when the primary key constraint is checked for an insert statement. Oracle verifies that a primary key exists for the row after all the triggers are fired. DB2 insists on the primary key being defined in the original insert statement. I.e., you must have the primary key in the insert statement and not in the trigger body.

One limitation in DB2 UDB triggers is that the trigger body cannot invoke stored procedures; you can only invoke DB2 functions and user-defined-functions (UDF). The main restriction of UDF is that it cannot contain SQL statements. These will be supported by DB2 in a later release.
In Oracle, the trigger body consists of an anonymous PL/SQL block. In DB2, the trigger body consists of one or more SQL statements. A trigger body containing more than one SQL statement must be enclosed between

`BEGIN COMPOUND AUTOMIC` and `END COMPOUND` (i.e., a compound SQL).

SIGNAL and SET statements can only be used in the trigger body. If you need to migrate triggers that have logic flow, here is a list of migration strategies you can employ to:

- Use a COMPOUND SQL statement,
- Use UDFs and value expressions such as `VALUES(UDF1(UDF2( ),UDF3( )..));`
- Use CASE expressions.
- DB2 triggers allow you to specify the columns of a tables that will cause the trigger to be fired. Split up the trigger based on the affected columns.
- Use the WHEN clause to handle some of the logic flow.
- If there are locally declared variables, eliminate them by using the same expression in the trigger body, or use common table expression for substitution.

Here is a set of examples of DB2 UDB triggers for the same update statement.

```
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.rtype = 2) > 0
                THEN raise_error ('78000','Can not update to super rtype ")
        ELSE 0
    END);
END!.

A convoluted and inadvisable roundabout way of executing complex query logic is to
make OS calls from a UDF. The OS 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.

Stored Procedures

In Oracle, stored procedures must be programmed in PL/SQL. DB2 UDB, however,
allows stored procedures to be in any language supported by the DB2 UDB pre-
compilers (part of the SDK package), including JAVA, C, C++, REXX, Fortran and Cobol.

The DB2 UDB stored procedure is basically the same as any loadable program, except it
is loaded and executed on the server. The stored procedure can be invoked by an EXEC
CALL statement from the client.

There are many methods of communicating between a DB2 UDB stored procedure and
the application. Prior to V6.1, the communication between a DB2 UDB Stored procedure
and the application is via the SQLDA and SQLCA data structures. In V6.1, DB2 UDB
allows the C-like standard parameter to pass between them. There is no need to use the
SQLDA anymore.

Below is a table of the client, the stored procedure code and the create procedure
statement.
<table>
<thead>
<tr>
<th>Action</th>
<th>Oracle PL/SQL</th>
<th>DB2 UDB C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calling code from a client</td>
<td>product.id%type v_id;&lt;br&gt;product.name%type v_name;&lt;br&gt;integer parm;</td>
<td>EXEC SQL BEGIN DECLARE SECTION;&lt;br&gt;char procnams[255]=&lt;br&gt;&quot;/home/sqllib/function/proc!SPInsertRow&quot;;&lt;br&gt;char id[10];&lt;br&gt;char name[20];&lt;br&gt;short idind,nameind;&lt;br&gt;int to_commit;&lt;br&gt;EXEC SQL END DECLARE SECTION;&lt;br&gt;&lt;br&gt;EXEC SQL CALL :procnams USING&lt;br&gt; :to_commit, :name;&lt;br&gt;&lt;br&gt;if (to_commit == true) /* stored procedure OK*/&lt;br&gt;.... /* commit <em>/&lt;br&gt;EXEC SQL COMMIT;&lt;br&gt;&lt;br&gt;else { /</em> the stored procedure failed */&lt;br&gt;&quot; handle error */&lt;br&gt;EXEC SQL ROLLBACK;</td>
</tr>
<tr>
<td></td>
<td>Package1.SP_InsertRow(:v_id,:v_name,:parm);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>.....check parm...</td>
<td></td>
</tr>
<tr>
<td>Create procedure SQL</td>
<td>Create procedure SPInsertRow(</td>
<td>Create procedure SPInsertRow(</td>
</tr>
<tr>
<td></td>
<td>out to_commit integer,&lt;br&gt;out name varchar(20))&lt;br&gt;EXTERNAL NAME 'proc!SPInsert'&lt;br&gt;LANGUAGE C&lt;br&gt;PARAMETER STYLE DB2GENERAL&lt;br&gt;MODIFIES SQL DATA&lt;br&gt;PROGRAM TYPE SUB;</td>
<td></td>
</tr>
</tbody>
</table>
In the new stored procedure invoking method, nested stored procedures are not allowed. This limitation will be lifted in the next release for C, C++ and Java. There is a simple workaround for this by using a wrapper for the nested procedures. For example, sample_import is a stored procedure, but the same procedure is called by another stored procedure. To get around the limitation, create a dummy wrapper for the stored procedure to be invoked directly from the client. All other nested cases should call sample_import_int instead.
One important difference between DB2 UDB and Oracle is that DB2 UDB does not support commit or rollback inside stored procedures. DB2 UDB does not handle transactional management in stored procedure. The commit and rollback have to be handled in the client code. The suggested porting method is to pass back a non-zero sqlcode when a rollback of the transaction is required.

The main advantage of stored procedure is to share the program with multiple users. From a performance perspective, there will be less network overhead because there will be less client-server traffic. The following table shows a quick mapping of a PL/SQL stored procedure to a DB2 UDB C stored procedure. Note that C does not support the %type declaration. Appendix I includes the complete DB2 UDB stored procedure sample program.

Java Stored Procedures

DB2 UDB supports Java stored procedures. SQLJ routines are supported using either the JDBC or embedded SQLJ interfaces. Appendix E contains a Java stored procedure. DB2 UDB also support JAR files containing one or more stored procedures. Use CALL SQLJ.install to install JAR files. For more details consult the Application development Guide.

FENCED and UNFENCED Stored Procedure

DB2 UDB has two types of stored procedures: FENCED and UNFENCED. In the process model section, we discussed the different processes or threads running in the DB2 UDB database server under the INSTANCE user id. For FENCED stored procedures. DB2 UDB will run spun a new process under the user id specified by the database parameter FENCEDID.

For UNFENCED stored procedures, the stored procedures are run in the same memory space as the database server to further improve performance as it now has less inter-process overhead. Performance is the primary reason to use UNFENCED stored procedures instead of FENCED stored procedures. However, DB2 UDB cannot protect data corruption caused by invalid pointers in UNFENCED stored procedure. Therefore, it
is important to have a C language UNFENCED stored procedure well-tested before it is installed.

To create a stored procedure in DB2, the process is identical to creating an application program. In addition, the executables must be placed on the server. 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`. FENCED is the default mode for a DB2 UDB stored procedure.

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, Oracle implementation can be best mapped to an UNFENCED version of a stored procedure.

**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.

In DB2 UDB, the users can register stored procedures in the System Catalog using the `CREATE PROCEDURE` statement. If the stored procedure is not installed in the default directory, the `CREATE PROCEDURE` can be used to direct DB2 UDB on the path of where to load the stored procedure. The `CREATE PROCEDURE` allows users to store the interface definitions. Users must maintain the source and object code. Note that the `CREATE PROCEDURE` must be used for JAVA stored procedures.

To port Oracle stored procedures, one must first choose which target language the stored procedure should be migrated to from PL/SQL. The application can choose DB2 UDB FENCED or UNFENCED stored procedures. However, if the stored procedure is very small (for example a couple of SQL statements) it may be easier (and with not much performance difference) to translate it to embedded SQL programs on the client side.

**Select from DUAL**

To get system information, such as SYSDATE, Oracle provides a dummy table call `DUAL`. In DB2 UDB, convert the query to a `VALUES` clause or create a simple assignment statement from special registers. Or, for dynamic applications, use a statement like “select current date from syscat.sysdummy1” to retrieve the values.

```
Oracle          DB2

select SYSDATE from SYS.DUAL variable = VALUES(CURRENT DATE)
or
select CURRENT DATE from sysibm.sysdummy1;
```
NVL

In Oracle, function NVL provides a conversion of NULL fields to specified values. NVL(TO_CHAR(MANAGER_ID), 'No Manager') converts all of the NULL manager_id column to a string and AVE(NVL(GRADE,0)) takes the average of column GRADE and treats the NULL as 0. Use the COALESCE(MANAGER_ID,'No Manager') in DB2 to replace NVL sample above.

Rename

Oracle supports rename table, view, and sequences. DB2 supports only rename tables. Applications that use the rename facility for view need to drop and recreate the view.

Alter Table

Both DB2 UDB and Oracle support the ALTER TABLE command. However, there are differences in what can be changed in an ALTER TABLE command.

The following table gives a quick summary of these differences.

<table>
<thead>
<tr>
<th>Function</th>
<th>Oracle</th>
<th>DB2 UDB V6</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, only if containing only NULL or no rows</td>
<td>no</td>
</tr>
<tr>
<td>Change column type</td>
<td>yes, only if containing NULL 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 here is no NULL in the column</td>
<td>no</td>
</tr>
<tr>
<td>Able/disable constraints</td>
<td>yes</td>
<td>no</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, only if all columns contain NULLs</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Array Fetches/Inserts

For Embedded or Static SQL, DB2 UDB does not support array inserts or fetches. However, the DB2 Call Level Interface (CLI) support host array fetches and inserts.

Objects

Oracle 8 supports the concept of objects. An object contains attributes and methods. Attributes are static, similar columns in a table. Methods are functions that act on the attributes. All attributes and methods in Oracle 8 are public (i.e., no private attributes or methods). You can modify the methods of an object, but you cannot alter attributes of an object.

DB2 supports typed tables where a column can be a composite type. Oracle 8 objects can be converted using typed tables and user-defined functions to achieve the identical functionality.
Application Development

A programming language is required in addition to 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. C and C++ are general programming languages that can be used by any application and include access to databases. DB2 UDB does not have an SQL extension like PL/SQL. DB2 UDB supports all the commonly used programming languages such as C, C++, REXX, Cobol, Fortran and Java. DB2 UDB also supports stored procedures in all of these languages. Oracle applications and stored procedures written in PL/SQL will need to be translated into one of these languages when porting to DB2 UDB. Currently, most typical translations map from PL/SQL to C. For that reason, all the samples here are C based.

DB2 databases can be accessed via Embedded SQL applications (static or dynamic) and/or a callable SQL interface called DB2 Call Level Interface (CLI). The latter 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) specifications.

DB2 UDB does not have a direct mapping to the Oracle Call Interface (OCI). Applications written using OCI should be converted to use CLI interfaces. Appendix I contains samples mapping OCI to ODBC interface.

C / C++

C is the most commonly used programming language with DB2 UDB, with Java currently becoming popular. All sample code in this porting guide is C based. The Oracle Proc*C/C++ Pre-compiler 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 pre-compiled with the following options to facilitate migration to DB2:

\[ \text{FIPS=YES} \ \text{MODE=ANSI} \ \text{DBMS=V7} \ \text{ORACA=NO}. \]

In the previous data types section, the table summarizes mapping DB2 datatype and C type definitions.
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 you declare size N+1 in C as C requires a null terminator for its string.

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

The following table contains the logic flow mapping of PL/SQL to C:

<table>
<thead>
<tr>
<th>PL/SQL Logic Flow Statement</th>
<th>C Logic Flow Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF condition</td>
<td>if (expression)</td>
</tr>
<tr>
<td>THEN</td>
<td>{</td>
</tr>
<tr>
<td>statements;</td>
<td>}</td>
</tr>
<tr>
<td>END IF;</td>
<td>}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PL/SQL Logic Flow Statement</th>
<th>C Logic Flow Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF condition</td>
<td>if (expression)</td>
</tr>
<tr>
<td>THEN</td>
<td>{</td>
</tr>
<tr>
<td>statements;</td>
<td>}</td>
</tr>
<tr>
<td>ELSE</td>
<td>else</td>
</tr>
<tr>
<td>statements;</td>
<td>{</td>
</tr>
<tr>
<td>END IF;</td>
<td>}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PL/SQL Logic Flow Statement</th>
<th>C Logic Flow Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOOP</td>
<td>do</td>
</tr>
<tr>
<td>statements;</td>
<td>{</td>
</tr>
<tr>
<td>EXIT WHEN condition;</td>
<td>}</td>
</tr>
<tr>
<td>END LOOP;</td>
<td>}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PL/SQL Logic Flow Statement</th>
<th>C Logic Flow Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHILE condition</td>
<td>while (expression)</td>
</tr>
<tr>
<td>LOOP</td>
<td>{</td>
</tr>
<tr>
<td>statements;</td>
<td>}</td>
</tr>
<tr>
<td>END LOOP;</td>
<td>}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PL/SQL Logic Flow Statement</th>
<th>C Logic Flow Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>FOR index IN lower_bound .. upper_bound LOOP statements; END LOOP;</td>
<td>FOR (init_expression;loop_condition;loop_expression) { statements; }</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PL/SQL Logic Flow Statement</th>
<th>C Logic Flow Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>FOR record_name IN cursor_name LOOP statements; END LOOP;</td>
<td>There is no logic flow structure that handles an implicit OPEN, FETCH and CLOSE cursor.</td>
</tr>
</tbody>
</table>
Java

You can write DB2 client applications and stored procedures in Java as we have discussed before. The syntax of Java is identical to C. Appendix E is a sample program of Java client calling a Java SP.

Static SQL

Both Oracle and DB2 UDB support embedded static SQL. Static SQL must be pre-compiled and bound prior to execution. Input and output of embedded SQL are passed via host variables and all DB2 UDB embedded static SQL must start with the keyword EXEC. Since compilation time can be significant, static SQL provides the most performance for running an application.

After compilation, DB2 UDB embedded SQL needs to be bound to a particular database. This 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 the next section.

The user who executes the application requires EXECUTE privilege. In Appendix C, we have included a sample program game, to update and retrieve data from the sample database.

As shown by the table below, the syntax of the embedded static SQL of the two databases is almost identical.

<table>
<thead>
<tr>
<th></th>
<th>Pro C with Embedded Static Oracle SQL</th>
<th>C Function with Embedded static DB2 SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declare</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</td>
<td>EXEC SQL INSERT INTO</td>
</tr>
<tr>
<td></td>
<td>table_name(col1) VALUES(:host_var1);</td>
<td>table_name(col1) VALUES(:host_var1);</td>
</tr>
<tr>
<td>Error handling</td>
<td>arg2_out = sqlda.sqlcode;</td>
<td>arg2_out = sqlda.sqlcode;</td>
</tr>
</tbody>
</table>
Embedded Dynamic SQL

Both Oracle and DB2 UDB support dynamic SQL. You can use the embedded dynamic SQL or the Call Level Interface (CLI). Embedded dynamic SQL still requires the pre-compile/compile/link phase. However, the binding and the selection of the access plan is done at run time. 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, then use the dynamic instead of static SQL.

Dynamic SQL also takes advantage of the latest table statistics for access plan evaluation. DB2 provides support for dynamic SQL methods 1-3, dynamic statements such as PREPARE, EXECUTE, EXECUTE IMMEDIATE, and DESCRIBE. A statement may also be prepared once and executed multiple times using different parameter markers and bind descriptors.

DB2 also provides an SQLDA for processing Method 4-type queries (also known as Varying-List Selects in DB2). The fundamental difference between the Oracle and DB2 applications will likely relate to differences in the SQLDA structure. The differences/similarities of the SQLDA are discussed below. A sample DB2 Method 4 application is provided in Appendix G - Sample DB2 Dynamic Embedded Application.

DB2 UDB caches previously compiled dynamic SQL statements. This capability can greatly improve the performance of the dynamic SQL. CLI also has many optimization techniques that allow the application to minimize the compilation cost and the network traffic cost.

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. These manage the session-related information through handles. Instead of using host variables, dynamic SQL uses parameter markers. Dynamic SQL requires the application to bind programming variables to the SQL statement. In C terminology, binding a variable is giving DB2 UDB a pointer, i.e., a.b address, of where to retrieve or return the data.

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

Appendix H contains two CLI stored procedure samples. They illustrate how environment and connection handles are passed between CLI clients and stored procedures. They also illustrate the DB2 SQL parameter passing style now supported in V6.
Outer Join

DB2 UDB supports three types of outer joins: right, left, and full. The syntax of outer joins differ slightly between Oracle and DB2.

The following example demonstrates the differences.

<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</td>
<td>SELECT A.last_name, A.id, B.name</td>
</tr>
<tr>
<td>FROM emp A, Customer B</td>
<td>FROM emp A RIGHT OUTER JOIN customer B</td>
</tr>
<tr>
<td>WHERE A.id(+) = B.sales_rep_id;</td>
<td>ON A.id = B.sales_rep_id;</td>
</tr>
</tbody>
</table>

Cursors

Cursors are very similar in Oracle and DB2 UDB. In PL/SQL, you have the ability to declare implicit cursors (see the section above on C/C++).

The table below gives a quick summary of the syntax of the PL/SQL cursor and the embedded SQL cursor.

<table>
<thead>
<tr>
<th>Oracle PL/SQL Cursor</th>
<th>Oracle Embedded SQL Cursor</th>
<th>DB2 UDB Embedded SQL Cursor</th>
</tr>
</thead>
<tbody>
<tr>
<td>CURSOR cursor_name(parm1 NUMBER) IS</td>
<td>EXEC SQL DECLARE cursor_name CURSOR FOR</td>
<td>EXEC SQL DECLARE cursor_name CURSOR FOR</td>
</tr>
<tr>
<td>SELECT col 1 FROM table</td>
<td>SELECT col1 FROM table WHERE col2='123';</td>
<td>SELECT col1 FROM table WHERE col2='123';</td>
</tr>
<tr>
<td>WHERE col2=PARM1;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPEN cursor_name(parm1);</td>
<td>EXEC SQL OPEN cursor_name</td>
<td>EXEC SQL OPEN cursor_name;</td>
</tr>
<tr>
<td>FETCH cursor_name INTO host_var1;</td>
<td>EXEC SQL FETCH cursor_name INTO :host_var1;</td>
<td>EXEC SQL FETCH cursor_name INTO :host_var1;</td>
</tr>
<tr>
<td>CLOSE cursor_name;</td>
<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 FOR N ROWS to control the number of rows returned.

See the section ROWCOUNT for details.

<table>
<thead>
<tr>
<th>Oracle Cursor Attribute</th>
<th>DB2 UDB Mapping</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ISOPEN</td>
<td>EXEC OPEN CURSOR cursor_name; if(SQLCODE == -502) { ... }</td>
<td>Boolean attribute. Returns TRUE if the cursor is open</td>
</tr>
<tr>
<td>%NOTFOUND</td>
<td>if (SQLCODE == 100 )</td>
<td>Boolean attribute. Returns TRUE if the fetch does not return a row</td>
</tr>
<tr>
<td>%FOUND</td>
<td>if (SQLCODE == 0 )</td>
<td>Boolean attribute. Returns TRUE if the fetch returns a row</td>
</tr>
<tr>
<td>%ROWCOUNT</td>
<td>use a counter variable or OPTIMIZE FOR N ROWS or FETCH FOR N ROWS</td>
<td>Numeric attribute. Returns the number of rows returned so far.</td>
</tr>
</tbody>
</table>

DB2 UDB and Oracle handle cursor exit condition differently. The following provides a quick comparison of how the three programming environments handling cursor existing conditions.

<table>
<thead>
<tr>
<th>Oracle PL/SQL Cursor Attributes</th>
<th>Oracle Embedded SQL return codes</th>
<th>DB2 UDB SQL Return Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOOP</td>
<td>status = SUCCESS;</td>
<td>status = SUCCESS;</td>
</tr>
<tr>
<td>FETCH cursor_name INTO var1;</td>
<td>while (status == SUCCESS)</td>
<td>while (status == SUCCESS)</td>
</tr>
<tr>
<td>EXIT WHEN cursor_name%NOTFOUND;</td>
<td>{</td>
<td>{</td>
</tr>
<tr>
<td>.....</td>
<td>EXEC SQL FETCH cursor_name</td>
<td>EXEC SQL FETCH</td>
</tr>
<tr>
<td>END LOOP;</td>
<td>INTO :var1;</td>
<td>cursor_name INTO :var1;</td>
</tr>
<tr>
<td></td>
<td>status = ORC_CODE;</td>
<td>status = SQLCODE;</td>
</tr>
<tr>
<td></td>
<td>if (status == SUCCESS)</td>
<td>if (status == SUCCESS)</td>
</tr>
<tr>
<td></td>
<td>{ ... }</td>
<td>{ ... }</td>
</tr>
<tr>
<td></td>
<td>else if (status==NO_DATA_FOUND)</td>
<td>else if (status == 100)</td>
</tr>
<tr>
<td></td>
<td>{ EXEC SQL CLOSE cursor_name}</td>
<td>{ EXEC SQL CLOSE</td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>cursor_name}</td>
</tr>
</tbody>
</table>

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 Reference.
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 /* Mapping to DB2 SQLDA */
{
    long N;  /* Descriptor size in number of entries */  /* sqlda->sqln */
    char **V; /* Ptr to Arr of addresses of main variables */ /* sqlda->sqlvar[i].sqldata */
    long *L; /* Ptr to Arr of lengths of buffers */ /* sqlda->sqlvar[i].sqllen */
    short *T; /* Ptr to Arr of types of buffers */ /* sqlda->sqlvar[i].sqltype */
    short **I; /* Ptr to Arr of addresses of indicator vars */ /* sqlda->sqlvar[i].sqlind */
    long F; /* Number of variables found by DESCRIBE */ /* sqlda->sqld */
    char **S; /* Ptr to Arr of variable name pointers */ /* sqlda->sqlvar[i].sqlname.data */
    short **Y; /* Ptr to Arr of max lengths of var. names */
    short **Z; /* Ptr to Arr of current lengths of var. names */
    short **X; /* Ptr to Arr of ind. var. name pointers */
    short *Y; /* Ptr to Arr of max lengths of ind. var. names */
    short *Z; /* 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 */  
};

union sql8bytelen
{   
    long reserve[2];   /* reserved for future 8 byte lengths. */  
    long sqllonglen;   /* this is what is currently used */  
};

struct sql8bytelen len  
{   
    long reservel[2];   /* reserved for future 8 byte lengths. */  
    long sqllonglen;   /* this is what is currently used */  
};

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

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. */  
};

struct sqldata           /* 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 */  
};

struct sqlda            
{   
    char sqldaid[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 sqldata sqlvar[1];  /* first SQLVAR element */  
};
```

SQLCA / ORACA

SQLCA is the SQL Communication 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</th>
<th>DB2 Messages and Warnings</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLCA: SQL Communications Area</td>
<td>SQLCA: SQL Communications Area</td>
</tr>
<tr>
<td>sqlca.sqlerrm.sqlerrml: in SQLCA contains the length of an error message.</td>
<td>sqlca.sqlerrm.sqlerrml: in SQLCA contains the length of sqlerrmc</td>
</tr>
<tr>
<td>sqlca.sqlerrm.sqlerrmc: in SQLCA contains the error message text</td>
<td>sqlca.sqlerrm.sqlerrmc: in SQLCA contains the error message token that need to be decoded in order to display a friendly message or warning</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 provides additional diagnostic information, which is specific to Oracle and therefore not supported in DB2.

**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>A lock rollback was detected (rc=2 timeout, rc=68 timeout)</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</td>
</tr>
<tr>
<td>-1,427</td>
<td></td>
<td>More than one row returned</td>
</tr>
</tbody>
</table>

The syntax of error handling of embedded SQL and Oracle PL/SQL are quite different, but they function the same way. Embedded SQL allows the application to specify the label to jump to when an error occurs instead of always going to the `EXCEPTION` label.

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.

<table>
<thead>
<tr>
<th>Oracle PL/SQL Exception Handling</th>
<th>DB2 UDB Embedded SQL C Exception Handling</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT ...</td>
<td>EXEC SQL WHENEVER NOT FOUND GO TO error;</td>
</tr>
<tr>
<td>...</td>
<td>EXEC SQL WHENEVER SQLWARNING CONTINUE;</td>
</tr>
<tr>
<td>EXCEPTION</td>
<td>EXEC SQL UPDATE..</td>
</tr>
<tr>
<td>WHEN NO_DATA_FOUND THEN</td>
<td>status = SQLCODE;</td>
</tr>
<tr>
<td>ROLLBACK;</td>
<td>return(status);</td>
</tr>
<tr>
<td>WHEN OTHERS THEN</td>
<td>...</td>
</tr>
<tr>
<td>ROLLBACK;</td>
<td>error:</td>
</tr>
<tr>
<td></td>
<td>status = SQLCODE;</td>
</tr>
<tr>
<td></td>
<td>EXEC SQL ROLLBACK;</td>
</tr>
<tr>
<td></td>
<td>return(status);</td>
</tr>
</tbody>
</table>

Oracle supports *user-defined errors* and the application can raise the exception. DB2 UDB does not support user-defined errors in the SQL error handling structure. However, the application can do the same thing with the standard logic flow in the programs.
<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:   | Do not declare, and allow the Oracle 7 Server to raise them implicitly | SELECT...
 ..
  EXCEPTION
  WHEN NO_DATA_FOUND THEN
  ROLLBACK;
  WHEN OTHERS THEN
  ROLLBACK; |
| Predefined Error     | NO_DATA_FOUND is ORA-01403.                       |                                                     |                                                   |
|                      |                                                   |                                                     |                                                   |
| Non-Predefined Error | Any other standard Oracle server 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.'); |
|                   |                                                   |                                                     |                                                   |
| User-Defined Error   | A condition that the developer determines is abnormal | Declare within the declarative section, and raise explicitly | DECLARE exception1 EXCEPTION;
  ...
  BEGIN...
  RAISE exception1;
  ...
  EXCEPTION
  WHEN exception1 THEN
  DBMS_OUTPUT.PUT_LINE('Error.'); |

DB2 UDB exceptions are communicated mostly through the SQLCODE. The *EXEC SQL WHENEVER* clause redirects the flow in each condition. Note that the syntax is the same for Oracle embedded SQL.
<table>
<thead>
<tr>
<th>DB2 UDB Exception</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLERROR</td>
<td>Identifies any condition where SQLCODE &lt; 0</td>
<td>... EXEC SQL WHENEVER SQLERROR GO TO Error; ... EXEC SQL UPDATE... status = SQLCODE; if (status == SUCCESS) .... Error: status = SQLCODE; EXEC SQL ROLLBACK; return(status);</td>
</tr>
<tr>
<td>SQLWARNING</td>
<td>Identifies any condition where SQLWARN(0)=W or SQLCODE &gt;0 but not equal to 100</td>
<td>... EXEC SQL WHENEVER SQL WARNING CONTINUE; ... EXEC SQL UPDATE... status = SQLCODE; return(status);</td>
</tr>
<tr>
<td>NOT FOUND</td>
<td>Identifies any condition where SQLCODE = 100</td>
<td>... EXEC SQL WHENEVER NOT FOUND GO TO Error; EXEC SQL UPDATE... status = SQLCODE; return(status); .... Error: status = SQLCODE; EXEC SQL ROLLBACK; return(status);</td>
</tr>
</tbody>
</table>


Constraints

Both DB2 UDB and Oracle support the primary key, unique, and referential integrity constraints with minor differences. Primary key constraint is identical between the two. Columns specified in a unique constraint must be defined as NOT NULL in DB2 UDB.

DB2 UDB does not allow disabling/abling one constraint at a time. However, the user can disable/able a group of constraints at a time. For example, one can disable/able all the constraints of a table or disable/able all the RI constraints of a table. The command is 

SET INTEGRITY. The following example shows how to disable/able all the constraint checking for table T1.

```
db2 set integrity for T1 off;
....do things;
db2 set integrity for T1 check immediate;
```

Deferred Unique Constraints

Deferred unique constraints should not be an issue with database and applications ported from Oracle to DB2 UDB V5 or higher. However, if you did your application migration from Oracle to DB2 UDB V2, you may notice the difference in behavior in unique constraint checking.

DB2 UDB V5 or higher enforces unique constraint checking at the statement boundary while DB2 UDB V2 enforces unique constraint checking as the column is being updated. Say you want to increase all the values for a column by 1 and the existing values are consecutive. In V2, you will get a duplicate error as DB2 UDB tries to update the first row. In V5 or higher, DB2 will defer the checking until the entire update statement is finished. If you encounter this problem because you migrate your applications from Oracle to DB2 UDB V2, then porting to V6 will fix this problem.

Referential Integrity

Both DB2 and Oracle support foreign keys where the referencing table's primary key is referenced by the child table. DB2 UDB, however, allows referential integrity (RI) to apply to any unique constants, not just the primary key. Note 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 of the applications check table dependencies by fetching from many dependent tables. They can be translated to RI constraints in a table declaration. This can significantly simplify the applications and also increase concurrency in DB2 UDB as we explain in the next section. DB2 UDB allows the SET NULL option on delete addition to be set to CASCADE, RESTRICT, and NO ACTION. The default for Oracle is DELETE RESTRICT and the default for DB2 is DELETE NO ACTION.
Concurrency Control

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

If you port your applications and find that the behavior between the two systems is similar, then you do not need to concern yourself with the topic of concurrency. However, if your applications involve frequent access to the same tables, you may find your applications behave differently.

To get the best result, sometimes it is worth reworking the applications to achieve the best parallelism in DB2 UDB. Once you understand how the concurrency control works in DB2 UDB V6, it should be obvious how to rework the application.

Concurrency and Locks

Transaction is an AUTOMATIC unit of work that is committed or rolled back. Both DB2 UDB and Oracle support AUTOMATIC transactions. Both DB2 and Oracle have row-level locking (as opposed to page-level locking in SQLServer). The difference is when the locks are acquired.

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

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

Due to these differences, ported applications that have updaters and readers accessing the same data from the same table may experience more lockwait time in DB2. This is not an issue if applications each access their own set of tables.
**Isolation Levels**

Before looking at how to improve concurrency with 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 are unlikely to be changed by another application. It is up to the applications to take care of the situation in which the data are 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 applications request to read the data, they will get the version from the rollback segment. Once the update transaction commits, the rollback segment version is erased and all other applications will 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, each to suit the need a particular type of application. The application can select the level of concurrency control to provide the proper level of isolation control. Here is a brief description of each isolation level. For more details, consult the DB2 UDB V5 Admin 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 updaters from changing the data. It also prevents phantom rows from being inserted or deleted. If you fetch from a cursor twice, you are guaranteed to see the same rows.

- **Read Stability (RS)** - Like RR, this level of isolation guarantees that the rows read by an application remain unchanged in a transaction. However, it does not prevent new rows from appearing during the transaction (also known as phantom rows). Oracle’s concurrency implementation most assembles RS level of locking.

- **Cursor Stability (CS)** - This level guarantees that only the row of a table will not change while your cursor is positioned on that row. This means that 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 DB2 UDB has in memory, even through the data is part of another transaction. The data is labeled as "dirty" because if the updater rolls back, the data read would be incorrect. 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 shared lock on the object. Hence, it does not prevent the data from being updated.

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

With the exception of Uncommitted Read transactions, DB2 UDB can guarantee the data the application read will 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 UR isolation levels used). This can reduce concurrency in the system if there are a lot of applications accessing 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 the same table access. 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 RI constraint 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 acquired in the database.

**Optimistic Locking**

If the application must have the same locking behavior when ported from Oracle, it can use the optimistic locking scheme to simulate the behavior. It is a technique used to reduce lockwait time. The cost here is to know how to handle the situation when the fetched data is changed.
The technique requires the table to have a field that can uniquely identify when the last change took place. *Timestamp* is the most typical example. Read the row and commit right away to release all the locks. This will allow other applications 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, i.e. the row has not changed since the last fetch, the application can go ahead and update the row. Otherwise, handle the exceptional situation.

```
Fetch row including the timestamp identifier
commit; (release all the locks)
... think time...
start transaction
fetch the row and compare the timestamp from the one retrieve last
if timestamps match
    do the work;
else
    handle the situation;
commit;
```

**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 table locks automatically without an explicit user request. DB2 performs **lock escalation** by converting many row locks to a table lock if it perceives that there are too many locks held at that time. The optimizer may also request a table lock if it thinks it is more efficient for the 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. It may temporarily slow the system down. However, once it is done, the application may run faster, because DB2 UDB no longer needs to acquire individual locks for each row.

The side effect of lock escalation is its impact on other applications running concurrently. It may block other applications from accessing the table. For example, if application A holds a shared table lock on a table, it blocks other applications from updating any rows in the table, because they would need to acquire an exclusive lock on a row in that table.

Lock escalation is DB2 UDB’s way of handling exceptional situations where the lock requests are much higher than usual. Lock escalation should not be a frequent event in a production environment. The database parameter that controls the amount of memory to
store locks the system is LOCKLIST, the number of 4K blocks used in DB2 UDB to hold the currently locks held by applications. To have predictable results, allocate sufficient memory, as specified by LOCKLIST. Acquire a table lock explicitly in your application when you feel it is appropriate.

Use Database Snapshot Monitor to determine if you have lock escalation for a given period and increase (or decrease) your LOCKLIST appropriately.

**DEADLOCKS**

As a possible result of different concurrency controls between 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 shared lock for readers, updaters may be blocked where that was not the case using Oracle.

A deadlock occurs when two or more applications are waiting for each other but none can proceed because each has locks that are required by others. The only way to resolve a deadlock is to roll back one of the applications.

DB2 UDB resolves deadlocks automatically, without user intervention. The DB2 UDB Database server 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 in the system. If so, it will pick a victim and roll back the transaction. Note the victim can be a read-only transaction. If a transaction is rolled back due to a deadlock, DB2 UDB will return an error code (SQLCODE -901, SQLSTATE 40001) indicating a rollback due to deadlock.

Oracle does not have a deadlock detector to resolve deadlocks. Applications use the NOWAIT clause or set a time-out for the session to avoid deadlocks.

You can change the frequency of the checks by the deadlock detector by setting the database parameter DLCHKTIME. The detection time applies to all applications for that database. You can also set time-out on a specific SQL statement.

You can monitor if a deadlock has occurred by using the Database 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 a deadlock Event Monitor. 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 to reschedule or redesign your application programs.
**How to Improve Concurrency**

To maximize the concurrency in DB2 UDB, keep the following things in mind.

- Commit your read-only transactions once in a while to release the shared locks acquired by DB2. In Oracle, this is not required since it does not acquire any locks for query.

- Use *Uncommitted Read (UR)* and *Cursor Stability (CS)* whenever possible. Check return codes for roll backs due to deadlocks.

- Use optimistic locking to reduce lockwaits in the system.

- Use *RI* to reduce the number of cursors used in the application.

- If required, reschedule your applications so that 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 LOCKWAIT value from the Database Monitor.

- Use the Database Monitor to find out how much of the LOCKLIST is used for your environment. Increase the LOCKLIST if your applications are encountering LOCK ESCALATION. Decrease your LOCKLIST if you do not need the space and you can make better use of the extra memory space elsewhere.

- Use the Database Snapshot Monitor 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 the previous architecture section, we briefly discussed the differences between how DB2 UDB and Oracle log the changes to the database. DB2 UDB does not have the rollback segments, logs read and updating database transactions.

DB2 UDB stores the changes to the database in log files. It saves log files in the directory specified by the database configuration parameter *LOGPATH*. Log files can be system files or raw device.
You can control the amount of LOGs DB2 holds using the `LOGPRIMARY`, `LOGSECONDARY`, and `LOGFILESZ` database parameters. Consult the DB2 UDB V5 Administration Guide for details. You can reduce the amount of information logged by DB2 by using the `NOT LOGGED INITIALLY` clause in your database updates. This can significantly improve the speed of the updates. However, keep in mind that the data updated during the `NOT LOGGED INITIALLY` period is not recoverable. One example of this is when an application first populates the table using the `IMPORT` command.

**Archiving**

DB2 allows three types of logging options, which are controlled by the two database configuration parameters `LOGRETAIN` and `USEREXIT`. If `LOGRETAIN` is OFF, commonly known as circular logging, DB2 UDB will reuse the log files once all the log files are full, assuming that there are no outstanding transactions in the log files. A log file is no longer active when all the data pages associated with the log records in that log file are written to permanent storage. 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. No user intervention is required for circular logging and the `USEREXIT` parameter is ignored.

Setting `LOGRETAIN` ON without `USEREXIT` tells DB2 UDB not to reuse the log files when the files are full. The database activities are blocked until the used log files are moved from the `LOGPATH` directory. `LOGRETAIN` ON with `USEREXIT` starts a stored procedure named `db2uexit` which can contain many activities. One of them includes moving the used log files to the archiving directory. This is the typical operational database behavior for a production system.

Oracle logging can be best mapped to the `LOGRETAIN` ON with `USEREXIT` defined.
Database Administration

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

Command Center

The Command Center is DB2 UDB’s graphical interface for database administrators and application programmers. Command Center runs on the Windows platforms and the Web. It is roughly equivalent to the Enterprise Manager in Oracle.

From the Command Center, you can perform almost all administrative duties by invoking different sub components such as Control Center, Journal, smartguides and monitors. For example, you can develop and schedule scripts using the Script Center, or administer a particular DB2 instance using the Control Center. Look at database snapshots and monitor events with the monitors. This sections gives a quick overview of capabilities of each component.

The main page of the Command Center
Control Center

The Control Center can save you a lot of time by looking up the syntax of various DB2 UDB commands and by reducing the learning curve for getting started with DB2 UDB. Use the Smartguide to change the database configuration parameters after creating a new database, because the default configuration parameters are likely to be too small for a typical database.

The Control Center provides drill-down capability for each database object. You can access the database snapshots, change configuration parameters, and create and manage objects in the database. You can also get a database snapshot and define events for the Events Monitor. Right click on an object to find out the different functions you can perform on the database object.

In DB2 V6, you are able to save the commands issued by the Control Center in a script. 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 the ability to add new servers and databases to the node catalog and database catalog. This should make the task of remote administration much more pleasant.
Script Center

Complicated Oracle SQL Plus scripts should be ported to the DB2 UDB Script Center. Script Center can be invoked from the Command Center. Script Center provides the ability to have scripts that contain SQL 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.

Journal

The Journal is used to monitor system 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 the messages seen by the database 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 CLP. Enter db2 to start CLP or just type db2 <command> to enter a CLP command. CLP also has the advantage of running on all DB2 UDB client platforms. CLP script can contain operating system commands and other executables. CLP is similar to SQL Plus in Oracle. However, some of the SQL Plus scripts can be best ported to Command Center scripts instead to a CLP scripts.

CLP is a great tool for querying the syntax of a SQL command by typing a question mark after the SQL. Typing "connect ?" will return the syntax of the SQL. Another use for CLP is to query the SQLCODE and SQLSTATE. "DB2 ? SQL08001" returns the explanation of sqlcode 8001.

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 statement, invoke the script using the -tvf flags.

```
db2 -tvf runstuf.clp
```

If you have multiple SQL statements in a CLP script, it is necessary to have a terminator other than a semicolon to indicate the end of the script. For example, if you have multiple
SQL statements separated by semicolons and use ! to indicate the end of the script, invoke the script using the flags

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

**Authentication**

DB2 utilizes the operating system for its authentication. You specify a user group to have SYSADM or SYSCTL privileges. When a user is assigned to an user group, the user inherits the privileges of the group. For example, if you define a group called ADMINGRP on the system and set the database manager parameter SYSADM to be ADMINGRP, then all users belonging to that group will have SYSADM privileges.

For the NT platform, DB2 UDB uses the system user authentication to control database access. If you log on to the server machine with a user ID that belongs to a group that has database access, DB2 UDB will allow you access to the database. To gain remote access to an NT database server, a user must provide a user ID and password to gain access to the database.

**Access Privileges**

Oracle has the concept of roles. A set a 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. Use GRANT PUBLIC to allow access by all users.

**Data Management**

DB2 UDB supports System Managed Space (SMS) and Database Managed Space (DMS) tablespaces. 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 like 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.

If you have table spaces with different page sizes, make sure you create temporary table space for each page size, as DB2 UDB has to have temporary table space for sort. Use SMS to simplify the administrative task.

**SMS Table Space**

DB2 supports the System Managed Space (SMS). It is the default storage used when the Create Database is issued. SMS uses the operating file system files to store the data. It is the easiest way of creating and managing a database. The limitations of SMS often
are imposed by the limits of the operating system. For instance, some operating systems do not allow the size of the container to exceed 2GB.

Since the concept of SMS does not exist in Oracle, the question is when should SMS be used during porting? SMS uses operating system files for containers and the containers grow as the data is entered into the database; they are not pre-allocated. This means you do not worry about pre-allocating too much space for the table space. For performance, you can use the extent size of the table space to control how much the container grows each time. The down side 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. The containers have to extended at run time and the data are written to a file rather than to a raw device. The performance difference is around 10% or less. It is not an issue, however, if most of the data in the table space are buffered by the bufferpool.

DB2 UDB recommends using a SMS for system catalog (table space SYSCATSPACE) and all temporary table spaces (TEMPSPACE1). The reason for doing this is because, in most cases, the system catalog tables are buffered in memory (bufferpool) since they are frequently accessed. Using SMS can reduce the amount of administration required without much performance cost. Temporary table spaces are used by DB2 UDB only when memory is running short. It may not be worth the effort to manage DMS temporary tablespaces.

For user data, applications can use SMS for ease of maintenance and to avoid having to pre-commit 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 the sandbox environment, SMS can save a lot of startup time. Also, if the database is not expected to grow to quickly, SMS may be more than adequate to suit the purpose.

**DMS Table Space**

**DMS** table space is designed for its flexibility, extendibility, and performance. For Oracle users, managing the DMS table spaces should be familiar. Containers can be added to the table space as the database grow. You can specify the containers as belonging to a table space when creating them or later. The containers are pre-allocated when they are added to a table space.

Oracle’s table space maps well to the DB2 UDB DMS table space with raw devices. In additional to raw devices, DB2 UDB also allows the operating files to be containers. When more than one container is defined for a tablespace, DB2 UDB strips the data across the containers. The strip size is the extent size defined for that tablespace. When a new container is added to the table space, DB2 UDB rebalances the containers automatically by spreading the data to the new container. This is the reason we recommend having all containers the same size in a table space so that the applications
can benefit from maximum degree of I/O parallelism. Autorebalance 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 over the storage and allows more efficient access to the storage. For this reason, DMS table spaces may have better performance than SMS table spaces. DMS table spaces are pre-allocated at creation time; this means no overhead is taken up by extending the storage during the run time, which can further improve the performance.

DB2 UDB data storage is completely page based, hence it does not have the fragmentation problem experienced in Oracle. You can always successfully reorganize a DB2 UDB table space, regardless of how full the table space is.

**Multiple Bufferpools**

DB2 UDB supports multiple data buffering areas (*bufferpools*) while Oracle supports only one *data cache* in the SGA. You can assign multiple table spaces to a particular bufferpool. Once assigned, the data (or index) in the table spaces can be cached in the bufferpool to reduce the I/O access to disk.

The data cache rate (hit ratio) is the single most important item that can affect the performance of the database. Hence, the ability to control the hit ratio by allocating different amounts of memory to buffer data in a table space can make a dramatic performance improvement to the database. For example, you can assign a table space to table data and another table space for its indexes. If the access to the table is almost all index probes, one can allocate a large *bufferpool* for the table space that holds the index and have a very small bufferpool for the table space that houses table data. This is like doubling the hit ratio for the table with the same amount of memory.

As you migrate your applications from Oracle to DB2 UDB, spend some effort considering how the database should lay out in the physical system and, at the same time, consider which table spaces should be buffered in bufferpools to get best hit ratio for your data. Since you can alter the bufferpools-table space combination as the system develops, you can make improvements over time as the data access pattern clarifies.

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

The default configuration for *creating database* is to have just one bufferpool (SYSDEFPOOL) used by all table spaces in the system. The size of this bufferpool is set
by the database parameter BUFFPOOL in 4K blocks. Note that the parameter only takes effect if IBMDEFAULTBP is set to -1.

```
db2 update db configuration using buffpool 20000 % set IBMDEFAULTBP size to 20,000 pages
db2 alter bufferpool ibmdefaultbp size -1 %
```

Use the `CREATE BUFFERPOOL` DDL to create more bufferpools.

Large objects are not buffered in bufferpools. If you fetch an LOB column, it will always trigger an I/O. Hence, it is important from the performance perspective to use LOB only when `varchar` type is not big enough.

**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 import file (`replace` option) `dat.tmp`.

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

Since the default DATE formats are not the same in Oracle and DB2, the DATE exported from Oracle needs to be converted to one of the standard formats. The DB2 default is the USA format (mm/dd/yyyy). Since the Oracle DATE type can be converted to DB2 data type DATE, TIME or TIMESTAMP, one sure way is to translate the Oracle DATE to DB2 `TIMESTAMP` format. The import utility will then pick up the relevant part from the `TIMESTAMP` field.
**Load**

Load is the most efficient way of populating the database. Load can receive data either from a file or a named pipe. Load postpones the integrity checking to the end of the load command. Many enhancements have been added to load in DB2 UDB V6 including incremental indexing and returning the tablespace normal state after the command “load-terminate”. Also option delprioritychar is supported by load in V6.

For best performance, build the index during the load so DB2 does not have to read in all the data to rebuild the index which requires a very large sort space.

**Redirected Restore**

One of the common challenges in database administration is to reorganize the database storage space. You cannot drop containers or modify the size of the container directly. Redirected restore allows you to redefine the container’s definitions in a table space during the restore process. Redirected restore is the only way to add containers to SMS table spaces.

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

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

Explain

*Explain* is a database facility for recording the access plan of queries. In both DB2 UDB and Oracle, the process to generate the access plan is similar. The major difference between the two 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 and populate them as you prepare for the 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 Oracle 7. DB2 UDB has always used the rule-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 will be easier to optimize and result in much superior plans. The users no longer have to worry about the details of how to write the best possible SQL statement, the optimizer shoulders much of the responsibility. When using Visual Explain, the users can see the rewritten statement and the plan. Often, the rewritten access plan is not very readable but it usually performs much better than the original.

In Oracle, the *PLAN_TABLE* can be created using the utlxplan_sql script and when the `EXPLAIN <SQL>` statement is issued, the table is populated with the database accessing 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.
(tkprof trace_file username/password explain=yes).

<per session>
set sql_trace=TRUE;
..submit job..
set sql_trace=false;
tkprof ...

To view the access plan, issue the SQL statement:

```
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 in preparation time, prior to the actual execution. In fact, you can get the estimated cost without ever executing a query by setting the `SNAPSHOT` option to `EXPLAIN`. To create the explain tables, use the CLP script `EXPLAIN.DDL` in the `sqllib/misc`. (or `sqllib\misc\`) directory.

```
db2 -tf EXPLAIN.DDL
```

Note the default of `explain` is to capture only embedded static SQL. To turn on the explain facility for dynamic SQL, set the `MODE` to `Yes`. The syntax for the command is:

```
db2 SET CURRENT MODE YES  - capture dynamic SQL statements
```

```
db2 SET CURRENT SNAPSHOT EXPLAIN - capture the access plan without running the statement
```

Once the explain tables are created, access plans for all SQL statements are captured in the explain tables. To access the explain output in a report format, use `db2expln` for static SQL and use `dynexpln` for dynamic SQL under the `sqllib/misc` or `sqllib\misc\` directory. Both work in interactive mode as well as taking input parameters. They are primarily designed to be used in scripts. For interactive mode, it is better to use Visual Explain.
To retrieve access plans for packages created by the user ID mingwu for database called sample, and to place the output in a file called plan.out:

```
db2expln -c mingwu -d sample -s % -p % -o plan.out
```

To get the detailed syntax, type

```
db2expln ?
```

Dynexpln accepts a similar format, except that it takes a file input containing a set of SQL statements, or takes statements from the command line. The following command takes the SQL statements in file myfile and places the access plan in dyn.out. The statements are separated by (;) in myfile.

```
dynexpln -d sample -f myfile -o dyn.out
```

From ease of use perspective, Visual Explain is far superior than the command line tools db2expln and dynexpln. However, the command line tool is used with script in mind. For instance, if you want to retrieve access plans of all SQL generated by a single user a script containing db2expln is a much better option than point and click for each statement.

**Visual Explain**

**Visual Explain** can be invoked via the Control Center or via the command db2vexp. **Visual Explain** returns a graphical representation of the access plan. It also gives you the cost at each node of the access plan. To generate the information for Visual explain, make sure you enable the EXPLAIN SNAPSHOT to YES to capture the SQL internal representation.

```
SET CURRENT EXPLAIN SNAPSHOT YES
```

Or,

```
EXPLAIN ALL WITH SNAPSHOT FOR " select * from syscat.tables"
```

If all these parameter settings look confusing, just access **Visual Explain** through the Control Center; parameters will be set automatically.
After the *query rewrite* phase, the access plan may bear little resemblance to the original query. *Visual Explain* shows the original SQL and the rewritten SQL. The cost of each operation in the access graph is in *timerons*. It is used to indicate the relative cost of each operation. You can then decide if there are ways to either improve the cost of the operation or remove a particular operation. For example, you could add an index to improve the search, or return the result set in the order of an index to eliminate a sort from the access plan.

**Database Snapshot Monitor**

The Snapshot monitor provides the ability to look into the current status of the database. One can capture snapshots for the Database Manager, database, application, bufferpools, tablespaces, table, and lock. The monitor can be invoked from the *Control Center*, 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 of what is going on. For example, `db2 get snapshot for locks on <database>` tells you who is holding what locks, and who are currently blocked by a deadlock. `db2 get snapshot for database on <database>` gives you a list of indicators on how well the system is doing. `db2 get snapshot for applications on <dbname>` gives the detailed status of each connection to the database. Snapshots for the bufferpool and tablespaces are particularly useful in analyzing the I/O hit ratio.

Make sure that the snapshot counters are cleared before you start to investigate problems in a given period. To reset counters for the monitor, issue the command below. It clears all the counters and snapshots return values after the last reset command.

`db2 reset monitor all.`

The amount of data gathered by the snapshot monitors 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 command *update monitor switches*. The switches are sorts, locks, tables, bufferpools, unit of work and SQL statements.

To change the setting either through the Control Center or with 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 is about 10 percent. The common practice is to turn on one switch at a time as you narrow down the problem. The simplest format for snapshot monitor with all switches turned off is

$db2 GET SNAPSHOT FOR
DATABASE|APPLICATIONS|BUFFERPOOLS|TABLESPACE|TABLES|LOCKS] ON <DBNAME>

One of the commonly requested features is to report the slowest performing SQL statements. DB2 snapshot monitor provides this ability through the DYNAMIC SQL option. When the command is issued, DB2 UDB writes all the SQL statements in the dynamic statement cache to a predefined file on the server. After that, you can use the table function SQLCHCHE_SNAPSHOT to determine what the problem SQL statements are. Reset monitor will not clear the APM content, only the counters in APM. Consult the System Monitor Guide and reference for more details.

% the following command writes the content of APM to a file
 db2 get snapshot for dynamic sql on database sample write to file
% the following command calls the UDF sysfun.SQLCACHE_SNAPSHOT.
 Table_name is an arbitrary valid SQL identifier
  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
Event Monitor

One disadvantage of the snapshot monitor is that it does not capture events that happened before the snapshot was issued. For example, `db2 get snapshot for database` shows you that you had three deadlocks since the last monitor reset but it shows no more information on the deadlocks unless you issued the snapshot request in the middle of the deadlock.

Use an event monitor to capture information on deadlock if you would like to know which applications and which tables are involved. You should have an event monitor for deadlocks to capture all the detailed information when it happened. The commands are shown here for completeness. It is much easier to set up the event from the Control Center.

To create an event for deadlock called evmon1, use the command:

```sql
db2 CREATE EVENT MONITOR evmon1 FOR DEADLOCKS WRITE TO FILE '/EVENTMONITORS/DEADLOCK/EVMON1' MAXFILES 3 MAXFILESIZE 1000
```

To turn on the event monitor, type:

```sql
SET EVENT MONITOR evmon1 STATE = 1
```

To turn off the event monitor, type:

```sql
SET EVENT MONITOR evmon1 STATE = 0
```

To view the output, use the formatter db2evmon which reads the binary records and displays them on the screen. It is best to redirect the output to a file for future analysis:

```sql
db2evmon -path /eventmonitors/deadlock/evmon1 or
db2evmon -db db2cert -evm evmon1
```
### Error Conditions

DB2 UDB stores important messages and error logs in one of these places: `db2diag.log`, `alerts` and the `message log`. You can access the `message log` and the `alerts` from the Command Center-Journal window. `db2diag.log` and the `trace` facility cannot be accessed via the Command Center.

#### DB2DIAG.LOG

`db2diag.log` is under `sqlib/db2dump` on UNIX, and under `sqlib/db2` on Windows platforms. It contains messages and trace information from the 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 the information logged in the `db2diag.log` by changing the database parameter DIAGLEVEL. Although it is not the prettiest file around, it is definitely the most comprehensive resource when you are trying to debug a non-trivial problem.

The `Journal` has mostly replaced the need to read the `db2diag.log` file, because problems are typically logged in much more readable format when they are accessed from the Command Center.

#### Trace facility

DB2 trace (`db2trc`) is used to diagnose an isolated problem. Once you have isolated an error scenario, you can turn it on to 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 codes from each function. One of the most useful kinds of information you can get quickly is the SQLCODE number in the file to find out at what stage the scenario failed. Our experience has been that this information can lead you quickly to your solution.

Because it has a heavy overhead, `db2trc` is not suitable for tracing timing problems. There are better tools for that work, such as the snapshot monitor and 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, time order and 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 trace output sorted by process ID

db2trc flw tracefile trace.flw             % sort the formatted output in time order
```

**Performance Tuning**

The question most frequently asked of DB2 support staff is how to make DB2 UDB run faster. Benchmark results have demonstrated that DB2 UDB outperforms Oracle in many workloads, but to tap in all the potential of in DB2 UDB, a certain amount of tuning is required.

Before you start poking at all these nice 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, and some other profiling tools on UNIX, and the performance monitor on NT). The information gathered from operating system monitoring tools also gives 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? Or it is taking too much I/O time while the system sits idle (I/O wait time)? Is there only one of the many processors busy while others sit idle? The following four cases demonstrate how the DB2 UDB 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 the CPU is working overtime. The Snapshot Monitor for statements also verifies it is indeed the statement that is taking up all the CPU cycles. With this information, the next step is to use the Explain or Visual Explain to determine if you have a bad access plan. Suppose you verified the optimizer is picking a poor plan. You may want to do a reorgcheck on the tables accessed to make sure the statistics are up-to-date.
Case 2

Again, the database is taking a long time to process a query. The operation system monitor tools indicate that the system is really not busy at all. 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, with concurrent applications are competing for the locks.

Case 3

Suppose, for the same symptom as above, 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 few reasons: bad physical layout of the database, a bufferpool that is too small for the tablespace, 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 tablespaces to find out if your I/O hit ratio is too low. To find out if you have a bad database physical layout, compare the access plan to your physical layout. You may want to limit the amount of I/O parallelism due to the placements 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 that SMP parallelism is not turned on. Use set database configuration to turn on SMP parallelism, and increase the application heap and ASLHEAP 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.
Appendix A - DATE data type

**ORACLE table HOLIDAY**

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

```sql
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**

```sql
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.

```sql
select Holiday, ActualDate, CelebratedDate
from Holiday
where CelebratedDate - ActualDate != 0
```

```sql
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>
Appendix B - Oracle Identifiers

As referenced in the ‘Mapping of Data Types / DDL Generation’ section, a sample sqlplus script is included below to help gather information from the Oracle Data Dictionary and identify all Oracle objects exceeding the 18 character limit imposed by DB2.

```
rem --------------------------------------------------------------
rem --------------------------------------------------------------
rem - Aleem Rajpar - 10/29/97
rem - Invoke via "sqlplus uid/pwd @filename"
set echo off
set linesize 256
set pagesize 1024
show user
rem --------------------------------------------------------------
--------------
prompt
prompt Data Dictionary!
rem desc dict;
rem select table_name, substr(comments,1,64) from dict;
prompt
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
prompt USER / SCHEMA!
rem get user/schema info
select * from all_users;
prompt
prompt VIEWS!
```
rem get (total #) AND (# objects with names > 18 chars) group by schema
rem + list non-system views
select count(*), owner from all_views group by owner;
select count(*), owner from all_views
  where owner not in ('SYS','SYSTEM')
  and length(view_name) > 18
  group by owner;
select owner, view_name
  from all_views
  where owner not in ('SYS','SYSTEM')
  order by owner, view_name;

prompt
prompt SYNONYMS!
rem get (total #) AND (# objects with names > 18 chars) group by schema
rem + list non-system synonyms
select count(*), owner from all_synonyms group by owner;
select count(*), owner from all_synonyms
  where owner not in ('SYS','SYSTEM','PUBLIC')
  and length(synonym_name) > 18
  group by owner;
select owner, synonym_name, table_owner, table_name
  from all_synonyms
  where owner not in ('SYS','SYSTEM','PUBLIC')
  order by owner, synonym_name;

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

prompt
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
prompt SEQUENCES!
rem get (total #) group by schema
select count(*), sequence_owner from all_sequences
  group by sequence_owner;
Appendix C - Simulating SEQUENCE Objects

DDL Statements for Create Trigger / Create Function

```sql
create table t1 (c1 INTEGER, c2 VARCHAR(10) );

CREATE UNIQUE INDEX seq_index ON t1 (c1 DESC);

CREATE FUNCTION seqno(INTEGER) RETURNS Integer
EXTERNAL NAME 'seqno!seqno'
SPECIFIC seqno
VARIANT
NO SQL
NO EXTERNAL ACTION
LANGUAGE C
NOT FENCED
PARAMETER STYLE DB2SQL
SCRATCHPAD
FINAL CALL;

create trigger sequencing no cascade before insert on t1
referencing new as n for each row mode db2sql
set n.c1= (select seqno(max(c1)) from t1);
```

Code for User-Defined Function

```c
#include <stdio.h>
#include <stdlib.h>
#include "sqludf.h"

#ifdef __cplusplus
extern "C"
#endif

void SQL_API_FN seqno2 (  
    long  *firstNum,      /* last sequence number used */
    long  *initNum,       /* initial sequence value */
    long  *returnValue,   /* return value, an integer */
    short *inputNull,     /* firstNum input NULL indicator*/
    short *input2Null,    /* initNum input NULL indicator */
    short *returnNull,    /* return indicator variable */
    char  *sqlstate,      /* returned SQLSTATE, char 6 */
    char  *fnName,        /* family name of fn, char 28 */
    char  *specificName,  /* specific fn name, char 19 */
    char  *message,       /* message area, char 70 */
    struct sqludf_scratchpad *scratchpad,  /* in sqludf.h */
    long  *call           /* first & last call indicator */
) {
    long *p;
    p = (long *) (scratchpad->data);  /* point at pad data */
    if (*call == -1) {
        if ( *inputNull == -1 ) {
            if ( *input2Null == -1 ) {
                *p = 0;
            } else {
                *p = *initNum - 1;
            }
        } else {
            *p = *firstNum;
        }
    }
}
```
*p = (*p)+1;            /* increment counter inside pad */
*returnValue = *p;      /* return the counter value */
*returnNull = 0;
return;
Appendix D - ROWID Column /Generate_unique

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

The UDB GENERATE_UNIQUE function returns a bit data character string that is unique compared to any other execution of the same function. The result of the function is a value that includes the internal form of the Universal Time and the partition number where the function was processed. The value includes the partition number where the function executed so that a table partitioned across multiple partitions also has unique values in some sequence. The sequence is based on the time the function was executed.

If the application design and database permit, a UDB primary key can be used to replace the Oracle ROWID. Analysis and testing should be done to verify that the primary key solution will be valid across tables and the database.

Oracle Sample Code:

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  
-------------------------------------  ---------     ---------------
AAAAfDAAEAAAAOvAAA   000001             Smith
AAAAfDAAEAAAAOvAAB   000002             Jones

SELECT LASTNAME FROM EMP_UPDATE WHERE ROWID = 'AAAAfDAAEAAAAOvAAA';
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');
INSERT INTO EMP_UPDATE (EMPNO, LASTNAME)
VALUES ('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' 000001</td>
<td>Smith</td>
<td></td>
</tr>
<tr>
<td>x'199901114195748124402000000' 000002</td>
<td>Jones</td>
<td></td>
</tr>
</tbody>
</table>

2 record(s) selected.

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

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

1 record(s) selected.

Following is sample code that demonstrates the results of a UDB primary key used in an SQL statement.

```
SELECT LASTNAME
FROM EMP_UPDATE
WHERE EMPNO = '000001'
```

LASTNAME
----------
Smith
Appendix E - A DB2 Java Stored Procedure Sample Program

This sample is taken from the DB2 Java sample directory. This sample illustrates how a java stored procedure is invoked to retrieve a multiple rows result set.

**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.

java.lang.StringPath path;
java.lang.StringPath path2;
java.lang.StringPath path3;

// 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);
}
catch (Exception e)
{
    throw e;
}
}

Client MRSPcli.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]");
    }

    // 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)
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 procedure");
    throw e;
}

// create the 3 sql statements to be executed in the Java Stored Procedure
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.executeUpdate();
int rsCount = 0;

while( true )
{	int rowCount = callStmt.getUpdateCount();
	if( rowCount > 0 )
	{
		System.out.println("=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-

Rows changed = " + rowCount);

callStmt.getMoreResults();
System.out.println();
continue;
	}

	if( rowCount == 0 )
	{
		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("="*80);
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 F - A DB2 Static SQL Sample Program

<table>
<thead>
<tr>
<th>FileName</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>prep.scr</td>
<td>Script file to pre compile all .sqc files</td>
</tr>
<tr>
<td>bind.scr</td>
<td>Script file to bind all .sqc files</td>
</tr>
<tr>
<td>makefile</td>
<td>C makefile used to compile .c files</td>
</tr>
<tr>
<td>comval.h</td>
<td>C header file that contains common values</td>
</tr>
<tr>
<td>database.h</td>
<td>C header file that contains function prototypes</td>
</tr>
<tr>
<td>dbapp.sqc</td>
<td>C main program for game application</td>
</tr>
<tr>
<td>delete.sqc</td>
<td>C function that performs a database delete</td>
</tr>
<tr>
<td>insert.sqc</td>
<td>C function that performs a database insert</td>
</tr>
<tr>
<td>makefile</td>
<td>C makefile used to compile .c files</td>
</tr>
<tr>
<td>util.sqc</td>
<td>C utility functions</td>
</tr>
</tbody>
</table>

Change INST10 to your own instance id.

prep.scr

# Build a C program containing embedded SQL
# Usage: prep.scr <db_name> <userid> <password>
# Connect to a database.
db2 connect to $1 user $2 using $3
# Pre-compile the program.

db2 prep dbapp.sqc OPTLEVEL 0 bindfile
db2 prep util.sqc OPTLEVEL 0 bindfile
db2 prep insert.sqc OPTLEVEL 0 bindfile
db2 prep selectr.sqc OPTLEVEL 0 bindfile
db2 prep selecta.sqc OPTLEVEL 0 bindfile
db2 prep update.sqc OPTLEVEL 0 bindfile
db2 prep delete.sqc OPTLEVEL 0 bindfile

# Disconnect from the database.
db2 connect reset

bind.scr
# bind script file
# Build a C program containing embedded SQL
# Usage: bind <db_name> <userid> <password>
# Connect to a database.
db2 connect to $1 user $2 using $3
# Bind the program to the database.
  db2 bind dbapp.bnd
  db2 bind util.bnd
  db2 bind insert.bnd
  db2 bind selectr.bnd
  db2 bind selecta.bnd
  db2 bind update.bnd
  db2 bind delete.bnd

# Disconnect from the database.
db2 connect reset

Makefile

#makefile for game
#Debra Eaton
# May 15, 1997

CCFLAGS = -I/usr/lpp/db2_05_00/include

game: util.o dbapp.o insert.o selectr.o selecta.o update.o delete.o
  xlc -o game dbapp.o util.o insert.o selectr.o selecta.o update.o delete.o -ldb2 -L/usr/lpp/db2_05_00/lib

util.o: util.c comval.h database.h
  xlc $(CCFLAGS) -c util.c

dbapp.o: dbapp.c comval.h database.h
  xlc $(CCFLAGS) -c dbapp.c

insert.o: insert.c comval.h database.h
  xlc $(CCFLAGS) -c insert.c

selectr.o: selectr.c comval.h database.h
  xlc $(CCFLAGS) -c selectr.c

selecta.o: selecta.c comval.h database.h
  xlc $(CCFLAGS) -c selecta.c

update.o: update.c comval.h database.h
  xlc $(CCFLAGS) -c update.c

delete.o: delete.c comval.h database.h
  xlc $(CCFLAGS) -c delete.c
**comval.h**

```c
/* comval.h */
#define SUCCESS 0
#define FAILURE -1
#define TRUE 1
#define FALSE 0
```

**database.h**

```c
void ProcessError(struct sqlca * sqlca);
long ConnectDb();
long Commit();
long Rollback();
long DisconnectDb();
long InsertRow();
long SelectRow();
long SelectAll();
long UpdateRow();
long DeleteRow();
void Menu();
```

**dbapp.sqc**

```c
/* dbapp.sqc */
/* Debra Eaton */
/* May 15, 1997 */

/* Header files */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "database.h"
#include "comval.h"
EXEC SQL INCLUDE SQLCA;
void main()
{
  /*Declare Variables */
  long status = 1;
  char command;

  /* Connect to the Database */
  printf("Welcome to the DB game.
" );

  printf("Let's connect to the database.
\n\n");
  if (SUCCESS == ConnectDb())
  {
    printf("Database connection is complete.\n");
  }
  else
  {
    printf("Sorry, you made mistake.\n");
    printf("Start the application again.\n");
    exit(EXIT_FAILURE);
  }
```
printf("Here are the Options for the game.
");
printf("NOTE: Assume deptno is a column with unique values.
\n\n");
printf("Enter i to insert a row.\n");
printf("Enter r to select a row.\n");
printf("Enter a to select all rows.\n");
printf("Enter u to update a row.\n");
printf("Enter d to delete a row.\n");
printf("Enter x to stop the game.\n");

while(status == TRUE)
{
    command = 0;
    fflush(stdin);
    printf("Command: ");
    scanf("%c", &command);
    switch(command)
    {
    case 'i':
        if (SUCCESS == InsertRow() )
        {
            printf("You successfully added a row.\n");
            printf("Try another option.\n\n");
            Menu();
        }
        else
        {
            printf("Sorry, you made a mistake.\n");
            printf("Try another option.\n\n");
            Menu();
        }
        break;
    case 'r':
        if (SUCCESS == SelectRow() )
        {
            printf("You successfully selected a row.\n");
            printf("Try another option.\n\n");
            Menu();
        }
        else
        {
            printf("Sorry, you made a mistake.\n");
            printf("Try another option.\n\n");
            Menu();
        }
        break;
    case 'a':
        if (SUCCESS == SelectAll() )
        {
            printf("You successfully selected all rows.\n");
            printf("Try another option.\n\n");
            Menu();
        }
        else
        {
            printf("Sorry, you made a mistake.\n");
            printf("Try another option.\n\n");
            Menu();
        }
        break;
    case 'u':
        if (SUCCESS == UpdateRow() )
        {
            printf("You successfully updated a row.\n");
            printf("Try another option.\n\n");
            Menu();
        }
        break;
    }
}
else
    {  
    printf("Sorry, you made a mistake.\n");
    printf("Try another option.\n\n");
    Menu();  }
break;

case 'd':
    if (SUCCESS == DeleteRow() )
    {  
    printf("You successfully deleted a row.\n");
    printf("Try another option.\n\n");
    Menu();  }
else
    {  
    printf("Sorry, you made a mistake.\n");
    printf("Try another option.\n\n");
    Menu();  }
break;

case 'x':
    printf("Disconnecting from the database.\n");
    if (SUCCESS == DisconnectDb() )
    {  
    printf("Database disconnect is complete.\n");
    printf("Exiting the application...\n");  }
else
    {  
    printf("Exiting the application with an error.\n");
    exit(EXIT_FAILURE);  }
    status = FALSE;
    break;

default:
    printf("Unknown menu option.\n");
    printf("Try again.\n");
    break;
}
}

util.sqc

/* util.sqc */
/* Debra Eaton */
/* May 15, 1997 */

/* Header files */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "comval.h"
#include <sqlenv.h>
#include "database.h"
EXEC SQL INCLUDE SQLCA;

long ConnectDb()
{
    EXEC SQL BEGIN DECLARE SECTION;
    char userid[7];
char password[7];
char dbname[7];
EXEC SQL END DECLARE SECTION;
EXEC SQL WHENEVER SQLWARNING CONTINUE;

printf("May I please have your... \n");
printf("USERID: ");
scanf("%s", &userid);
printf("Password: ");
scanf("%s", &password);
printf("Database Name: ");
scanf("%s", &dbname);

EXEC SQL CONNECT TO :dbname USER :userid USING :password;
if (SQLCODE == SUCCESS)
{ return(SUCCESS); }
else
{ ProcessError(&sqlca);
  return(FAILURE); }
}

void ProcessError(struct sqlca * sqlca)
{
  char buffer[512];
  short buff_size = sizeof(buffer);
  short line_len = 50;

  sqlaintp(buffer, buff_size, line_len, sqlca);
  printf(buffer);
}

long Commit()
{
  EXEC SQL WHENEVER SQLWARNING CONTINUE;
  EXEC SQL COMMIT;
  if (SQLCODE == SUCCESS)
  { return(SUCCESS); }
  else
  { ProcessError(&sqlca);
    return(FAILURE); }
}

long Rollback()
{
  EXEC SQL ROLLBACK;
  return(SQLCODE);
}

long DisconnectDb()
{
  EXEC SQL WHENEVER SQLWARNING CONTINUE;
  EXEC SQL CONNECT RESET;
  if (SQLCODE == SUCCESS)
  { return(SUCCESS); }
}
else
  {  ProcessError(&sqlca);
      return(FAILURE);  }
}
Appendix G - Sample DB2 Dynamic Embedded Application

As referenced in the "Host Language Applications" section, a sample DB2 Embedded SQL application is included below. The 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 the example below.

```
/*
  10/18/97 - created dynamic4.sqc for embedded dynamic sql method 4 app 
    (varying-list select)
  - example only allows varying # of INTEGER columns in select-list
*/

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

#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;
  EXEC SQL INCLUDE SQLCA;
  CHECKERR("connect");
  method4();
  EXEC SQL CONNECT RESET;
  CHECKERR("connect reset");

EXEC SQL INCLUDE SQLDA;
EXEC SQL INCLUDE SQLTC;
EXEC SQL INCLUDE SQLCA;
EXEC SQL INCLUDE SQLDA;

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 */
  /* sqcode 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/sqind 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);
  } else
    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 */

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 	", *(int *)sqldaptr->sqlvar[i].sqldata);
    }
    printf(" \n");
    rownum++;
  }
} /* do while SQLCODE != 100 */
while (SQLCODE != 100);

EXEC SQL CLOSE c4;
CHECKERR("close cursor c4");

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 ("%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 H - CLI Stored Procedure

These two examples convert the samples inpsrv and outsrv from the sqllib/sample directory using the DB2SQL parameter passing style. They also illustrate a common CLI SP framework on how to pass the environment and connection handles.

Procedure creating commands:

DROP PROCEDURE INPSRV2;

CREATE PROCEDURE INPSRV2(IN TAB_NAME CHAR(9),
                          IN PRESIDENT1 CHAR(10),
                          IN PRESIDENT2 CHAR(10),
                          IN PRESIDENT3 CHAR(10))
    EXTERNAL NAME 'inpsrv2!inpsrv2'
    LANGUAGE C
    PARAMETER STYLE DB2SQL
    PROGRAM TYPE SUB;

DROP PROCEDURE OUTSRV2;

CREATE PROCEDURE OUTSRV2(OUT MED_SAL DOUBLE)
    EXTERNAL NAME 'outsrv2!outsrv2'
    LANGUAGE C
    PARAMETER STYLE DB2SQL
    PROGRAM TYPE SUB;

dll export files:

  outsrv2_new.def

LIBRARY OUTSRV2_new
EXPORTS outsrv2_new

  inpsrv2_new.def

LIBRARY INPSRV2_new
EXPORTS inpsrv2_new

Inpcli2_new.c

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlda.h>
#include <sqlcli1.h>
#include "samputil.h"          /* Header file for CLI sample code */
/* For the Macintosh environment when generating 68K applications */
#if define DB268K
  /* Need to include ASLM for 68K applications */
#include <LibraryManager.h>
#endif

/*
Global Variables for user id and password.
To keep samples simple, not a recommended practice.
*/
extern SQLCHAR server[SQL_MAX_DSN_LENGTH + 1];
extern SQLCHAR uid[MAX_UID_LENGTH + 1];
extern SQLCHAR pwd[MAX_PWD_LENGTH + 1];

/* main */
int main( int argc, char * argv[] ) {
    SQLHANDLE henv, hdbc, hstmt;
    SQLRETURN rc;

    /*---> SQLX142.SCRIPT */
    SQLCHAR * stmt = ( SQLCHAR * ) "CALL rnewman.inpsrv2(?, ?, ?, ?)" ;
    /<*-- */

    SQLCHAR * Tab_Name = ( SQLCHAR * ) "PRESIDENT" ;
    int Tab_Name_Length = 9;

    SQLCHAR * Pres_Name[] = {
        ( SQLCHAR * ) "Washington",
        ( SQLCHAR * ) "Jefferson",
        ( SQLCHAR * ) "Lincoln"
    } ;
    int Pres_Name_Length = 10;

    struct sqlca sqlca;

    /* For the Macintosh environment when generating 68K applications */
    #ifdef DB268K
    /*
    Before making any API calls for 68K environment,
    need to initialize the Library Manager
    */
    InitLibraryManager(0,kCurrentZone,kNormalMemory);
    atexit(CleanupLibraryManager);
    #endif

    /* macro to initialize server, uid and pwd */
    INIT_UID_PWD ;

    /* allocate an environment handle */
    rc = SQLAllocHandle( SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv ) ;
    if ( rc != SQL_SUCCESS ) return( terminate( henv, rc ) ) ;

    /* allocate a connect handle, and connect */
rc = DBconnect( henv, &hdbc )
if ( rc != SQL_SUCCESS ) return( terminate( henv, rc ) ) ;
rc = SQLAllocHandle( SQL_HANDLE_STMT, hdbc, &hstmt ) ;
CHECK_HANDLE( SQL_HANDLE_DBC, hdbc, rc ) ;

/*---> */
rc = SQLPrepare( hstmt, stmt, SQL_NTS ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;
rc = SQLBindParameter( hstmt,
  1,
  SQL_PARAM_INPUT,
  SQL_C_CHAR,
  SQL_CHAR,
  9,
  0,
  Tab_Name,
  10,
  &Tab_Name_Length
  ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;
rc = SQLBindParameter( hstmt,
  2,
  SQL_PARAM_INPUT,
  SQL_C_CHAR,
  SQL_CHAR,
  10,
  0,
  Pres_Name[0],
  11,
  &Pres_Name_Length
  ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;
rc = SQLBindParameter( hstmt,
  3,
  SQL_PARAM_INPUT,
  SQL_C_CHAR,
  SQL_CHAR,
  10,
  0,
  Pres_Name[1],
  11,
  &Pres_Name_Length
  ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;
rc = SQLBindParameter( hstmt,
  4,
SQL_CHAR,
10,
0,
Pres_Name[2],
11,
&Pres_Name_Length
);
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

rc = SQLExecute( hstmt ) ;
/* Ignore Warnings */
if ( rc != SQL_SUCCESS_WITH_INFO )
  CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;
/* <-- */

rc = SQLFreeHandle( SQL_HANDLE_STMT, hstmt ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

/* COMMIT, free resources and exit */
rc = SQLEndTran( SQL_HANDLE_DBC, hdbc, SQL_COMMIT ) ;
CHECK_HANDLE( SQL_HANDLE_DBC, hdbc, rc ) ;

printf( "n>Disconnecting ....\n" ) ;
rc = SQLDisconnect( hdbc ) ;
CHECK_HANDLE( SQL_HANDLE_DBC, hdbc, rc ) ;

rc = SQLFreeHandle( SQL_HANDLE_DBC, hdbc ) ;
CHECK_HANDLE( SQL_HANDLE_DBC, hdbc, rc ) ;

rc = SQLFreeHandle( SQL_HANDLE_ENV,  henv ) ;
if ( rc != SQL_SUCCESS ) return( terminate( henv, rc ) ) ;

return( SQL_SUCCESS ) ;
}                      /* end main */
inpsrv2_new.c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlda.h>
#include <sqlcli1.h>
#include "sql.h"
#include "samputil.h"        /* Header file for CLI sample code */

int SQL_API_FN inpsrv2_new( SQLCHAR           table_name[10],
    SQLCHAR           president1[11],
    SQLCHAR           president2[11],
    SQLCHAR           president3[11],
    short           * nullinds,
    SQLCHAR           sqlst[6],
    SQLCHAR           qualname[28],...
/* Declare a local SQLCA */
struct sqlca sqlca;

SQLCHAR table_stmt[80] = "CREATE TABLE ";
SQLCHAR insert_stmt[80] = "INSERT INTO ";
SQLCHAR insert_data[21];
SQLINTEGER insert_data_ind;

/* Declare Miscellaneous Variables */
int cntr;
short table_name_length;
SQLCHAR * data_item[3];
short data_item_length[3];

/* Must initialize this value since we do not have an SQLD field */
/* like in the SQLDA. */
int num_of_data = 4;
int num_of_presidents;

/* Declare CLI Variables */
SQLHANDLE henv, hdbc, hstmt;
SQLRETURN rc;

/* We don't have to access the SQLDA because all of the arguments */
/* bound to the call are in the parameter list of the function. */
/* The order of arguments in the parameter list should be the same */
/* as in the CREATE PROCEDURE statement for this procedure. */

/* The length/null indicator of each parameter is in the nullinds */
/* array in the parameter list. The order is the same as the */
/* order of the parameter list. */
/* Since the parameters are CHAR, they have a constant length, so */
/* nullinds holds the NULL indicator for each. We can specify the */
/* the lengths ourselves. */
table_name_length = 9;

num_of_presidents = num_of_data - 1;
data_item[0] = president1;
data_item_length[0] = 10;
data_item[1] = president2;
data_item_length[1] = 10;
data_item[2] = president3;
data_item_length[2] = 10;
/* Setup CLI required environment */
/*================================================================*/

SQLAllocHandle( SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv ) ;
SQLAllocHandle( SQL_HANDLE_DBC, henv, &hdbc ) ;

/*================================================================*/
/* Issue NULL Connect, since in CLI we need a statement handle */
/* and thus a connection handle and environment handle. */
/* A connection is not established, rather the current */
/* connection from the calling application is used */
/*================================================================*/
SQLConnect( hdbc, NULL, SQL_NTS, NULL, SQL_NTS, NULL, SQL_NTS ) ;

/*================================================================*/
/* Be sure to include this line!!! The server does not pick the */
/* attribute up from the client. */
/*================================================================*/
SQLSetConnectAttr(hdbc, SQL_ATTR_AUTOCOMMIT, SQL_AUTOCOMMIT_OFF, SQL_NTS);

SQLAllocHandle( SQL_HANDLE_STMT, hdbc, &hstmt ) ;

/*================================================================*/
/* Create President Table */
/* - For simplicity, we'll ignore any errors from the */
/* CREATE TABLE so that you can run this program even when the */
/* table already exists due to a previous run. */
/*================================================================*/

strncat( ( char * ) table_stmt,
         ( char * ) table_name,
         table_name_length
       ) ;
strcat( ( char * ) table_stmt, " (name CHAR(20))" ) ;
SQLExecDirect( hstmt, table_stmt, SQL_NTS ) ;
SQLFreeStmt( hstmt, SQL_RESET_PARAMS ) ;

/*================================================================*/
/* Generate and execute a PREPARE for an INSERT statement, and */
/* then insert the three presidents. */
/*================================================================*/

strncat( ( char * ) insert_stmt,
         ( char * ) table_name,
         table_name_length
       ) ;
strcat( ( char * ) insert_stmt, " VALUES (?)" ) ;

if ( SQLPrepare(hstmt, insert_stmt, SQL_NTS) != SQL_SUCCESS ) goto ext ;

/* Bind insert_data to parameter marker */
```c
SQLBindParameter( hstmt,
    1,
    SQL_PARAM_INPUT,
    SQL_C_CHAR,
    SQL_CHAR,
    20,
    0,
    insert_data,
    21,
    &insert_data_ind ) ;

for ( cntr = 0; cntr < num_of_presidents; cntr++ ) {
    strncpy( ( char * ) insert_data,
        ( char * ) data_item[cntr],
        data_item_length[cntr]) ;
    insert_data_ind = data_item_length[cntr] ;
    if ( SQLExecute( hstmt ) != SQL_SUCCESS ) goto ext ;
}

outcli2_new.c:
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlcli1.h>
#include "samputil.h"          /* Header file for CLI sample code */
/* For the Macintosh environment when generating 68K applications */
#define DB268K
    /* Need to include ASLM for 68K applications */
#include <LibraryManager.h>
#endif
/*
Global Variables for user id and password.
To keep samples simple, not a recommended practice.
*/
extern SQLCHAR server[SQL_MAX_DSN_LENGTH + 1] ;
extern SQLCHAR uid[MAX_UID_LENGTH + 1] ;
extern SQLCHAR pwd[MAX_PWD_LENGTH + 1] ;

/* main */
int main( int argc, char * argv[] ) {
    SQLHANDLE henv, hdbc, hstmt ;
    SQLRETURN rc ;
    /*---> SOLL1X45.SCRIPT */
    SQLCHAR * stmt = ( SQLCHAR * ) "CALL rnewman.outsrv2( ? )" ;
```
/* Declare Local Variables for Holding Returned Data */
SQLDOUBLE sal = 0.00; /* Bound to parameter marker in stmt */
SQLINTEGER salind = 0; /* Indicator variable for sal */

/* SPECIAL NOTE: */
/* If you use the SQL_NULL_DATA for the indicator, the Stored Procedure will not return a value for the variable. */
/* If you uncomment the following line and comment the above line, sal will keep it's original value. */
/* SQLINTEGER salind = SQL_NULL_DATA; */
/* Indicator variable for sal */

/* For the Macintosh environment when generating 68K applications */
#ifdef DB268K
/* Before making any API calls for 68K environment, need to initialize the Library Manager */
InitLibraryManager(0, kCurrentZone, kNormalMemory);
atexit(CleanupLibraryManager);
#endif

/* macro to initialize server, uid and pwd */
INIT_UID_PWD;

/* allocate an environment handle */
rc = SQLAllocHandle( SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv ) ;
if ( rc != SQL_SUCCESS ) return( terminate( henv, rc ) ) ;

/* allocate a connect handle, and connect */
rc = DBconnect( henv, &hdbc ) ;
if ( rc != SQL_SUCCESS ) return( terminate( henv, rc ) ) ;

rc = SQLAllocHandle( SQL_HANDLE_STMT, hdbc, &hstmt ) ;
CHECK_HANDLE( SQL_HANDLE_DBC, hdbc, rc ) ;

/* Call the Remote Procedure via CALL with bound parameters */
printf("Use CALL with Host Variable to invoke the Server Procedure named outsrv\n");

/* Prepare the call statement */
rc = SQLPrepare(hstmt, stmt, SQL_NTS);
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

/* Bind the parameter to application variables (sal and salind) */
rc = SQLBindParameter(hstmt, 1, SQL_PARAM_OUTPUT, SQL_C_DOUBLE, SQL_DOUBLE,
0, 0, &sal, 0, &salind/*, NULL*/); 
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

SQLExecute(hstmt); /* Ignore Warnings */
if (rc != SQL_SUCCESS & rc != SQL_SUCCESS_WITH_INFO)
    CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

printf("Server Procedure Complete.\n");

/* Print Salary Returned in the bound application variables */
if (salind == SQL_NULL_DATA) /* Check for null value */
    printf("Median Salary = NULL\n");
else
    printf("Median Salary = %.2f\n", sal );
/*<-- */

rc = SQLFreeHandle( SQL_HANDLE_STMT, hstmt ) ;
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt, rc ) ;

/* COMMIT, free resources and exit */
rc = SQLEndTran( SQL_HANDLE_DBC, hdbc, SQL_COMMIT ) ;
CHECK_HANDLE( SQL_HANDLE_DBC, hdbc, rc ) ;

printf("\n>Disconnecting .....\n") ;
rc = SQLDisconnect( hdbc ) ;
CHECK_HANDLE( SQL_HANDLE_DBC, hdbc, rc ) ;

rc = SQLFreeHandle( SQL_HANDLE_DBC, hdbc ) ;
CHECK_HANDLE( SQL_HANDLE_DBC, hdbc, rc ) ;

rc = SQLFreeHandle( SQL_HANDLE_ENV,  henv ) ;
if ( rc != SQL_SUCCESS ) return( terminate( henv, rc ) ) ;

return( SQL_SUCCESS ) ;
}                            /* end main */

outsrv2_new.c:

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlda.h>
#include <sqlcli1.h>
#include "samputil.h"          /* Header file for CLI sample code */

/*-- SQLL1X45.SCRIPT */
int SQL_API_FN outsrv2_new(SQLDOUBLE        * median_salary,
                          short           * nullinds,
                          SQLCHAR           sqlst[6],
                          SQLCHAR           qualname[28],
                          SQLCHAR           specname[19],
                          SQLCHAR           diagmsg[71])

/*<-- */
{
    /* Declare a local SQLCA */
struct sqlca sqlca;

/* Declare CLI Variables */
SQLHANDLE henv, hdbc, hstmt1, hstmt2;
SQLRETURN rc;

SQLSMALLINT num_records;
SQLINTEGER indicator;
/*--> */

SQLCHAR * stmt1 = (SQLCHAR *) "SELECT salary FROM STAFF ORDER BY salary";
SQLCHAR * stmt2 = (SQLCHAR *) "SELECT count(*) FROM STAFF";

SQLINTEGER counter = 0;

/*-----------------------------------------------------------------*/
/* Setup CLI required environment                                 */
/*-----------------------------------------------------------------*/
rc = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
if (rc != SQL_SUCCESS) return(terminate(henv, rc)) ;
rc = SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
if (rc != SQL_SUCCESS) return(terminate(henv, rc)) ;

/*-----------------------------------------------------------------*/
/* Issue NULL Connect, since in CLI we need a statement handle     */
/* and thus a connection handle and environment handle.            */
/* A connection is not established, rather the current            */
/* connection from the calling application is used                */
/*-----------------------------------------------------------------*/
SQLConnect(hdbc, NULL, SQL_NTS, NULL, SQL_NTS, NULL, SQL_NTS) ;

/*-----------------------------------------------------------------*/
/* Be sure to include this line!!! The server does not pick the    */
/* attribute up from the client.                                  */
/*-----------------------------------------------------------------*/
SQLSetConnectAttr(hdbc, SQL_ATTR_AUTOCOMMIT, SQL_AUTOCOMMIT_OFF, SQL_NTS);

SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt1);
SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt2);

/* Execute a Statement to Obtain and Order all Salaries */
rc = SQLExecDirect(hstmt1, stmt1, SQL_NTS);
if (rc != SQL_SUCCESS) goto ext;

/* Execute a Statement to */
/* determine the Total Number of Records */
SQLExecDirect(hstmt2, stmt2, SQL_NTS);
SQLFetch(hstmt2);
rc = SQLGetData(hstmt2, 1, SQL_C_SHORT, &num_records, 0, NULL);
/* Fetch Salaries until the Median Salary is Obtained */
while ( counter++ < num_records/2 + 1 ) {
    rc = SQLFetch(hstmt1);
    if (rc == SQL_ERROR) goto ext;
}

/* No need to use SQLDA here, just use the function argument as a */
/* parameter of the SQLGetData call.                           */
rc = SQLGetData(hstmt1, 1, SQL_C_DOUBLE, median_salary, 0, &indicator);
nullinds[0] = (short)indicator;

if (rc != SQL_SUCCESS) goto ext;

/* Return to caller                                         */
/*----------------------------------------------------------*/

ext:
if (rc != SQL_SUCCESS) printf("RC = %d\n", rc);
SQLGetSQLCA(henv, hdbc, hstmt1, &sqlca);

rc = SQLEXecSQL(henv, hstmt1, SQL_F指纹指2的, SQLڈی*=-, hstmt1, rc);
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt1, rc );

rc = SQLEXecSQL(henv, hstmt2, SQL_F纹20的, SQLڈی*=-, hstmt2, rc );
CHECK_HANDLE( SQL_HANDLE_STMT, hstmt2, rc );

rc = SQLDisconnect( hdbc );
CHECK_HANDLE( SQL_HANDLE_DBC, hdbc, rc );

rc = SQLFreeHandle( SQL_HANDLE_DBC, hdbc );
CHECK_HANDLE( SQL_HANDLE_DBC, hdbc, rc );

rc = SQLFreeHandle( SQL_HANDLE_ENV, henv );
if (rc != SQL_SUCCESS)
    return( terminate( henv, rc ) );

return(1); /* Return SQLZ_DISCONNECT_PROC */
}
Appendix I - C Stored Procedure

SAMPLE 1:

The first sample shows how an Oracle PL/SQL stored procedure is mapped to a DB2 UDB C embedded stored procedure

Oracle Stored Procedure Sample Code
Following is sample code that shows an Oracle stored procedure. The procedure will insert a row into the PRODUCT table.

```sql
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;
```

UDB Stored Procedure Sample Code
Following is sample code that shows an UDB stored procedure. The procedure will INSERT a row into the PRODUCT table.

```c
#include <stdio.h>
#include <memory.h>
#include <stdlib.h>
#include <string.h>
#include <sqlenv.h>
#include <sqlutil.h>
#include <sql.h>
#include <sqlca.h>

SQL_API_RC SQL_API_FN SPInsertRow(
  void *reserved1,
  void *reserved2,
  struct sqlda *inout_sqlda,
  struct sqlca *ca)
/* Declare local SQLCA */
EXEC SQL INCLUDE SQLCA;
/* Declare host variables */
EXEC SQL BEGIN DECLARE SECTION;
  char id1[10];
  char name1[20];
EXEC SQL END DECLARE SECTION;
```
/* Declare miscellaneous variables */
long int cntr;
long int SUCCESS = 0;

/* SLQCA Warning handling */
EXEC SQL WHENEVER SQLWARNING CONTINUE;
EXEC SQL WHENEVER SQLERROR CONTINUE;

/* The client initializes the SQLDA structure.*/
/* The SQLDA structure is sent from the client to the stored procedure.*/
/* Data is assigned from the SQLDA structure to host variables in the stored procedures.*/
strcpy( id1, inout_sqlda->sqlvar[0].sqldata);
strcpy( name1, inout_sqlda->sqlvar[1].sqldata);

/* The INSERT statement is executed.*/
EXEC SQL INSERT INTO PRODUCT VALUES (:id1, :name1);

/* Return the following to the client via the SQLDA structure:*/
/* - the SQLCA */
/* - the updated output SQLDA */
/* - No return data, set -128 */
memcpy(ca, &sqlca, sizeof(struct sqlca));
if(inout_sqlda != NULL)
{
for (cntr = 0; cntr < inout_sqlda->sqld; cntr++)
{ *(inout_sqlda->sqlvar[cntr].sqlind) = -128; }
return(SQLZ_DISCONNECT_PROC);
}

The ProcessError() function is called from the client. The stored procedure returns the value of the SQLCA code via the SQLDA structure to the client. DB2 UDB error messages are called via the sqlaintp() function.

void ProcessError(struct sqlca * sqlca)
{
    char buffer[512];
    short buff_size = sizeof(buffer);
    short line_len = 50;

    sqlaintp(buffer, buff_size, line_len, sqlca);
    printf(buffer);
}
SAMPLE 2:

This code sample shows how the sample program **game** can be implemented using a stored procedure. The bldsp script places the stored procedure **proc** in the fenced directory sqllib/functions.

<table>
<thead>
<tr>
<th>FileName</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE.SQL</td>
<td>Create the table the application will use</td>
</tr>
<tr>
<td>TEST.SQL</td>
<td>Sample INSERT statement</td>
</tr>
<tr>
<td>PREP</td>
<td>Script file to pre compile client .sqc files</td>
</tr>
<tr>
<td>BIND</td>
<td>Script file to bind client .sqc files</td>
</tr>
<tr>
<td>MAKEFILE</td>
<td>C makefile used to compile and link client .c files</td>
</tr>
<tr>
<td>BLDSP</td>
<td>Script file to pre compile, bind, compile and link the stored procedures</td>
</tr>
<tr>
<td>PROC.EXP</td>
<td>Export file that contains a list of stored procedures</td>
</tr>
<tr>
<td>PROC.SQC</td>
<td>Stored procedure library</td>
</tr>
<tr>
<td>SPAPP.SQC</td>
<td>C main program/driver for game application</td>
</tr>
<tr>
<td>COMVAL.H</td>
<td>C header file that contains common values</td>
</tr>
<tr>
<td>DATABASE.H</td>
<td>C header file that contains prototypes</td>
</tr>
<tr>
<td>SPAPP.SQC</td>
<td>C main program/driver for game application</td>
</tr>
<tr>
<td>UTIL.SQC</td>
<td>Utility functions used by the application</td>
</tr>
<tr>
<td>CDELETE.SQC</td>
<td>Client function that requests a database delete</td>
</tr>
<tr>
<td>CINSERT.SQC</td>
<td>Client function that requests a database insert</td>
</tr>
<tr>
<td>CSELECT.SQC</td>
<td>Client function that requests a count of table rows</td>
</tr>
<tr>
<td>CUPDATE.SQC</td>
<td>Client function that requests a database update</td>
</tr>
</tbody>
</table>

Use the "DB2 Building Applications for UNIX Environments", S10J-8161 manual, Chapter 4, section "Building C Stored Procedures" to build your application. In addition, please do the following steps:

1. Copy all the files to a directory with a user ID that has the correct permission and authority for creating stored procedures.
2. Create the PRODUCT table using table.sql.
3. Update the variable in each client program’s host declare section with the correct stored procedure path.
4. bldsp database userid password <enter>
5. Update prep.start with your user ID, password.
6. Update bind.start with your user ID, password.
7. prept <enter>
8. make <enter>
9. bind <enter>
10. game <enter>
**makefile:**

```
#makefile for spapp

CCFLAGS = -I/usr/lpp/db2_05_00/include

game: util.o spapp.o cinsert.o cselect.o cupdate.o cdelete.o
    xlc -o game spapp.o util.o cinsert.o cselect.o cupdate.o cdelete.o -ldb2
    -L/usr/lpp/db2_05_00/lib

util.o: util.c comval.h database.h
    xlc $(CCFLAGS) -c util.c

spapp.o: spapp.c comval.h database.h
    xlc $(CCFLAGS) -c spapp.c

cinsert.o: cinsert.c comval.h database.h
    xlc $(CCFLAGS) -c cinsert.c

cselect.o: cselect.c comval.h database.h
    xlc $(CCFLAGS) -c cselect.c

cupdate.o: cupdate.c comval.h database.h
    xlc $(CCFLAGS) -c cupdate.c

cdelete.o: cdelete.c comval.h database.h
    xlc $(CCFLAGS) -c cdelete.c
```

**bldsp:**

```
#!/bin/ksh
# Debra Eaton, IBM
# Last usage = 12/29/98

# bldsp script file
# Builds a stored procedure
# Usage: bldsp [ <db_name> [ <userid> <password> ]]

# Connect to a database.
if (($# < 2))
    then
db2 connect to sample
else (($# < 3))
    then
db2 connect to $1
else
db2 connect to $1 user $2 using $3
fi

# Pre-compile the program.
db2 prep proc.sqc bindfile

# Bind the program to the database.
db2 bind proc.bnd
```
# Disconnect from the database.
db2 connect reset

# Compile the program.
xlc -I/usr/lpp/db2_05_00/include -c proc.c

# Link the program using the export file $1.exp,  
# creating a shared library called $1 with the default  
# entry point $1.
xlc -o proc proc.o -ldb2 -L/usr/lpp/db2_05_00/lib \  
   -H512 -T512 -bE:proc.exp -e SPInsertRow

# Copy the shared library to the function sub directory of the db2 instance.  
# Note: this assumes the user has write permission to this directory.  
eval "H=~$DB2INSTANCE"
cp proc $H/sql/lib/function

### table.sql:

# table.sql  
# Create the PRODUCT table for the spapp application

CREATE TABLE product  
  (ID    CHAR(10),
   NAME   CHAR(20));

### test.sql:

#test.sql

INSERT INTO PRODUCT (ID, NAME)  
VALUES ('123','ABC');
**proc.sqc**

/* proc.sqc */
/* File that contains stored procedures for spapp */

/**************************** SPInsertRow.sqc* ****************************/
/* Sample stored procedure */
/* Last update = 12/22/98 */
/* Debra Eaton, IBM */
/************************** SPInsertRow.sqc* ****************************/
/* This stored procedure will insert rows into the */
/* the SAMPLE database PRODUCT table. */
/************************** SPInsertRow.sqc* ****************************/
/* Dependencies = cinsert.sqc */
/************************** SPInsertRow.sqc* ****************************/

/**************************** SPInsertRow.sqc* ****************************/
/* Include C library definitions */
/************************** SPInsertRow.sqc* ****************************/
#include <stdio.h>
#include <memory.h>
#include <stdlib.h>
#include <string.h>
#include <sqlenv.h>
#include <sqlutil.h>
#include <sql.h>
#include <sqlda.h>
#include <sqlca.h>

SQL_API_RC SQL_API_FN SPInsertRow(
    void *reserved1,
    void *reserved2,
    struct sqlda *inout_sqlda,
    struct sqlca *ca)
{
    /* Declare local SQLCA */
    EXEC SQL INCLUDE SQLCA;

    /* Declare host variables */
    EXEC SQL BEGIN DECLARE SECTION;
    char id1[10];
    char name1[20];
    EXEC SQL END DECLARE SECTION;

    /* Declare miscellaneous variables */
    long int cntr;
    long int SUCCESS = 0;

    /* SQLCA Warning handling */
    EXEC SQL WHENEVER SQLWARNING CONTINUE;
    EXEC SQL WHENEVER SQLERROR CONTINUE;

    /* Assign data from the SQDLA to host variables */
    strcpy( id1, inout_sqlda->sqlvar[0].sqldata);
    strcpy( name1, inout_sqlda->sqlvar[1].sqldata);

    EXEC SQL INSERT INTO PRODUCT VALUES (:id1, :name1);
SPUpdateRow.sqc

/******************** SPUpdateRow.sqc********************/
/* Sample stored procedure */
/* Last update = 12/28/98 */
/* Debra Eaton, IBM */
/***************************************************************************
/* This stored procedure will update rows in the */
/* the SAMPLE database PRODUCT table. */
/***************************************************************************
/* Dependencies = cupdate.sqc */
/***************************************************************************/

/***************************************************************************/
/* Include C library definitions */
/***************************************************************************/
#include <stdio.h>
#include <memory.h>
#include <stdlib.h>
#include <string.h>
#include <sqlenv.h>
#include <sqlutil.h>
#include <sql.h>
#include <sqlda.h>
#include <sqlca.h>

SQL_API_RC SQL_API_FN SPUpdateRow(
    void *reserved1,
    void *reserved2,
    struct sqlda *inout_sqlda,
    struct sqlca *ca)
{
    /* Declare local SQLCA */
    EXEC SQL INCLUDE SQLCA;

    /* Declare host variables */
    EXEC SQL BEGIN DECLARE SECTION;
    char id2[10];
    char name2[20];
    EXEC SQL END DECLARE SECTION;
/* Declare miscellaneous variables */
long int cntr;
long int SUCCESS = 0;

/* SQLCA Warning handling */
EXEC SQL WHENEVER SQLWARNING CONTINUE;
EXEC SQL WHENEVER SQLERROR CONTINUE;

/* Assign data from the SQLDA to host variables */
strcpy( id2, inout_sqlda->sqlvar[0].sqldata);
strcpy( name2, inout_sqlda->sqlvar[1].sqldata);

EXEC SQL UPDATE PRODUCT SET NAME = :name2 WHERE ID = :id2;

/* Return to caller: */
/* Copy the SQLCA */
/* Update the output SQLDA */
/* No return data, set -128 */
memcpy(&sqlca, sqlca, sizeof(struct sqlca));
if(inout_sqlda != NULL)
{
    for (cntr = 0; cntr < inout_sqlda->sqld; cntr++)
    {
        *(inout_sqlda->sqlvar[cntr].sqlind) = -128;
    }
}
return(SQLZ_DISCONNECT_PROC);
SPDeleteRow.sqc

/* Sample stored procedure */
/* Last update = 12/29/98 */
/* Debra Eaton, IBM */
/* This stored procedure will delete rows from the SAMPLE database PRODUCT table. */
/* Dependencies = cdelete.sqc */

/*****************************************************************************/
/* Include C library definitions */
/*****************************************************************************/
#include <stdio.h>
#include <memory.h>
#include <stdlib.h>
#include <string.h>
#include <sqlenv.h>
#include <sqlutil.h>
#include <sql.h>
#include <sqlca.h>
#include <sqlda.h>

SQL_API_RC SQL_API_FN SPDeleteRow(
    void *reserved1,
    void *reserved2,
    struct sqlda *inout_sqlda,
    struct sqlca *ca)
{
    /* Declare local SQLCA */
    EXEC SQL INCLUDE SQLCA;

    /* Declare host variables */
    EXEC SQL BEGIN DECLARE SECTION;
    char id3[10];
    char name3[20];
    EXEC SQL END DECLARE SECTION;

    /* Declare miscellaneous variables */
    long int cntr;
    long int SUCCESS = 0;

    /* SQLCA Warning handling */
    EXEC SQL WHENEVER SQLWARNING CONTINUE;
    EXEC SQL WHENEVER SQLERROR CONTINUE;

    /* Assign data from the SQLDA to host variables */
    strcpy( id3, inout_sqlda->sqlvar[0].sqldata);
    strcpy( name3, inout_sqlda->sqlvar[1].sqldata);

    EXEC SQL DELETE FROM PRODUCT WHERE ID = :id3;

    /* Return to caller: */
    /* Copy the SQLCA */
    /* Update the output SQLDA */
SPSelectCount.sqc

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sql.h>
#include <sqlda.h>
#include <sqlca.h>

SQL_API_RC SQL_API_FN SPSelectCount(
    void *reserved1,
    void *reserved2,
    struct sqlda *inout_sqlda,
    struct sqlca *ca)
{
    /* Declare local SQLCA */
    EXEC SQL INCLUDE SQLCA;

    /* Declare host variables */
    EXEC SQL BEGIN DECLARE SECTION;
    long number;
    char table[20];
    char select_stmt[80] = "SELECT COUNT(*) FROM ";
    EXEC SQL END DECLARE SECTION;

    /* Declare miscellaneous variables */
    long int cntr;
    long int SUCCESS = 0;

    /* No return data, set -128 */
    memcpy(ca, &sqlca, sizeof(struct sqlca));
    if(inout_sqlda != NULL)
    {
        for (cntr = 0; cntr < inout_sqlda->sqld; cntr++)
        {
            *(inout_sqlda->sqlvar[cntr].sqlind) = -128;
        }
    }
    return(SQLZ_DISCONNECT_PROC);
}
/* SLQCA Warning handling */
EXEC SQL WHENEVER SQLWARNING CONTINUE;
EXEC SQL WHENEVER SQLERROR CONTINUE;

/* Assign data from the SQLDA to host variables */
strcpy(table, inout_sqlda->sqlvar[0].sqldata);

/* Create the statement */
strcat(select_stmt, table);

EXEC SQL DECLARE count_em CURSOR FOR s0;
EXEC SQL PREPARE s0 FROM :select_stmt;
EXEC SQL OPEN count_em;
EXEC SQL FETCH count_em INTO :number;
EXEC SQL CLOSE count_em;

/* Return to caller: */
/* Copy the SQLCA */
/* Update the output SQLDA */
/* No return data, set -128 */
memcpy(ca, &sqlca, sizeof(struct sqlca));

*(inout_sqlda->sqlvar[0].sqlind) = -128;
*(inout_sqlda->sqlvar[1].sqlind) = 0;
*(inout_sqlda->sqlvar[1].sqldata) = number;

return(SQL2_DISCONNECTPROC);
}
comval.h

/************************* comval.h *************/
/* Common values for the stored procedure */
/* application spapp.sqc */
/* Last update = 12/17/98 */
/* Debra Eaton, IBM */
/********************************************************/

#define SUCCESS 0
#define FAILURE -1
#define TRUE 1
#define FALSE 0

database.h:

/************************* database.h *******************/
/* This module contains the function prototypes for */
/* spapp.sqc, the stored procedure application. */
/* Last update = 12/17/98 */
/* Debra Eaton, IBM */
/********************************************************/

void ProcessError(struct sqlca * sqlca);
long ConnectDb();
long Commit();
long Rollback();
long DisconnectDb();
long cinsertr();
long cselect();
long cupdate();
long cdelete();
void Menu();

util.sqc:

/************************* util.sqc***************************/
/* Utility functions for the stored procedure */
/* application spapp.sqc */
/* Last update = 12/17/98 */
/* Debra Eaton, IBM */
/********************************************************/
/* These are utility functions for the stored */
/* procedure application spapp.sqc */
/********************************************************/
/* Header files */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "comval.h"
#include "sqlenv.h"
#include "database.h"
EXEC SQL INCLUDE SQLCA;

long ConnectDb()
{
    EXEC SQL BEGIN DECLARE SECTION;
    char userid[7];
    char password[7];
    char dbname[7];
    EXEC SQL END DECLARE SECTION;

    EXEC SQL WHENEVER SQLWARNING CONTINUE;
    printf("May I please have your... 
");
    printf("USERID: ");
    scanf("%s", &userid);
    printf("Password: ");
    scanf("%s", &password);
    printf("Database Name: ");
    scanf("%s", &dbname);

    EXEC SQL CONNECT TO :dbname USER :userid USING :password;
    if (SQLCODE == SUCCESS)
    {
        return(SUCCESS);
    }
    else
    {
        ProcessError(&sqlca);
        return(FAILURE);
    }
}

void ProcessError(struct sqlca * sqlca)
{
    char buffer[512];
    short buff_size = sizeof(buffer);
    short line_len = 50;
    sqlaintp(buffer, buff_size, line_len, sqlca);
    printf(buffer);
}

long Commit()
{
    EXEC SQL WHENEVER SQLWARNING CONTINUE;
    EXEC SQL COMMIT;
    if (SQLCODE == SUCCESS)
    {
        return(SUCCESS);
    }
    else
    {
        ProcessError(&sqlca);
        return(FAILURE);
    }
}

long Rollback()
{
EXEC SQL ROLLBACK;
    return(SQLCODE);
}

long DisconnectDb()
{
    EXEC SQL WHENEVER SQLWARNING CONTINUE;
    EXEC SQL CONNECT RESET;
    if (SQLCODE == SUCCESS)
    {
        return(SUCCESS);
    }
    else
    {
        ProcessError(&sqlca);
        return(FAILURE);
    }
}

void Menu()
{
    printf("Here are the Options for the game.\n");
    printf("NOTE: Assume deptno is a column with unique values.\n");
    printf("Enter i to insert a row.\n");
    printf("Enter s to select a row count.\n");
    printf("Enter u to update a row.\n");
    printf("Enter d to delete a row.\n");
    printf("Enter x to stop the game.\n");
}
**spapp.sqc:**

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "database.h"
#include "comval.h"

EXEC SQL INCLUDE SQLCA;

void main()
{
    /*Declare Variables */
    long status = 1;
    char command;

    /* Connect to the Database */
    printf("\nWelcome to the DB game.\n");
    printf("Let's connect to the database.\n\n");
    if (SUCCESS == ConnectDb() )
    {
        printf("Database connection is complete.\n");
    } else
    {
        printf("Sorry, you made mistake.\n");
        printf("Start the application again.\n");
        exit(EXIT_FAILURE);
    }

    printf("Here are the Options for the game.\n");
    printf("NOTE: Assume deptno is a column with unique values.\n\n");
    printf("Enter i to insert a row. \n");
    printf("Enter s to select a row count. \n");
    printf("Enter u to update a row. \n");
    printf("Enter d to delete a row. \n");
    printf("Enter x to stop the game.\n");

    while (status == TRUE)
```
{  
    command = 0;
    fflush(stdin);
    printf("Command: ");
    scanf("%c", &command);

    switch(command)
    {
    case 'i':
        if (SUCCESS == cinsert())
        {
            printf("You successfully added a row.\n");
            printf("Try another option.\n\n");
            Menu();
        }
        else
        {
            printf("Sorry, you made a mistake.\n");
            printf("Try another option.\n\n");
            Menu();
        }
        break;
    case 's':
        if (SUCCESS == cselect())
        {
            printf("You successfully selected a row count.\n");
            printf("Try another option.\n\n");
            Menu();
        }
        else
        {
            printf("Sorry, you made a mistake.\n");
            printf("Try another option.\n\n");
            Menu();
        }
        break;
    case 'u':
        if (SUCCESS == cupdate())
        {
            printf("You successfully updated a row.\n");
            printf("Try another option.\n\n");
            Menu();
        }
        else
        {
            printf("Sorry, you made a mistake.\n");
            printf("Try another option.\n\n");
            Menu();
        }
        break;
    case 'd':
        if (SUCCESS == cdelete())
        {
            printf("You successfully deleted a row.\n");
            printf("Try another option.\n\n");
            Menu();
        }
else
{
    printf("\nSorry, you made a mistake.\n");
    printf("Try another option.\n\n");
    Menu();
}
break;

case 'x':
    printf("\nDisconnecting from the database.\n");
    if (SUCCESS == DisconnectDb() )
    {
        printf("Database disconnect is complete.\n");
        printf("Exiting the application...\n");
    }
    else
    {
        printf("Exiting the application with an error.\n");
        exit(EXIT_FAILURE);
    }
    status = FALSE;
    break;

default:
    printf("\nUnknown menu option.\n");
    printf("\nTry again.\n");
    break;
    break;
}
**cinsert.sqc:**

```c
/* Sample client program for stored procedure */
/* Last update = 12/18/98 */
/* Debra Eaton, IBM */

/* This client program will allocate and initialize data areas for a stored procedure. Also, the CALL statement will communicate with the stored procedure */

/* Dependencies = SPInsertRow.sqc */

/* Dependencies = SPInsertRow.sqc */

/* Include C library definitions */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sqlenv.h>
#include <sqlutil.h>
#include <sql.h>
#include <sqlda.h>
#include <sqlca.h>
#include "comval.h"
#include "database.h"

EXEC SQL INCLUDE SQLCA;

long int cinsert()
{

  /* Declare host variables */
  EXEC SQL BEGIN DECLARE SECTION;
  char procname[255] = "/home/inst10/sp/proc!SPInsertRow";
  char id[10];
  char name[20];
  short idind, nameind;
  EXEC SQL END DECLARE SECTION;

  /* Declare Variables for CALL */
  struct sqlda sqlca;
  struct sqlda *inout_sqlda = NULL;

  /* Initialize variables */
  fflush(stdin);
  printf("May I please have the ... \n");
  printf("ID: ");
  scanf("%s", &id);
  printf("NAME: ");
  scanf("%s", &name);
  printf("\n You will insert ID %s and NAME %s\n", id, name);

  /* Allocate and Initialize Input SQLDA */
  /* sqlcdabc = (44 * sqln) + 16 bytes */
  inout_sqlda = (struct sqlda *)malloc(SQLDASIZE(2));
  inout_sqlda->sqln = 2;
  inout_sqlda->sqld = 2;
```

EXEC SQL WHENEVER SQLWARNING CONTINUE;

EXEC SQL CALL :procname USING DESCRIPTOR :*inout_sqlda;
printf("Server procedure complete.\n\n");
if (SQLCODE == SUCCESS)
{
    free(inout_sqlda);
    if  (SUCCESS == Commit())
    {
        return(SUCCESS);
    }
    else
    {
        ProcessError(&sqlca);
        Rollback();
        return(FAILURE);
    }
}
else
{
    ProcessError(&sqlca);
    Rollback();
    free(inout_sqlda);
    return(FAILURE);
}
cdelete.sqc:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sqlenv.h>
#include <sqlutil.h>
#include <sql.h>
#include <sqlda.h>
#include <sqlca.h>
#include "comval.h"
#include "database.h"

EXEC SQL INCLUDE SQLCA;

long int cdelete()
{
    /* Declare host variables */
    EXEC SQL BEGIN DECLARE SECTION;
    char procnam[255] = "/home/inst10/sp/proc!SPDeleteRow"
    char id[10];
    char name[20];
    short idind, nameind;
    EXEC SQL END DECLARE SECTION;

    /* Declare Variables for CALL */
    struct sqlca sqlca;
    struct sqlda *inout_sqlda = NULL;

    /* Initialize variables */
    fflush(stdin);
    printf("May I please have the ... \n");
    printf("ID: ");
    scanf("%s", &id);
    printf("NAME: ");
    scanf("%s", &name);
    printf("\n You will delete the row with the ID %s and NAME %s\n",id,name);

    /* Allocate and Initialize Input SQLDA */
    /* sqlcdabc = (44 * sqln) + 16 bytes */
    inout_sqlda = (struct sqlda *)malloc(SQLDASIZE(2));
```
inout_sqlda->sqln = 2;
inout_sqlda->sqld = 2;
inout_sqlda->sqldabc = 104;

inout_sqlda->sqlvar[0].sqltype = SQL_TVP_NCSTR;
inout_sqlda->sqlvar[0].sqldata = id;
inout_sqlda->sqlvar[0].sqlllen = strlen(id)+1;
inout_sqlda->sqlvar[0].sqlind = &idind;

inout_sqlda->sqlvar[1].sqltype = SQL_TVP_NCSTR;
inout_sqlda->sqlvar[1].sqldata = name;
inout_sqlda->sqlvar[1].sqlllen = strlen(name)+1;
inout_sqlda->sqlvar[1].sqlind = &nameind;

/* Set SQL Warning */
EXEC SQL WHENEVER SQLWARNING CONTINUE;

/* Call the remote procedure via CALL with SQLDA */
printf("Use CALL with SQLDA to invoke the Server Procedure\n");
idind = nameind = 0;

EXEC SQL CALL :procname USING DESCRIPTOR :*inout_sqlda;
printf("Server procedure complete.\n\n");
if (SQLCODE == SUCCESS)
{
   free(inout_sqlda);
   if (SUCCESS == Commit())
   {
      return(SUCCESS);
   }
   else
   {
      ProcessError(&sqlca);
      Rollback();
      return(FAILURE);
   }
}
else
{
   ProcessError(&sqlca);
   Rollback();
   free(inout_sqlda);
   return(FAILURE);
}
cselect.sqc:

/*------------------- cselect.sqc -------------------*/
/* Sample client program for stored procedure */
/* Last update = 12/29/98 */
/* Debra Eaton, IBM */
/* This client program will allocate and initialize */
/* data areas for a stored procedure. Also, the CALL*/
/* statement will communicate with the stored procedure */
/* Dependencies = SPSelectCount.sqc */
/*---------------------------------------------------------------------*/

/******************************************************** **
/* Include C library definitions */
/******************************************************** **
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sqlenv.h>
#include <sqlutil.h>
#include <sql.h>
#include <sqlda.h>
#include <sqlca.h>
#include "comval.h"
#include "database.h"

EXEC SQL INCLUDE SQLCA;

long int cselect()
{

  /* Declare host variables */
  EXEC SQL BEGIN DECLARE SECTION;
  char procname[255] = "/home/inst10/sp/proc!SPSelectCount";
  long number;
  char table[20];
  short numberind, tableind;
  EXEC SQL END DECLARE SECTION;

  /* Declare Variables for CALL */
  struct sqlca sqlca;
  struct sqlda *inout_sqlda = NULL;

  /* Initialize variables */
  fflush(stdin);
  printf("May I please have the ... 
"TABLE NAME: ");
  scanf("%s", &table);
  printf("\n You will count the rows in TABLE NAME %s\n",table);

  /* Allocate and Initialize Input SQLDA */
  /* sqldabc = (44 * sqln) + 16 bytes */
  inout_sqlda = (struct sqlda *)malloc(SQLDASIZE(2));
  inout_sqlda->sqln = 2;
  inout_sqlda->sqld = 2;
  inout_sqlda->sqldabc = 104;
inout_sqlda->sqlvar[0].sqltype = SQL_TYP_NCSTR;
inout_sqlda->sqlvar[0].sqldata = table;
inout_sqlda->sqlvar[0].sqllen = strlen(table)+1;
inout_sqlda->sqlvar[0].sqlind = &tableind;

inout_sqlda->sqlvar[1].sqltype = SQL_TYP_NINTEGER;
inout_sqlda->sqlvar[1].sqldata = (char *)&number;
inout_sqlda->sqlvar[1].sqllen = sizeof(number);
inout_sqlda->sqlvar[1].sqlind = &numberind;

/* Set SQL Warning */
EXEC SQL WHENEVER SQLWARNING CONTINUE;

/* Call the remote procedure via CALL with SQLDA */
printf("Use CALL with SQLDA to invoke the Server Procedure\n");
tableind = 0;
numberind = -1;
EXEC SQL CALL :procname USING DESCRIPTOR :*inout_sqlda;
printf("Server procedure complete.\n\n");
number = *(inout_sqlda->sqlvar[1].sqldata);
printf("The number of rows in the table was %d\n", number);
if (SQLCODE == SUCCESS)
{
    free(inout_sqlda);
    if (SUCCESS == Commit())
    {
        return(SUCCESS);
    }
    else
    {
        ProcessError(&sqlca);
        Rollback();
        return(FAILURE);
    }
}
else
{
    ProcessError(&sqlca);
    Rollback();
    free(inout_sqlda);
    return(FAILURE);
}
/** ****************************cupdate.sqc ***************************/
/*  Sample client program for stored procedure */
/*  Last update = 12/28/98 */
/*  Debra Eaton, IBM */
/******************************************************************************/
/*  This client program will allocate and initialize */
/*  data areas for a stored procedure. Also, the CALL*/
/*  statement will communicate with the stored procedure */
/******************************************************************************/
/*  Dependencies = SPUpdateRow.sqc */
/******************************************************************************/
/*/ Include C library definitions */
/******************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
EXEC SQL INCLUDE SQLCA;
long int cupdate()
{
    /* Declare host variables */
    EXEC SQL BEGIN DECLARE SECTION;
    char procname[255] = "/home/inst10/sqlib/function/proc!SPUpdateRow";
    char id[10];
    char name[20];
    short idind, nameind;
    EXEC SQL END DECLARE SECTION;
    /* Declare Variables for CALL */
    struct sqlca sqlca;
    struct sqlda *inout_sqlda = NULL;
    /* Initialize variables */
    fflush(stdin);
    printf("May I please have the ... \n");
    printf("ID to locate the row: ");
    scanf("%s", &id);
    printf("The new NAME: ");
    scanf("%s", &name);
    printf("You will update NAME to %s\n", name);
    printf("where the ID is %s\n", id);
    /* Allocate and Initialize Input SQLDA */
    /* sqlcdabc = (44 * sqln) + 16 bytes */
inout_sqlda = (struct sqlda *)malloc(SQLDASIZE(2));
inout_sqlda->sqln = 2;
inout_sqlda->sqld = 2;
inout_sqlda->sqldabc = 104;

inout_sqlda->sqlvar[0].sqltype = SQL_TYP_NCSTR;
inout_sqlda->sqlvar[0].sqldata = id;
inout_sqlda->sqlvar[0].sqllen = strlen(id)+1;
inout_sqlda->sqlvar[0].sqlind = &idind;

inout_sqlda->sqlvar[1].sqltype = SQL_TYP_NCSTR;
inout_sqlda->sqlvar[1].sqldata = name;
inout_sqlda->sqlvar[1].sqllen = strlen(name)+1;
inout_sqlda->sqlvar[1].sqlind = &nameind;

/* Set SQL Warning */
EXEC SQL WHENEVER SQLWARNING CONTINUE;

/* Call the remote procedure via CALL with SQLDA */
printf("Use CALL with SQLDA to invoke the Server Procedure\n");
idind = nameind = 0;

EXEC SQL CALL :procname USING DESCRIPTOR :*inout_sqlda;
printf("Server procedure complete.\n\n");
if (SQLCODE == SUCCESS)
{
    free(inout_sqlda);
    if (SUCCESS == Commit())
    {
        return(SUCCESS);
    }
    else
    {
        ProcessError(&sqlca);
        Rollback();
        return(FAILURE);
    }
}
else
{
    ProcessError(&sqlca);
    Rollback();
    free(inout_sqlda);
    return(FAILURE);
}
DB2 Resources

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

DB2 NewsGroup - news:comp.databases.ibm-db2


Software Migration Project Office - http://www.ibm.com/Solutions/softwaremigration

IBM DB2 Migration Portal - http://www.ibm.com/software/data/db2/migration

IBM Oracle/DB2 Conversion Redbook - http://www.rs6000.ibm.com/resource/aix_resource/Pubs/redbooks/htmlbooks/sg242567.00/2567fm.html (also accessible via http://www.redbooks.ibm.com)


Oracle/DB2 Conversion Tool - Conversion SQL Conversion Workbench by ManTech International Corporation (http://www.mantech.com)
Trademarks

The following terms are trademarks or registered trademarks of the IBM Corporation in the United States and/or other countries:

<table>
<thead>
<tr>
<th>Trademark</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIX</td>
<td></td>
</tr>
<tr>
<td>AIXwindows</td>
<td></td>
</tr>
<tr>
<td>AS/400</td>
<td></td>
</tr>
<tr>
<td>C Set++</td>
<td></td>
</tr>
<tr>
<td>C/370</td>
<td></td>
</tr>
<tr>
<td>DATABASE 2</td>
<td></td>
</tr>
<tr>
<td>DataJoiner</td>
<td></td>
</tr>
<tr>
<td>DataPropagator</td>
<td></td>
</tr>
<tr>
<td>DataRefresher</td>
<td></td>
</tr>
<tr>
<td>DB2</td>
<td></td>
</tr>
<tr>
<td>DB2 Connect</td>
<td></td>
</tr>
<tr>
<td>DB2 Universal Database</td>
<td></td>
</tr>
<tr>
<td>DB2 OLAP Server</td>
<td></td>
</tr>
<tr>
<td>Distributed Relational Database Architecture</td>
<td></td>
</tr>
<tr>
<td>DRDA</td>
<td></td>
</tr>
<tr>
<td>IBM</td>
<td></td>
</tr>
<tr>
<td>IMS</td>
<td>MVS/ESA</td>
</tr>
<tr>
<td></td>
<td>MVS/XA</td>
</tr>
<tr>
<td></td>
<td>OS/400</td>
</tr>
<tr>
<td></td>
<td>OS/390</td>
</tr>
<tr>
<td></td>
<td>OS/2</td>
</tr>
<tr>
<td></td>
<td>RISC System/6000</td>
</tr>
<tr>
<td></td>
<td>SQL/DS</td>
</tr>
<tr>
<td></td>
<td>SQL/400</td>
</tr>
<tr>
<td></td>
<td>S/370</td>
</tr>
<tr>
<td></td>
<td>System/370</td>
</tr>
<tr>
<td></td>
<td>System/390</td>
</tr>
<tr>
<td></td>
<td>VisualAge</td>
</tr>
<tr>
<td></td>
<td>VM/ESA</td>
</tr>
<tr>
<td></td>
<td>VSE/ESA</td>
</tr>
<tr>
<td></td>
<td>VTAM</td>
</tr>
<tr>
<td></td>
<td>WIN-OS/2</td>
</tr>
</tbody>
</table>

The following terms are trademarks or registered trademarks of the companies listed:

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

- **Microsoft, Windows, Windows NT, Windows 95** and the **Windows 98** are trademarks or registered trademarks of Microsoft Corporation in the United States, other countries, or both.

- **UNIX** is a registered trademark in the United States, other countries, or both and is licensed exclusively through X/Open Company Limited.

- **C++** is a trademark of American Telephone and Telegraph Company, Inc.

- **Oracle, Pro*C and PL/SQL** are trademarks of the Oracle Corporation.

- **SQL Conversion Workbench** is a trademark of ManTech Systems Solutions Corporation.

Other trademarks and trade names mentioned herein are the property of their respective owners.