DB2 SQL Workshop
for Experienced Users
(Course Code CF13)

Student Notebook

ERC 5.0

IBM Learning Services
Worldwide Certified Material
Trademarks

IBM® is a registered trademark of International Business Machines Corporation.

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

<table>
<thead>
<tr>
<th>AIX</th>
<th>AS/400</th>
<th>DataJoiner</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB2</td>
<td>DB2 Universal Database</td>
<td>IBM</td>
</tr>
<tr>
<td>OS/2</td>
<td>OS/390</td>
<td>OS/400</td>
</tr>
<tr>
<td>Perform</td>
<td>PowerPC</td>
<td>PS/2</td>
</tr>
<tr>
<td>QMF</td>
<td>SQL/DS</td>
<td></td>
</tr>
</tbody>
</table>

Linux is a registered trademark of Linus Torvalds in the United States and other countries.

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

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

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

November 2002 Edition

The information contained in this document has not been submitted to any formal IBM test and is distributed on an “as is” basis without any warranty either express or implied. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer’s ability to evaluate and integrate them into the customer’s operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will result elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk. The original repository material for this course has been certified as being Year 2000 compliant.

© Copyright International Business Machines Corporation 1998, 2002. All rights reserved. This document may not be reproduced in whole or in part without the prior written permission of IBM. Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.
Contents

Trademarks ............................................................... vii

Course Description ......................................................... ix

Agenda ................................................................. xi

Unit 1. Introduction ................................................... 1-1
  Objectives ................................................................. 1-2
  1.1 Introduction and SQL Basics Review ....................... 1-3
    DB2 Products ........................................................... 1-4
    DB2 Catalog ............................................................ 1-5
    A Database in DB2 UDB for UNIX, Windows, OS/2 ...... 1-7
    Subsystem in DB2 for OS/390 ...................................... 1-8
    Selecting Rows and Columns ...................................... 1-9
    Column Functions ..................................................... 1-11
    Summarizing Group Values ........................................ 1-13
    GROUP BY ROLLUP ..................................................... 1-15
    GROUP BY CUBE ........................................................ 1-17
    Grouping Function - Why? ......................................... 1-18
    Grouping Function - Example ...................................... 1-19
    GROUP BY Grouping Sets ............................................ 1-20
    Super Groups - Syntax ............................................... 1-21
    OLAP Function for Ranking ......................................... 1-22
    Additional SQL in This Course .................................... 1-23
    Checkpoint ............................................................ 1-24
    Summary ............................................................... 1-25

Unit 2. Create Objects ............................................... 2-1
  Objectives ............................................................... 2-3
  2.1 Creating Objects ................................................ 2-5
    Scenario (Part 1) ..................................................... 2-6
    Creating Tables and Adding Columns ......................... 2-8
    Creating Tables (Cont.) ............................................. 2-10
    Scenario (Part 2) ..................................................... 2-12
    Check Constraints - The Need ..................................... 2-13
    Check Constraints ................................................... 2-14
    Scenario Part 3 ...................................................... 2-16
    Create Index ........................................................ 2-18
    Referential Integrity (Part 1) .................................... 2-19
    Referential Integrity - Keys ....................................... 2-21
    Referential Integrity - Delete Rules ............................ 2-23
    Alter Table - Adding Referential Constraints ................ 2-24
    Referential Integrity (Part 2) .................................... 2-25
    Scenario (Part 4) ..................................................... 2-26
Triggers - The Need ................................................................. 2-27
Create Trigger ................................................................. 2-29
Create Trigger (Cont.) ..................................................... 2-31
View Considerations ......................................................... 2-33
Sample View ................................................................. 2-35
View Hierarchy ................................................................. 2-37
Read-Only Views ............................................................... 2-39
Changing Data Through a View ........................................... 2-41
Check Option ................................................................. 2-43
View Merge ................................................................. 2-45
View Materialization ......................................................... 2-46
Checkpoint (1 of 2) ........................................................... 2-47
Checkpoint (2 of 2) ........................................................... 2-48
Summary ................................................................. 2-49

Unit 3. Join. ........................................................................ 3-1
  Objectives ........................................................................ 3-2
  3.1 Join Review and Outer Join Techniques ......................... 3-3
    Sample Tables for Unit .................................................... 3-4
    Join ........................................................................ 3-5
    Inner Join .................................................................. 3-7
    Right Outer Join .......................................................... 3-8
    Left Outer Join ............................................................ 3-9
    Full Outer Join ............................................................. 3-10
    Joins of More Than Two Tables ....................................... 3-11
    Joins and Local Predicates (Part 1) ................................. 3-14
    Joins and Local Predicates (Part 2) ................................. 3-15
    Checkpoint ................................................................ 3-17
    Summary ................................................................... 3-18

Unit 4. CASE, CAST and Summary Tables ................................. 4-1
  Objectives ........................................................................ 4-2
  4.1 CASE, CAST and Summary Tables ................................. 4-3
    CASE Expressions in SELECT ........................................ 4-4
    CASE Expression in WHERE Clause ............................... 4-6
    CASE Expression in a Function ..................................... 4-7
    Nested CASE Expression ............................................. 4-8
    CAST Specifications ....................................................... 4-9
    Automatic Summary Tables ........................................... 4-11
    Summary Tables - Example .......................................... 4-12
    Summary Tables - Considerations .................................. 4-14
    Checkpoint ................................................................ 4-15
    Summary ................................................................... 4-16

Unit 5. Using Subqueries .......................................................... 5-1
  Objectives ........................................................................ 5-2
  5.1 Using Subqueries .......................................................... 5-3
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subquery With Basic Predicate</td>
<td>5-4</td>
</tr>
<tr>
<td>Visual Caption</td>
<td>5-6</td>
</tr>
<tr>
<td>Subquery With NOT IN Predicate</td>
<td>5-7</td>
</tr>
<tr>
<td>IN Predicate for Nullable Column</td>
<td>5-8</td>
</tr>
<tr>
<td>Subquery With ALL Predicate (Part 1)</td>
<td>5-10</td>
</tr>
<tr>
<td>Subquery With ALL Predicate (Part 2)</td>
<td>5-12</td>
</tr>
<tr>
<td>Subquery With ANY or SOME Predicate (Part 1)</td>
<td>5-14</td>
</tr>
<tr>
<td>Subquery With ANY or SOME Predicate (Part 2)</td>
<td>5-15</td>
</tr>
<tr>
<td>Subquery With EXISTS Predicate (Part 1)</td>
<td>5-17</td>
</tr>
<tr>
<td>Subquery With EXISTS Predicate (Part 2)</td>
<td>5-18</td>
</tr>
<tr>
<td>Correlated Subquery (Part 1)</td>
<td>5-20</td>
</tr>
<tr>
<td>Correlated Subquery (Part 2)</td>
<td>5-21</td>
</tr>
<tr>
<td>Correlated Subquery with EXISTS</td>
<td>5-22</td>
</tr>
<tr>
<td>Checkpoint</td>
<td>5-23</td>
</tr>
<tr>
<td>Summary</td>
<td>5-24</td>
</tr>
<tr>
<td>Unit 6. Scalar Functions</td>
<td>6-1</td>
</tr>
<tr>
<td>Objectives</td>
<td>6-3</td>
</tr>
<tr>
<td>6.1 Scalar Functions</td>
<td>6-5</td>
</tr>
<tr>
<td>Scalar Functions</td>
<td>6-6</td>
</tr>
<tr>
<td>Scalar Function - SUBSTR</td>
<td>6-7</td>
</tr>
<tr>
<td>Scalar Function - COALESCE/VALUE (Part 1)</td>
<td>6-9</td>
</tr>
<tr>
<td>Scalar Function - COALESCE/VALUE (Part 2)</td>
<td>6-11</td>
</tr>
<tr>
<td>Scalar Function - DECIMAL</td>
<td>6-12</td>
</tr>
<tr>
<td>Scalar Function - ROUND</td>
<td>6-14</td>
</tr>
<tr>
<td>Scalar Function - DIGITS</td>
<td>6-16</td>
</tr>
<tr>
<td>Scalar Function - CHAR (Part 1)</td>
<td>6-18</td>
</tr>
<tr>
<td>Scalar Function - CHAR (Part 2)</td>
<td>6-20</td>
</tr>
<tr>
<td>Scalar Function - LENGTH</td>
<td>6-21</td>
</tr>
<tr>
<td>Scalar Functions - LTRIM / RTRIM</td>
<td>6-23</td>
</tr>
<tr>
<td>Date-Related Scalar Functions (Part 1)</td>
<td>6-25</td>
</tr>
<tr>
<td>Date-Related Scalar Functions (Part 2)</td>
<td>6-27</td>
</tr>
<tr>
<td>Date-Related Scalar Functions (Part 3)</td>
<td>6-30</td>
</tr>
<tr>
<td>Date-Related Scalar Functions (Part 4)</td>
<td>6-32</td>
</tr>
<tr>
<td>Date/Time-Related Scalar Functions (Part 4)</td>
<td>6-34</td>
</tr>
<tr>
<td>Time/Timestamp-Related Scalar Functions (Part 2)</td>
<td>6-36</td>
</tr>
<tr>
<td>Labeled Durations</td>
<td>6-37</td>
</tr>
<tr>
<td>Checkpoint</td>
<td>6-39</td>
</tr>
<tr>
<td>Summary</td>
<td>6-40</td>
</tr>
<tr>
<td>Unit 7. Table Expressions and Recursive SQL</td>
<td>7-1</td>
</tr>
<tr>
<td>Objectives</td>
<td>7-2</td>
</tr>
<tr>
<td>7.1 Table Expressions and Recursive SQL</td>
<td>7-3</td>
</tr>
<tr>
<td>Nested Table Expressions</td>
<td>7-4</td>
</tr>
<tr>
<td>Nested Table Expressions in Joins (Part 1)</td>
<td>7-6</td>
</tr>
<tr>
<td>Nested Table Expressions in Joins (Part 2)</td>
<td>7-8</td>
</tr>
<tr>
<td>Nested Table Expressions in Joins (Part 3)</td>
<td>7-9</td>
</tr>
</tbody>
</table>
Nested Table Expressions in Joins (Part 4) ..................................................... 7-10
Common Table Expressions .............................................................. 7-11
Recursive SQL ................................................. 7-13
Recursive SQL - Initialization Select .............................................. 7-15
Recursive SQL - First Iteration ............................................... 7-16
Recursive SQL - Second Iteration .............................................. 7-18
Main Select ...................................................... 7-19
Controlling Depth of Recursion (Part 1) ........................................ 7-20
Controlling Depth of Recursion (Part 2) ........................................ 7-22
Recursive SQL - Recommendations ........................................... 7-23
Checkpoint ......................................................... 7-24
Summary ............................................................. 7-25

Unit 8. UDTs/UDFs and Performance .............................................. 8-1
Objectives ............................................................. 8-2
8.1 User-Defined Distinct Types .............................................. 8-3
User-Defined Distinct Types - The Need ........................................ 8-4
User-Defined Distinct Types - Definition .................................... 8-5
User-Defined Functions - The Need ........................................... 8-7
User-Defined Functions - Sourced ............................................. 8-8
User-Defined Functions - External ........................................... 8-10
8.2 Performance Considerations ............................................... 8-11
Index Concepts ......................................................... 8-12
Index Structure .......................................................... 8-13
Predicates ............................................................... 8-15
Predicate Processing .......................................................... 8-16
Index Utilization ............................................................... 8-18
The Optimizer ............................................................. 8-21
General Guidelines - Examine Correlated Subqueries ...................... 8-23
General Guidelines - Minimize DB2 Sorts ................................... 8-25
General Guidelines - Use Views ................................................ 8-26
General Guidelines - Expressions ............................................ 8-27
General Guidelines - '<>' Predicates ........................................ 8-28
General Guidelines - Arithmetic ............................................. 8-29
General Guidelines - Search Value Length ................................... 8-30
General Guidelines - Conversions ............................................. 8-31
General Guidelines - Statements ............................................. 8-32
Checkpoint ............................................................... 8-34
Summary ............................................................. 8-35

Appendix A. Checkpoint Solutions .............................................. A-1

Bibliography .............................................................. X-1

Index .............................................................. X-3
Trademarks

The reader should recognize that the following terms, which appear in the content of this training document, are official trademarks of IBM or other companies:

IBM® is a registered trademark of International Business Machines Corporation.

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

- AIX
- DB2
- OS/2
- Perform
- QMF
- AS/400
- DB2 Universal Database
- OS/390
- PowerPC
- SQL/DS
- DataJoiner
- IBM
- OS/400
- PS/2
- Linux is a registered trademark of Linus Torvalds in the United States and other countries.
- Windows and Windows NT are trademarks of Microsoft Corporation in the United States, other countries, or both.
- UNIX is a registered trademark of The Open Group in the United States and other countries.

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

DB2 SQL Workshop for Experienced Users

Duration : 2.5 days

Purpose

This course teaches you how to make use of advanced SQL techniques to access DB2 databases in different environments. This course is appropriate for customers working in all DB2 environments, that is OS/390, z/OS, VM/VSE, AS/400, UNIX, Windows and OS/2.

Audience

Experienced SQL end users, application programmers, database administrators, and user support staff who need more advanced knowledge of SQL.

Prerequisites

Before taking this course, you should have experience with coding and executing basic SQL statements.

These skills can be developed by attending CF12, DB2 SQL Workshop, or equivalent experience.

Objectives

After completing this course, you should be able to:

- Use advanced SQL constructs including: recursive SQL, table expressions, check constraints, and triggers
- Discuss basic relational database concepts including: databases, tables, indexes, and referential integrity
- Create tables, views and indexes
- Use outer join
- Use CASE and CAST
- Use complex subqueries
- Use several of the most popular scalar SQL functions
• Use the 'new' GROUPING functions CUBE, ROLLUP, and RANK
• Understand the impact of Summary Tables
• Understand User-Defined Distinct Types and User-Defined Functions
• Avoid several of the most common causes for poorly-performing SQL

Contents

This course covers the following major topics:
• Introduction and review
• Creating objects (tables, indexes, views....)
• Join
• CASE, CAST and Summary Tables
• Subqueries
• Scalar Functions
• Table Expressions and Recursive SQL
• User-Defined Distinct Types and Functions
• Performance Considerations
Agenda

Day 1

Welcome
Unit 1 - Introduction (SQL Basics Review + ROLLUP, CUBE and RANK)
Unit 2 - Create Objects
Unit 3 - Join

Day 2

Unit 4 - CASE, CAST and Summary Tables
Unit 5 - Using Subqueries
Unit 6 - Scalar Functions

Day 3

Unit 7 - Table Expressions and Recursive SQL
Unit 8 - UDTs/UDFs and Performance
Unit 1. Introduction

What This Unit Is About

This unit explains the key environmental differences between the different DB2 platforms. There is also a brief review of SQL basics and theory on CUBE, ROLLUP, and RANK.

What You Should Be Able to Do

After completing this unit, you should be able to:

- Describe the key differences among the IBM DB2 platforms
- Identify the purpose of the clauses in the SELECT statement
- Use the new grouping functions: ROLLUP, CUBE, and RANK.

How You Will Check Your Progress

Accountability:

- Unit Checkpoint
- Machine Lab

References

DB2 UDB for OS/390 and z/OS V7
  SQL Reference (SC26-9944-02)
DB2 Administration Guide (DB2 UDB 7.2 for Windows)
DB2 SQL Reference (DB2 UDB 7.2 for Windows)
Objectives

After completing this unit, you should be able to:

- Identify the purpose of the clauses in the SELECT statement
- Describe the key differences among the IBM DB2 platforms
- Describe and use the *super groups* feature (ROLLUP and CUBE)
- Describe and use the OLAP function for ranking

Figure 1-1. Objectives

Notes:
1.1 Introduction and SQL Basics Review
Notes:

- DB2 is a relational database management system (RDBMS) that enables users to create, update, and control relational databases using Structured Query Language (SQL). Designed to meet the information needs of small and large businesses alike, it is available on a variety of platforms, including large systems (z/OS, OS/390, VM, and VSE); mid-sized systems (OS/400®, AIX®); single-user or LAN-based systems (OS/2, DOS, Windows 95, and Windows NT) and on Hewlett-Packard, Sun, or Siemens-Nixdorf UNIX operating systems as well as Linux. All members of the DB2 family are based on the same relational theory and as a result, have a commonality of features and functions that ensure portability from one platform to another.

- DB2 UDB users have the power of distributed query across any DB2 family database or OLE DB source. Users and applications can use the DB2 UDB SQL syntax and APIs to access data that resides at heterogeneous data sources. With this functionality, users and applications have the capability of referencing multiple data sources in a single SQL statement. With Relational Connect, distributed queries can also include Oracle, Sybase, SQL Server, and Informix databases.
DB2 System Catalog Tables

- A set of System Catalog Tables is created and maintained to administer the DB2 Objects.

- The System Catalog Tables can be read with SQL.

- Some of the System Catalog Tables include information on:
  - Table/Index definitions
  - Column data types
  - Defined constraints
  - Object dependencies
  - Object privileges
  - Statistical information about objects

**Notes:**

- DB2 stores all information about your tables, indexes, columns, and other objects in the DB2 catalog.

- The DB2 catalog consists of normal tables and views, just like the ones in your own databases. They can be read via SQL, just like your own tables.

- On some platforms, including DB2 for OS/390, catalog information is presented in tables. On other platforms, including DB2 UDB for UNIX, Windows and OS/2, catalog information is also presented in views. The *SQL Reference* for that platform contains full documentation of the catalog tables or views, whichever is preferred for that platform.

- Catalog tables and views can be distinguished by their high-level qualifiers, that is, authorization ID or schema names. The authorization ID used for catalog tables in DB2 for OS/390 is SYSIBM. SYSIBM.SYSTABLES is a commonly-used catalog table in DB2 for OS/390. SYSCAT and SYSSTAT are the schema names used for catalog views in DB2 UDB for UNIX, Windows and OS/2. SYSCAT.TABLES is a commonly-used catalog view in DB2 UDB UNIX, Windows and OS/2.
SYSIBM.SYSTABLES/SYSCAT.TABLES contains information about every table and view. SYSIBM.SYSCOLUMNS/SYSCAT.COLUMNS describes every column of every table and view, and SYSIBM.SYSTABAUTH/SYSCAT.TABAUTH lists table privileges granted to authorization IDs.

DB2 uses the catalog extensively as you work. For example, if you execute a query against a table named EMPLOYEE, DB2 does the following:

- looks in (SYS)TABLES to see if the EMPLOYEE table exists
- looks in (SYS)TABAUTH to see whether you are allowed to access the EMPLOYEE table
- uses (SYS)COLUMNS to get the column names and the column sequence for the EMPLOYEE table so that the data can be obtained for you

If the table exists and you are authorized to use it, DB2 will present the result to you. Otherwise, it will return an error message identifying the problem.

In DB2 UDB for AS/400® the schema names of the catalog tables and views are QSYS and/or QSYS2.

Note: The DB2 Catalog is fully documented in the SQL Reference for each platform.
Notes:

- Each DB2 UDB for UNIX, Windows and OS/2 database has its own catalog.
- A database is a collection of tables. A table consists of a defined number of columns and a varying number of rows.
- In DB2 UDB for UNIX, Windows and OS/2, you connect to a database with the CONNECT TO statement.
  Currently a single application in DB2 UDB for Windows, UNIX, OS/2 cannot join tables from two different databases at the same time (without using the Federated Database capability of DB2 UDB for Windows, UNIX, OS/2 V7 or higher, or without using a separate product like Data Joiner [Ref.: http://www.software.ibm.com/data/datajoiner/]), so it is important to put related tables in the same database.
- A DB2 Federated Database system enables users and applications to reference multiple database management systems or databases within a single SQL statement.
Notes:

- In the OS/390 environment, a DB2 subsystem is similar to a database in the Workstation environment because there is one catalog for the subsystem. They are also different since a DB2 subsystem usually contains many databases. In DB2 for OS/390, a database is a name for a group of related tables.

- You can connect to a subsystem explicitly with SQL statements, but connections are usually made implicitly via parameters in your DB2 software. For example, the subsystem name is identified in the DB2I (DB2 Interactive) Defaults panel so that DB2I can make the connection for you when you use tools like SPUFI. An end user does not have to know the name of the database to which his table belongs.

- Joins between tables which are located in different databases of the same subsystem are possible. Joins between tables which are located in different subsystems are not possible.
Selecting Rows and Columns

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>LASTNAME</th>
<th>SALARY</th>
<th>JOB</th>
<th>EDLEVEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>000120</td>
<td>O'CONNELL</td>
<td>29250.00</td>
<td>CLERK</td>
<td>14</td>
</tr>
<tr>
<td>000100</td>
<td>SPENSER</td>
<td>26150.00</td>
<td>MANAGER</td>
<td>14</td>
</tr>
<tr>
<td>000130</td>
<td>QUINTANA</td>
<td>23800.00</td>
<td>ANALYST</td>
<td>16</td>
</tr>
<tr>
<td>000280</td>
<td>SCHNEIDER</td>
<td>26250.00</td>
<td>OPERATOR</td>
<td>17</td>
</tr>
<tr>
<td>000250</td>
<td>SMITH</td>
<td>19180.00</td>
<td>CLERK</td>
<td>15</td>
</tr>
<tr>
<td>000060</td>
<td>STERN</td>
<td>32250.00</td>
<td>MANAGER</td>
<td>16</td>
</tr>
</tbody>
</table>

EMPLOYEE Table

```
SELECT LASTNAME, SALARY, EMPNO
FROM EMPLOYEE
WHERE EDLEVEL <= 16
    AND SALARY BETWEEN 19000 AND 30000
    AND LASTNAME LIKE 'S%'
    AND JOB IN ('MANAGER', 'OPERATOR', 'ANALYST')
ORDER BY LASTNAME
```

Result

<table>
<thead>
<tr>
<th>LASTNAME</th>
<th>SALARY</th>
<th>EMPNO</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPENSER</td>
<td>26150.00</td>
<td>000100</td>
</tr>
</tbody>
</table>

Figure 1-6. Selecting Rows and Columns

Notes:

- Select the column or columns by naming each column you want in the SELECT clause of the SELECT statement. The columns will appear in the result set in the left-to-right order you specify in the SELECT clause, regardless of their actual sequence in the table.
- Identify the table(s) or view(s) that contain your data with the FROM clause.
- Use a WHERE clause to select the rows that you want. A WHERE clause specifies one or more predicates. A predicate specifies a test you want DB2 to apply to each table row.
  - Use LIKE to specify a character string that is similar to the column values of rows you want to select. Two wild-card characters can be used in the comparison string. Use a percent sign (%) to indicate any string of zero or more characters. Use an underscore (_) to indicate any single character.
- You can use BETWEEN to select rows in which a column has a value within a range. Specify the lower boundary of the BETWEEN predicate first, then the upper boundary. The range is inclusive.

- You can use the IN predicate to select each row that has a column value equal to one of several listed values.

• Use an ORDER BY clause to sort the data in the result set.

• The example requests three columns, LASTNAME, SALARY, and EMPNO, from the EMPLOYEE table. Only rows which have an education level (EDLEVEL) of less than 16 or equal to 16, a yearly salary (SALARY) in the range from 19,000 to 30,000 inclusive, a last name that starts with 'S', and a job of 'MANAGER', 'OPERATOR', or 'ANALYST' will appear in the result. The result will be sorted in ascending sequence by last name.
Column Functions

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>LASTNAME</th>
<th>SALARY</th>
<th>JOB</th>
<th>EDLEVEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>000120</td>
<td>O'CONNELL</td>
<td>29250.00</td>
<td>CLERK</td>
<td>14</td>
</tr>
<tr>
<td>000100</td>
<td>SPENSER</td>
<td>26150.00</td>
<td>MANAGER</td>
<td>14</td>
</tr>
<tr>
<td>000130</td>
<td>QUINTANA</td>
<td>23800.00</td>
<td>ANALYST</td>
<td>16</td>
</tr>
<tr>
<td>000280</td>
<td>SCHNEIDER</td>
<td>26250.00</td>
<td>OPERATOR</td>
<td>17</td>
</tr>
<tr>
<td>000250</td>
<td>SMITH</td>
<td>19180.00</td>
<td>CLERK</td>
<td>15</td>
</tr>
<tr>
<td>000060</td>
<td>STERN</td>
<td>32250.00</td>
<td>MANAGER</td>
<td>16</td>
</tr>
</tbody>
</table>

EMPLOYEE Table

```
SELECT SUM(SALARY) AS TOTAL, MIN(SALARY) AS MINIMUM,
MAX(SALARY) AS MAXIMUM,
DECIMAL(AVG(SALARY),8,2) AS AVERAGE,
COUNT(*) AS #EMP,
COUNT(DISTINCT EDLEVEL) AS #LVL
FROM EMPLOYEE
```

<table>
<thead>
<tr>
<th>TOTAL</th>
<th>MINIMUM</th>
<th>MAXIMUM</th>
<th>AVERAGE</th>
<th>#EMP</th>
<th>#LVL</th>
</tr>
</thead>
<tbody>
<tr>
<td>156880.00</td>
<td>19180.00</td>
<td>32250.00</td>
<td>26146.66</td>
<td>6</td>
<td>4</td>
</tr>
</tbody>
</table>

Figure 1-7. Column Functions

**Notes:**

- A column function returns a single value for a group of rows. The column function operates on the column specified in the function argument and examines all the rows that satisfy the WHERE clause. If no GROUP BY is present, a column function returns a single value, no matter how many rows are read. The main column functions are as follows:

  - `SUM()` - Returns the total value.
  - `MIN()` - Returns the minimum value.
  - `AVG()` - Returns the average value.
  - `MAX()` - Returns the maximum value.
  - `COUNT(DISTINCT)` - Returns the number of unique values.
  - `COUNT(*)` - Returns the number of selected rows.

Some newer column-functions are:

  - `COUNT_BIG(*)` - Like `COUNT(*)` but result can be bigger than integer.
  - `STDDEV()` - Returns the standard deviation.
  - `VARIANCE()` - Returns the variance.
The example uses the function DECIMAL which is not a column function, but a scalar function. Scalar functions can be used whenever an expression can be used. They apply to all rows of the result rather than returning a value for a group of rows.

- You can use SUM(), AVG(), VARIANCE() and STDDEV() only with values that are stored in columns defined as numeric. You can use MIN(), MAX(), COUNT(DISTINCT) and COUNT_BIG(DISTINCT) on any type of column. COUNT(*) and COUNT_BIG(*) do not have a column name as its argument so they can also be used on columns of any data type.

- Whenever an expression is used for a column of the result table, the column does not have a name unless an AS clause assigns a name to the column. AS clauses are very useful in conjunction with column functions since they allow you to give the result of the column function a meaningful name.

- The example reads all rows of the table on the visual and determines the total yearly salary paid to all employees, the lowest yearly salary, the highest yearly salary, the average yearly salary, the number of employees, and the number of different education levels. In the result table, the corresponding columns are named TOTAL, MINIMUM, MAXIMUM, AVERAGE, #EMP, and #LVL, respectively.
Summarizing Group Values

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>LASTNAME</th>
<th>SALARY</th>
<th>JOB</th>
<th>EDLEVEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>000120</td>
<td>O’CONNELL</td>
<td>29250.00</td>
<td>CLERK</td>
<td>14</td>
</tr>
<tr>
<td>000100</td>
<td>SPENSER</td>
<td>26150.00</td>
<td>MANAGER</td>
<td>14</td>
</tr>
<tr>
<td>000130</td>
<td>QUINTANA</td>
<td>23800.00</td>
<td>ANALYST</td>
<td>16</td>
</tr>
<tr>
<td>000280</td>
<td>SCHNEIDER</td>
<td>26250.00</td>
<td>OPERATOR</td>
<td>17</td>
</tr>
<tr>
<td>000250</td>
<td>SMITH</td>
<td>19180.00</td>
<td>CLERK</td>
<td>15</td>
</tr>
<tr>
<td>000060</td>
<td>STERN</td>
<td>32250.00</td>
<td>MANAGER</td>
<td>16</td>
</tr>
</tbody>
</table>

EMPLOYEE Table

SELECT EDLEVEL, SUM(SALARY) AS GROUP_TOTAL
FROM EMPLOYEE
GROUP BY EDLEVEL
HAVING COUNT(*) > 1

<table>
<thead>
<tr>
<th>EDLEVEL</th>
<th>GROUP_TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>55400.00</td>
</tr>
<tr>
<td>16</td>
<td>56050.00</td>
</tr>
</tbody>
</table>

Figure 1-8. Summarizing Group Values

Notes:

• GROUP BY is used to make DB2 summarize information for different groups within the data. For example, you could use GROUP BY to get the average salary for each department in a company.

• The GROUP BY clause may identify one or more columns. When GROUP BY includes more than one column, there will be one row in the result for each combination of values that exist in the columns named in the GROUP BY clause.

• When you use GROUP BY, the SELECT clause normally contains a mix of column names and column functions: the GROUP BY clause must specify all of the columns which are not arguments of the column functions in the SELECT clause.

• The GROUP BY clause is optional, but when it is used, it must follow the WHERE clause (or the FROM clause if there is no WHERE clause) and precede the HAVING clause (or the ORDER BY clause if there is no HAVING clause).
• Use HAVING to specify additional search conditions that the retrieved **groups** must satisfy. The HAVING clause acts like a WHERE clause for groups. It can contain the same kind of predicates as the WHERE clause.

A HAVING clause can also contain column functions like COUNT(*), unlike the WHERE clause which cannot directly use column functions. The search conditions in the HAVING clause test properties of each group rather than properties of individual rows in the group.

• The example reads the table in the graphic and produces a one-row summary for each of the education levels in the table. Each summary row contains the education level and the sum of the yearly salaries (GROUP_TOTAL) paid to people in the group. Education levels that have only one member are excluded from the result set by means of the HAVING clause.
GROUP BY ROLLUP

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>LASTNAME</th>
<th>SALARY</th>
<th>JOB</th>
<th>EDLEVEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>000120</td>
<td>O’CONNELL</td>
<td>29250.00</td>
<td>CLERK</td>
<td>14</td>
</tr>
<tr>
<td>000100</td>
<td>SPENSER</td>
<td>26150.00</td>
<td>MANAGER</td>
<td>14</td>
</tr>
<tr>
<td>000130</td>
<td>QUINTANA</td>
<td>23800.00</td>
<td>ANALYST</td>
<td>16</td>
</tr>
<tr>
<td>000280</td>
<td>SCHNEIDER</td>
<td>26250.00</td>
<td>OPERATOR</td>
<td>17</td>
</tr>
<tr>
<td>000250</td>
<td>SMITH</td>
<td>19180.00</td>
<td>CLERK</td>
<td>15</td>
</tr>
<tr>
<td>000060</td>
<td>STERN</td>
<td>32250.00</td>
<td>MANAGER</td>
<td>16</td>
</tr>
</tbody>
</table>

**Notes:**

- GROUP BY ROLLUP is used when you need to analyze a collection of data in a single dimension, but at more than one level of detail. The example above gives the average salary by edlevel and the overall average salary.

- GROUP BY ROLLUP is part of SUPER GROUPS. You usually employ these functions in online analytical processing, or OLAP, where there is a large collection of data spanning several dimensions.

- GROUP BY ROLLUP can also be used on several levels. When you use the same EMPLOYEE table as above (restricted to certain employees only), you obtain following result:

```
SELECT EDLEVEL, JOB, AVG(SALARY) AS AVG_SALARY
FROM EMPLOYEE
WHERE EMPNO IN ('000060', '000100', '000120', '000130', '000250', '000280')
GROUP BY ROLLUP (EDLEVEL, JOB)
```
<table>
<thead>
<tr>
<th>EDLEVEL</th>
<th>JOB</th>
<th>AVG_SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>- -</td>
<td>CLERK</td>
<td>29250,00000000000000000000000000000</td>
</tr>
<tr>
<td>14 -</td>
<td>MANAGER</td>
<td>26150,00000000000000000000000000000</td>
</tr>
<tr>
<td>15 -</td>
<td>CLERK</td>
<td>19180,00000000000000000000000000000</td>
</tr>
<tr>
<td>16 -</td>
<td>ANALYST</td>
<td>23800,00000000000000000000000000000</td>
</tr>
<tr>
<td>16 -</td>
<td>MANAGER</td>
<td>32250,00000000000000000000000000000</td>
</tr>
<tr>
<td>17 -</td>
<td>OPERATOR</td>
<td>26250,00000000000000000000000000000</td>
</tr>
</tbody>
</table>

11 record(s) selected.

- In this example, we get the average salary for all employees, the average salary per edlevel and the average salary per combination of edlevel and job.
GROUP BY CUBE

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>LASTNAME</th>
<th>SALARY</th>
<th>JOB</th>
<th>EDLEVEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>000120</td>
<td>O'CONNELL</td>
<td>29250.00</td>
<td>CLERK</td>
<td>14</td>
</tr>
<tr>
<td>000100</td>
<td>SPENSER</td>
<td>26150.00</td>
<td>MANAGER</td>
<td>14</td>
</tr>
<tr>
<td>000130</td>
<td>QUINTANA</td>
<td>23800.00</td>
<td>ANALYST</td>
<td>16</td>
</tr>
<tr>
<td>000280</td>
<td>SCHNEIDER</td>
<td>26250.00</td>
<td>OPERATOR</td>
<td>17</td>
</tr>
<tr>
<td>000250</td>
<td>SMITH</td>
<td>19180.00</td>
<td>CLERK</td>
<td>15</td>
</tr>
<tr>
<td>000060</td>
<td>STERN</td>
<td>32250.00</td>
<td>MANAGER</td>
<td>16</td>
</tr>
</tbody>
</table>

EMPLOYEE Table

```sql
SELECT EDLEVEL, JOB, 
DECIMAL(AVG(SALARY),8,2) AS AVG_SALARY
FROM EMPLOYEE
WHERE EDLEVEL IN(14,15)
GROUP BY CUBE(JOB,EDLEVEL)
```

Result

<table>
<thead>
<tr>
<th>EDLEVEL</th>
<th>JOB</th>
<th>AVG_SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>-</td>
<td>27700.00</td>
</tr>
<tr>
<td>15</td>
<td>-</td>
<td>19180.00</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>24860.00</td>
</tr>
<tr>
<td>-</td>
<td>CLERK</td>
<td>24215.00</td>
</tr>
<tr>
<td>-</td>
<td>MANAGER</td>
<td>26150.00</td>
</tr>
<tr>
<td>14</td>
<td>CLERK</td>
<td>29250.00</td>
</tr>
<tr>
<td>15</td>
<td>CLERK</td>
<td>19180.00</td>
</tr>
<tr>
<td>14</td>
<td>MANAGER</td>
<td>26150.00</td>
</tr>
</tbody>
</table>

Figure 1-10. GROUP BY CUBE

Notes:

- GROUP BY CUBE is used when you need to analyze a collection of data in a several dimensions. The above example gives both average salary by edlevel, by job, by the combination of edlevel and job and the overall average salary.
### Grouping Function - Why?

#### EMPLOYEE Table

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>LASTNAME</th>
<th>SALARY</th>
<th>JOB</th>
<th>EDLEVEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>000120</td>
<td>O'CONNELL</td>
<td>29250.00</td>
<td>CLERK</td>
<td>14</td>
</tr>
<tr>
<td>000100</td>
<td>SPENSER</td>
<td>26150.00</td>
<td>MANAGER</td>
<td>14</td>
</tr>
<tr>
<td>000130</td>
<td>QUINTANA</td>
<td>23800.00</td>
<td>ANALYST</td>
<td>16</td>
</tr>
<tr>
<td>000280</td>
<td>SCHNEIDER</td>
<td>26250.00</td>
<td>OPERATOR</td>
<td>17</td>
</tr>
<tr>
<td>000250</td>
<td>SMITH</td>
<td>19180.00</td>
<td>CLERK</td>
<td>-</td>
</tr>
<tr>
<td>000060</td>
<td>STERN</td>
<td>32250.00</td>
<td>MANAGER</td>
<td>16</td>
</tr>
</tbody>
</table>

**SELECT** EDLEVEL, **DECIMAL**(AVG(SALARY),8,2) **AS** AVG_SALARY **FROM** EMPLOYEE **GROUP BY** ROLLUP(EDLEVEL)

<table>
<thead>
<tr>
<th>EDLEVEL</th>
<th>AVG_SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>26146.66</td>
</tr>
<tr>
<td>14</td>
<td>27700.00</td>
</tr>
<tr>
<td>-</td>
<td>19180.00</td>
</tr>
<tr>
<td>16</td>
<td>28025.00</td>
</tr>
<tr>
<td>17</td>
<td>26250.00</td>
</tr>
</tbody>
</table>

**Figure 1-11.** Grouping Function - Why?

**Notes:**

- When using CUBE and ROLLUP it is sometimes difficult to tell what kind group the row represents. In the visual above there are two rows with NULL values in EDLEVEL. One of these rows represents the group that contains all rows selected in the WHERE clause, the other row represent the row with unknown (NULL) EDLEVEL.
Grouping Function - Example

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>LASTNAME</th>
<th>SALARY</th>
<th>JOB</th>
<th>EDLEVEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>000120</td>
<td>O'CONNELL</td>
<td>29250.00</td>
<td>CLERK</td>
<td>14</td>
</tr>
<tr>
<td>000100</td>
<td>SPENGER</td>
<td>26150.00</td>
<td>MANAGER</td>
<td>14</td>
</tr>
<tr>
<td>000130</td>
<td>QUINTANA</td>
<td>23800.00</td>
<td>ANALYST</td>
<td>16</td>
</tr>
<tr>
<td>000280</td>
<td>SCHNEIDER</td>
<td>26250.00</td>
<td>OPERATOR</td>
<td>17</td>
</tr>
<tr>
<td>000250</td>
<td>SMITH</td>
<td>19180.00</td>
<td>CLERK</td>
<td>-</td>
</tr>
<tr>
<td>000060</td>
<td>STERN</td>
<td>32250.00</td>
<td>MANAGER</td>
<td>16</td>
</tr>
</tbody>
</table>

Notes:

- In CUBE and ROLLUP, the grouping function can be applied to any of the columns or expressions used inside CUBE or ROLLUP.
- Grouping function gives 1 or 0 for each row. 1 means that this row represents a higher grouping, 0 means that grouping is on the same level as grouping function uses.
- In the result-set above the second column represents grouping function. For the first row the value 1 tells us that this row represents a higher grouping (aggregation) than grouping(edlevel). This means in our example that the average salary on this rows represents all employees. For all other rows, the value 0 tells us that each row is grouped by edlevel-level and the third row in the result-set is the average salary for all employees with unknown (NULL) edlevel.
- This grouping function is very useful together with CASE. We will cover CASE in Unit 4.
**GROUP BY Grouping Sets**

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>LASTNAME</th>
<th>SALARY</th>
<th>JOB</th>
<th>BIRTHDATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>000120</td>
<td>O’CONNELL</td>
<td>29250.00</td>
<td>CLERK</td>
<td>1942-10-18</td>
</tr>
<tr>
<td>000100</td>
<td>SPENSER</td>
<td>26150.00</td>
<td>MANAGER</td>
<td>1956-12-18</td>
</tr>
<tr>
<td>000130</td>
<td>QUINTANA</td>
<td>23800.00</td>
<td>ANALYST</td>
<td>1925-09-15</td>
</tr>
<tr>
<td>000280</td>
<td>SCHNEIDER</td>
<td>26250.00</td>
<td>OPERATOR</td>
<td>1936-03-28</td>
</tr>
<tr>
<td>000250</td>
<td>SMITH</td>
<td>19180.00</td>
<td>CLERK</td>
<td>1939-11-12</td>
</tr>
<tr>
<td>000060</td>
<td>STERN</td>
<td>32250.00</td>
<td>MANAGER</td>
<td>1945-07-07</td>
</tr>
</tbody>
</table>

**EMPLOYEE Table**

```sql
SELECT JOB, DAY(BIRTHDATE),
DECIMAL(AVG(SALARY),8,2) AS AVG_SALARY
FROM EMPLOYEE
WHERE YEAR(BIRTHDATE) > 1938
GROUP BY
GROUPING SETS (DAY(BIRTHDATE), JOB)
```

<table>
<thead>
<tr>
<th>JOB</th>
<th>DAY</th>
<th>AVG_SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLERK</td>
<td>-</td>
<td>24215.00</td>
</tr>
<tr>
<td>MANAGER</td>
<td>-</td>
<td>29200.00</td>
</tr>
<tr>
<td>-</td>
<td>7</td>
<td>32250.00</td>
</tr>
<tr>
<td>-</td>
<td>12</td>
<td>19180.00</td>
</tr>
<tr>
<td>-</td>
<td>18</td>
<td>27700.00</td>
</tr>
</tbody>
</table>

**Notes:**

- The GROUPING SETS operator lets you list exactly the groups you want.
- GROUP BY GROUPING SETS((JOB),(DAY(BIRTHDATE))) is equivalent to
  - GROUP BY JOB
  - GROUP BY DAY(BIRTHDATE)
- If some of the grouping criteria include more than one column or expression, enclose those grouping criteria in parentheses.
- Several grouping specifications can be combined in the same GROUP BY clause. The total number of groups produced by the GROUP BY clause is the product of the number of groups produced by each grouping specification.
Super Groups - Syntax

Notes:
- This visual shows as a summary a simplified syntax of the GROUP BY clause. For the complete syntax, please refer to the SQL Reference manual.
Notes:
If we want to order the employees according to their salaries in a result table, where the employees should be listed in ascending sequence by their employee number, we can utilize the RANK function.

On-Line Analytical Processing (OLAP) functions provide the ability to return ranking, row numbering and existing column function information as a scalar value in a query result.

The ranking function computes the ordinal rank of a row within the window. Rows that are not distinct with respect to the ordering within their window are assigned the same rank. The results of ranking may be defined with or without gaps in the numbers resulting from duplicate values.

If RANK is specified, the rank of a row is defined as 1 plus the number of rows that strictly precede the row. Thus, if two or more rows are not distinct with respect to the ordering, then there will be one or more gaps in the sequential rank numbering.

We are informed that SMITH has the lowest salary. (His rank is 6).
Additional SQL in This Course

Notes:
The visual lists the major topics we will discuss during the rest of the course.
Checkpoint

1. List three operating systems that can run DB2.

2. T/F. There is one DB2 catalog for an entire DB2 for OS/390 subsystem, which can contain many databases, and a separate DB2 catalog for each database in DB2 UDB for UNIX, Windows and OS/2.

3. Assume that you need to list the minimum, the maximum, and the average salary in your company. Which functions do you use?

4. Assume you have to prepare a report that lists each union classification on a line by itself, along with the total salary paid out to all employees in that union classification. Which clause will you need in your SQL statement to ensure that there is a separate line on the report for each union classification?

Notes:
Write down your answers here:

1. 

2. 

3. 

4. 

You will find an example of the answers in the Appendix.
Summary

Now that you have completed this unit, you should be able to:

- Identify the purpose of the clauses in the SELECT statement
- Describe the key differences among the IBM DB2 platforms
- Describe and use the *super groups* feature (ROLLUP and CUBE)
- Describe and use the OLAP function for ranking

Notes:
Unit 2. Create Objects

What This Unit Is About

This unit examines referential integrity, constraints, and triggers and illustrates how to create them with SQL.

What You Should Be Able to Do

After completing this unit, you should be able to:

- Code statements to
  - Create tables and views
  - Alter tables
  - Create indexes
  - Implement referential integrity (RI)
  - Define triggers and check constraints
- Identify impacts and advantages of referential integrity, including impacts of delete rules
- Identify considerations when using triggers and check constraints

How You Will Check Your Progress

Accountability:

- Unit Checkpoint
- Machine Lab

Labs

- CREATE TABLE
  — CHECK CONSTRAINT
- ALTER TABLE
  — ADD PRIMARY KEY
  — ADD FOREIGN KEY
- CREATE INDEX
- CREATE TRIGGERS
- CREATE VIEWS
References

DB2 UDB for OS/390 and z/OS V7
   SQL Reference (SC26-9944-02)

DB2 Administration Guide (DB2 UDB 7.2 for Windows)
DB2 SQL Reference (DB2 UDB 7.2 for Windows)
Objectives

After completing this unit, you should be able to:

- Code statements to:
  - Create tables and views
  - Alter tables
  - Create indexes
  - Implement referential integrity (RI)
  - Define triggers and check constraints
- Identify impacts and advantages of referential integrity, including impacts of delete rules
- Identify considerations when using triggers and check constraints
- Identify the advantages of views

Notes:

- Database administrators normally create objects. Still, you should know something about the procedure so that you can discuss problems with your database administrator. Your employer will probably let you have a test or personal environment so that you can develop your own databases or do testing.

- Triggers and constraint checking can be used to reduce the number of business rule validations that must be coded in applications. The data and business rules are defined to the database manager. Thus, they are centralized, so that your applications do not have to perform these tasks.
2.1 Creating Objects
Notes:

- Bill has a small store. He wants to implement a data model to administer his orders. The design was done by a consulting company. Bill needs the following tables:

  - An ORDER table with the following columns: ORDER_NO, ORDER_DATE, CUST_NO, and ORDER_REF. Each order, that is, row of the table, will contain an order number (ORDER_NO), the date when the order was placed (ORDER_DATE), and a customer number (CUST_NO), but it need not have a reference to another order (ORDER_REF).

  - An ORDER_ITEM table with the following columns: ORDER_NO, ART_NO, and ORDER_QTY. Each row of the table represents an ordered item and must contain the order number (ORDER_NO) of the associated order, the article number (ART_NO) of the ordered article, and the quantity ordered (ORDER_QTY).

  - An ARTICLE table with the following columns: ART_NO, ART_NAME, and STOCK_QTY. Each row of the table represents an article sold by Bill's company and must contain the article number for the article (ART_NO), the name for the article (ART_NAME), and the number of items in stock (STOCK_QTY).
Bill asks Susan, a newly-hired consultant and database administrator, for help in implementing the model in DB2.

**Bill:** “Susan, I need your help to implement a small data model.”

**Susan:** “Sure, what would you like?”

**Bill:** “I need to create a persistent data store for orders, articles and order_items, as I earlier described to you.

**Susan:** “That is easy. I will just create the tables for you....”
Creating Tables and Adding Columns

CREATE TABLE ORDER
  (ORDER_NO INTEGER NOT NULL,
   ORDER_DATE DATE NOT NULL,
   CUST_NO SMALLINT NOT NULL)

ALTER TABLE ORDER
  ADD ORDER_REF INTEGER

CREATE TABLE ARTICLE
  (ART_NO INTEGER NOT NULL,
   ART_NAME CHAR(20) NOT NULL,
   STOCK_QTY INTEGER NOT NULL)
  IN DATABASE PRODDB -- OS/390 only

Notes:

- Platform-dependent is the syntax for creating a table. In a DB2 UDB for UNIX, Windows and OS/2 environment, you can code CREATE TABLE.....IN tablespace-name. This identifies the table space in which the table will be created. If the IN clause is not specified, then the table is created in one of three possible default table spaces, depending on which of them exists. If none of the default table spaces exist, the CREATE fails.

- In an z/OS or OS/390 environment, there are three valid approaches for creating a table:
  - CREATE TABLE ... IN database-name.tablespace-name. This approach puts the table in a specified table space of a specified database.
  - CREATE TABLE ... IN DATABASE database-name. This approach puts the table in a specified database and creates a simple table space within that database to hold the table.
- CREATE TABLE .... This approach puts the table in the default database, DSNDB04, and creates a simple table space within that database to hold the table.

- Data types BLOB, CLOB, and DBCLOB are available in DB2 UDB for OS/390 Versions 6 as well as on DB2 UDB for UNIX, Windows and OS/2 and DB2 UDB for AS/400.

- Generally, you specify the columns of the table when you create the table. If you need to add a column to the table later, you can use the ALTER TABLE statement with the ADD option. The new column must be defined as nullable or with NOT NULL WITH DEFAULT since the table may already contain rows at that point in time. If the column was defined with NOT NULL, DB2 will not know which value to assign to the column for the existing rows. If the column is defined as nullable, DB2 assigns a value of null to the column for existing rows; if it is defined with NOT NULL WITH DEFAULT, the appropriate default value (system-defined or user-defined) is assigned to the column for existing rows.

- The first example creates a table named ORDER containing the three columns ORDER_NO, ORDER_DATE, and CUST_NO. The fourth column (ORDER_REF) for the table has been forgotten and is added later on.

  If created in DB2 UDB for UNIX, Windows and OS/2, the table will go into one of three possible default table spaces, depending on which of these table spaces exist. If created in DB2 for z/OS or OS/390, a simple table space will be created in the default database, DSNDB04, and the table will be put there.

- The second example adds the forgotten column ORDER_REF to the ORDER table. Fortunately, ORDER_REF need not always have a value so that it can be defined as nullable. If ORDER_REF always required a value, NOT NULL WITH DEFAULT would have to be used.

- The third example creates a table named ARTICLE containing three columns. This example could only be executed in DB2 for OS/390 because the IN DATABASE clause is not supported in DB2 UDB for UNIX, Windows and OS/2. DB2 will create a simple table space in the PRODDB database and store the ARTICLE table there.

Note: In general you cannot drop a column, change a column’s name, change a column’s data type, or change a column’s null attribute with ALTER TABLE. You have to drop the whole table and redefine it. Since dropping the table results in the immediate drop of all dependent objects, including indexes, views, privileges, etc., it is very risky to drop an object unless you have stored the SQL necessary to recreate the dependent objects somewhere outside the DB2 catalog. Otherwise, it will be very difficult to re-create the dependent objects. In DB2 UDB for UNIX, Windows, OS/2, OS/390 and AS/400 you can rename a table using RENAME TABLE and you can change the length of a VARCHAR-column using ALTER TABLE. In DB2 UDB AS/400, you can even DROP a Column of a table, refer to SQL Reference for DB2 UDB AS/400.
Creating Tables (Cont.)

CREATE TABLE NEW_ORDER
LIKE OLD_ORDER;

CREATE TABLE NEW_STAFF
AS
(SELECT COL, COL2, COL5 FROM STAFF)
DEFINITION ONLY;

Notes:

- This visual shows alternative syntax for creating a new table by taking the definitions from an existing table.
- CREATE TABLE ....LIKE creates a new table with the same column definitions (column name, data type and null attribute) as the existing table.
  - No unique constraints, foreign key constraints, triggers, or indexes are created.
  - The table is not populated.
  - This syntax is available in DB2 UDB z/OS and OS/390 and DB2 UDB for UNIX, Windows and OS/2. It is not available in DB2 UDB for OS/400 V4 R4.
  - CREATE TABLE --- DEFINITION ONLY uses a query to define a new table.
  - The table is not populated using the result of query and the REFRESH TABLE statement cannot be used.
  - When the CREATE TABLE statement is completed, the table is no longer considered a summary table. (We will cover summary tables briefly in a later unit.)
- The columns of the table are defined based on the definitions of the columns that result from the fullselect. If the fullselect references a single table in the FROM clause, select list items that are columns of that table are defined using the column name, data type, nullability characteristic and column default value of the referenced table.

- CREATE TABLE... DEFINITION ONLY is available on DB2 UDB for UNIX, Windows and OS/2.
**Scenario (Part 2)**

![Image of two people, one asking how to limit the ordered quantity to not exceed 100, and the other responding with Check Constraints.]

**Notes:**

Bill is concerned. His warehouse is quite small and he does not want one customer to order a quantity that will reduce his stock so other customer orders can not be handled.

**Bill:** "I am pleased with tables you created, but I have some special requirements. As you know, our warehouse is quite small. I want to ensure that we never accept an order for more than 100 of any article regardless of its article number. Since articles with an article number of 50000 and higher are very large, I want to make sure that we never accept an order for more than 10 of these items.

**Susan:** "That is easy. I will just add some Check Constraints to the tables for you...."

**Bill:** "And Susan, please make sure that there are no duplicate values in the key elements of the table."

**Susan:** "OK, no problem."
Check Constraints - The Need

DATA RULE:  No value in the ORDER_QTY column in the table ORDER_ITEM may exceed 100.

Notes:

- Many data rules are specific to the column values in a single table. Check constraints can be used to define such rules to the DB2 database manager. The database manager will then enforce these rules.
- Constraints will be checked during INSERT and UPDATE operations.
- Installations requiring such data checking should consider using constraints rather than enforcing data content via applications. Constraints are defined once, directly within DB2, and are applied globally, eliminating the need for applications to do this work.
- The left side of the graphic shows applications that contain rules coded within the applications themselves. Unless the code for these rules is made common to the applications via modularization or copy books, there is a risk that, over time, the rules in the applications will start to diverge and function differently and, therefore, inconsistently, which could cause serious problems. The right side of the graphic shows a situation where the rules are in DB2 itself, rather than in the applications. Since there is only one copy of each rule, there is no possibility of rule inconsistencies emerging over time. The absence of rules makes the applications smaller, which makes them easier to code and test.
Check Constraints

CREATE TABLE ORDER_ITEM
  (ORDER_NO INTEGER NOT NULL,
   ART_NO INTEGER NOT NULL,
   ORDER_QTY SMALLINT NOT NULL
   CONSTRAINT order_limit
   CHECK(ORDER_QTY <= 100),
   CONSTRAINT special_order
   CHECK(ART_NO <= 50000 OR
   ORDER_QTY <= 10) )

ALTER TABLE ORDER_ITEM
  DROP CONSTRAINT special_order

ALTER TABLE ORDER_ITEM
  ADD CONSTRAINT special_order
  CHECK(ART_NO <= 50000 OR
  ORDER_QTY <= 10)

Notes:

- Single-column constraints can be defined at the column level or the table level. Multi-column constraints must be defined at the table level.
- Check constraints can be defined on a new table with the CREATE TABLE statement.
- Check constraints can be added to an existing table with the ALTER TABLE statement.
- Constraints may include the basic WHERE clause constructs:
  - Basic comparisons (> , < , = , >= , etc.)
  - BETWEEN
  - LIKE
  - IN
- Constraints may NOT include:
  - Subqueries
  - Column functions
  - Special registers (such as CURRENT DATE)
  - On OS/390, the NOT logical operator
See the *SQL Reference* for your platform for full details.

- The constraint can be explicitly named when it is defined. If it is not named, DB2 will create a name. Explicit naming is also possible on the *ALTER TABLE* statement by providing the `CONSTRAINT` clause before the `CHECK` specification, just as in the example above.

- Constraints can also be removed by means of the *ALTER TABLE* statement as illustrated in the above visual.

- The example implements the constraints that Bill requested:
  
  - In the *CREATE TABLE* statement, the order quantity is constrained at the column level so that it never exceeds 100, regardless of the article number. At the table level, a second constraint named 'special_order' ensures that the order quantity does not exceed 10 if the article number is greater than or equal to 50000: it checks the row and makes sure the article number is less than or equal to 50000 or, if it is not, verifies that the order quantity is less than or equal to 10.

  - The constraint implemented on the table level in the *CREATE TABLE* statement is not quite what Bill had in mind since it would accept orders of more than 10 items for the article with article number 50000. To remedy the problem, we need to drop constraint 'special_order' and add it again by means of the *ALTER TABLE* statement.
Scenario (Part 3)

Notes:

Bill: “Susan, I need your help again. I have another requirement for my new DB2 system.”

Susan: “No problem. What is up?”

Bill: “As I tested my system, I encountered some problems. There are order items in the ORDER_ITEM table whose order numbers are not in the ORDER table and whose article numbers are not in the ARTICLE table. Also, there are orders in the order table that refer to orders that do not exist. I am not sure how this happened, but we have to prevent it from happening in the production system.”

Susan: “Sure, we have to implement referential integrity. What relationships do you have between your tables?”

Bill: “When creating a new row in the ORDER_ITEM table, the order number should already exist in the ORDER table and the article number should already exist in the ARTICLE table.”
"If I delete an order in the ORDER table, all rows in the ORDER_ITEM table that refer to that order should be deleted automatically.

"It should not be possible to delete a row in the ARTICLE table if the ORDER_ITEM table has rows which refer to this article.

"When creating a new row in the ORDER table, the ORDER_REF column may refer to another order of the ORDER table. This order must already exist."

Susan: "OK, no problem."
Create Index and Primary Key

CREATE UNIQUE INDEX ORDER_IND
ON ORDER (ORDER_NO)

ALTER TABLE ORDER
ADD PRIMARY KEY (ORDER_NO)

Notes:

- If you are searching for names in a card file and you have no index, you have to read the complete card file. DB2 must also read (scan) a complete table if no index is specified on a column used in a predicate of a WHERE clause.
- An index is an ordered collection of the key values in the column on which the index is defined. Each key value has a RID (Row IDentifier) or RIDs that points to the actual data row(s) which contain the key value.
- A table can have many indexes. Each index can span one column or several columns. Each index can be unique or nonunique.
- The usefulness of an index depends on its key. Columns that you use frequently in WHERE clauses, joins, grouping and ordering operations are good candidates for indexes.
- DB2 indexes allow duplicate values by default. If you want to prevent duplicate values in a key column, use CREATE UNIQUE INDEX.
- The example creates a unique index called ORDER_IND on the ORDER_NO column of the ORDER table. This satisfies one of Bill's requirements.
Referential Integrity

Notes:

- The above example shows the relationships between the ORDER table and the ORDER_ITEM table on one hand and between the ARTICLE table and the ORDER_ITEM table on the other hand.
- The ORDER table has a primary key on the ORDER_NO column. The values in this column must be unique and not null.
- The ARTICLE table has a primary key on the ART_NO column. The values in this column must be unique and not null.
- The ORDER_ITEM table has a primary key on the combination of the ORDER_NO and ART_NO columns. The values in the primary key must be unique and not null. In other words, the combination of ORDER_NO and ART_NO must be unique and not null in each row of the table.
- The ORDER and ARTICLE tables are parent tables. For both parent tables, the ORDER_ITEM table is the dependent table. ORDER_NO in the ORDER_ITEM table is a foreign key, which means its value must exist in the ORDER table's primary key when
the ORDER_ITEM row is created. ART_NO in the ORDER_TABLE is also a foreign key, which means its value must exist in the ARTICLE table’s primary key when the ORDER_ITEM row is created.

- DB2 permits one exception to the rule that a foreign key value must match a value in the corresponding primary/unique key: If desired by the user, a foreign key may also be null, even though a primary/unique key will never contain a null. In this example, the foreign keys are both part of a primary key so they will not be allowed to contain nulls.
Referential Integrity - Keys

- A PRIMARY KEY value must be UNIQUE and cannot be NULL

- By definition, a FOREIGN KEY matches a PRIMARY KEY of a UNIQUE KEY or can be null

- DB2 enforcement of referential constraints occurs during insert/update/load of FOREIGN KEY

- DB2 ensures that a PRIMARY KEY/UNIQUE KEY is updated only if no FOREIGN KEY matches the original value of the primary/unique key

Notes:

- A unique key is a key (set of columns) that is constraint so that no two values are equal. The columns of a unique key cannot contain null values.

- A primary key is a special case of a unique key. It is normally used to refer to the rows of the appropriate table.

- A table can have multiple unique keys, but cannot have more than one primary key. A key can either be defined as unique key or as primary key, but not both. In DB2 UDB for UNIX, Windows and OS/2 and DB2 UDB for AS/400, CREATE TABLE and ALTER TABLE can be used to identify a key as unique key. In DB2 for OS/390, a key can only be defined as unique key by means of the CREATE TABLE statement.

For both DB2 UDB for OS/390, DB2 UDB for UNIX, Windows and OS/2, and DB2 UDB for AS/400 primary keys can be defined using CREATE TABLE or ALTER TABLE.

- By definition, the values of a foreign key always match the values of a primary key or unique key.
• A referential constraint is a rule, enforced by the database manager, that controls the relationship between a primary key/unique key and a foreign key. Referential constraints are defined to DB2 when the foreign key is defined. They are enforced when rows containing a foreign-key value are inserted, updated, or loaded.

• DB2 prevents the updating of primary key/unique key values if there are matching foreign key values. The new values are not propagated to the appropriate rows of the dependent table. In other words, a primary key/unique key value can only be updated if no foreign key value matches the original value of the primary/unique key.
Referential Integrity - Delete Rules

**Problem:**

When deleting a primary/unique key, what should be done to rows that contain matching foreign keys?

**DELETE Rules:**

- **CASCADE** - Delete the rows with the matching values in the dependent table
- **SET NULL** - Change the matching values in the dependent table to null
- **RESTRICT or NO ACTION** - Disallow the DELETE in the parent table if matching values exist in the dependent table

**Notes:**

- The delete rules are defined in the REFERENCES clause of the foreign key definition in the CREATE TABLE or ALTER TABLE statement.
- The only difference between NO ACTION and RESTRICT is when the referential constraint is enforced. RESTRICT enforces the rule immediately and NO ACTION enforces the rule at the end of the statement.
Alter Table - Adding Referential Constraints

```
ALTER TABLE ORDER
ADD PRIMARY KEY (ORDER_NO)

ALTER TABLE ARTICLE
ADD PRIMARY KEY (ART_NO)

ALTER TABLE ORDER_ITEM
ADD PRIMARY KEY (ORDER_NO, ART_NO)

ALTER TABLE ORDER_ITEM
ADD CONSTRAINT IN ORDER
FOREIGN KEY (ORDER_NO)
REFERENCES ORDER
ON DELETE CASCADE

ALTER TABLE ORDER_ITEM
ADD CONSTRAINT OF ARTICLE
FOREIGN KEY (ART_NO)
REFERENCES ARTICLE
ON DELETE RESTRICT
```

Figure 2-13. Alter Table - Adding Referential Constraints

Notes:

- The examples illustrate how to implement the relationships between the ORDER table and the ORDER_ITEM table and between the ARTICLE table and the ORDER_ITEM table.

- DB2 for OS/390 requires that a unique index has been defined for the columns for the primary key before the primary key can be added by means of the ALTER TABLE statement. In DB2 UDB for UNIX, Windows and OS/2 and DB2 UDB for AS/400, if not provided, the unique index is defined automatically.

- ALTER TABLE can also be used to add or drop check constraints and to add a column to an existing table as we have seen before.
Referential Integrity (Part 2)

<table>
<thead>
<tr>
<th>ORDER_NO</th>
<th>ORDER_DATE</th>
<th>ORDER_QTY</th>
<th>ORDER_REF</th>
</tr>
</thead>
<tbody>
<tr>
<td>22333</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22444</td>
<td></td>
<td>22333</td>
<td></td>
</tr>
<tr>
<td>35555</td>
<td></td>
<td>22444</td>
<td></td>
</tr>
</tbody>
</table>

Delete rule for self-referencing constraint can only be
CASCADING or NO ACTION

```
ALTER TABLE ORDER
ADD FOREIGN KEY (ORDER_REF)
    REFERENCES ORDER
    ON DELETE NO ACTION
```

Notes:

- The primary/unique key does not necessarily have to be in a table other than the one for
  the foreign key. It can be in the same table. The non-null values in column ORDER_REF
  of the ORDER table must have a counterpart in the ORDER_NO column of the ORDER table.
  Thus, ORDER_REF represents a foreign key in the same table as the associated primary key.

- If the foreign key is in the same table as the corresponding primary key, the delete rule
  for the referential constraint must be CASCADING or NO ACTION. It cannot be SET
  NULL or RESTRICT. This may have an impact on the programs for the application.

- In the example above, order 22444 refers to order 22333 and order 35555 to order
  22444. The delete rule chosen is NO ACTION which, for example, prevents the deletion
  of order 22333 because order 22444 depends on it. Since order 35555 is not
  dependent on any other order, it can be deleted.

  If the delete rule was CASCADING, the deletion of order 22333 would cause the deletion
  of order 22444 which, in turn, would cause the deletion of order 35555.

- There also exist restrictions for other, more complex, referential structures. However,
  their discussion is outside the scope of this course.
Scenario (Part 4)

Notes:

Bill: “Susan, I am very happy. I created a table by myself. It is a table for orders to my suppliers. The name of the table is SUP_ORDERS. Everything works fine except for a last problem.”

Susan: “How can I help?”

Bill: “I want DB2 to automatically insert a row in the SUP_ORDERS table if the stock quantity of an article in the ARTICLE table decreases to less than 50. I want the new row in the SUP_ORDERS table to contain the article number and the current stock on that article. Is that possible?”

Susan: “Sure, we can use a trigger…”

Bill: “That is great.”
Triggers - The Need

BUSINESS RULE: When value in the ORDER_QTY column gets below 50, then insert reorder information into table SUP_ORDERS.

BUSINESS RULES:

INSERT, UPDATE, and DELETE actions against a table may require another action or set of actions to occur to meet the needs of the business.

Notes:

A trigger is a set of actions that will be executed when a defined event occurs. The triggering events can be the following SQL statements.

- INSERT
- UPDATE
- DELETE

Triggers are defined for a specific table and once defined, a trigger is automatically active. Trigger definitions are stored in the system catalog tables.

Triggers can be defined to DB2 so that the database manager automatically enforces business rules that should follow the triggering event operation against the specified table. There are many benefits when you let DB2 enforce such rules:

- You eliminate the need to code programs to perform this task.
- Rules are defined once and enforced globally.
- If business requirements change, you can easily change the rules.
Some of the uses of a trigger include:

- Data Validation - ensures that a new data value is within the proper range. This is similar to table-check constraints, but is a more flexible data validation mechanism.

- Data Conditioning - implemented using triggers that fire before data record modification. This allows the new data value to be modified or conditioned to a predefined value.

- Data Integrity - can be used to ensure that cross-table dependencies are maintained. The triggered action could involve updating data records in related tables. This is similar to referential integrity, but is a more flexible alternative.
Create Trigger

CREATE TRIGGER re_order
    AFTER UPDATE OF STOCK_QTY ON ARTICLE
    REFERENCING NEW AS N OLD AS O
    FOR EACH ROW
    MODE DB2SQL
    WHEN (N.STOCK_QTY < 50 AND O.STOCK_QTY >= 50)
    INSERT INTO SUP_ORDERS
    VALUES (N.ART_NO, N.STOCK_QTY)

- Triggered action can contain INSERT, searched UPDATE or DELETE, and single row SELECT
- SQL extensions, such as CASE expressions, can be exploited

DROP TRIGGER re_order

Notes:

- Triggers can be defined with the CREATE TRIGGER statement.
- In the example:
  - AFTER UPDATE OF STOCK_QTY ON ARTICLE indicates that a triggered action should take place after the STOCK_QTY column in has been updated in the ARTICLE table. The triggered action can be made to occur either before or after an event. The event can be an INSERT, an UPDATE of a specific column or columns, an UPDATE of any column of a table, or a DELETE.
  - REFERENCING is used to correlate the event with the triggered action. In this case, we want the row that is being added to the SUP_ORDERS table in the triggered action to contain the values in the ART_NO and STOCK_QTY columns, but we want these to be the values after the update of the ARTICLE table. In other words, we want the new values, not the old values.

We use the REFERENCING clause to designate the new values with a correlation name of N. We also use the REFERENCING clause to designate the old values with
a correlation name of O. We want to ensure that the trigger is fired only if the old
stock quantity is greater than or equal to 50 and the new stock quantity is less than
50.

These variables defined by the REFERENCING clause are called transition
variables.

- FOR EACH ROW indicates that the trigger should fire for each row inserted. FOR
  EACH STATEMENT, which causes the trigger to fire once per triggering statement,
is another option. The difference between these two options should be obvious if
you consider an INSERT that adds more than one row to a table:
  — If FOR EACH ROW is used, the triggered action will take place once for each
    row updated.
  — If FOR EACH STATEMENT is used, the triggered action will be performed only
    once, no matter how many rows are updated.

- MODE DB2SQL identifies the mode of the trigger. DB2SQL is the only valid mode at
  present.

- WHEN restricts the triggered action. The action will only be executed if the condition
  of the WHEN evaluates to TRUE. It is important that the triggered action occur only
  the first time the stock quantity goes below 50. If the WHEN condition was
  'N.STOCK._QTY < 50', the trigger would be fired every time the value went below
  50, even if it had previously been below 50. Each time the trigger was fired, we
  would place an additional order for the product and would soon be flooded with new
  orders of the product.

- INSERT INTO SUP_ORDERS... is the triggered action. Note that it takes advantage
  of the correlation name established with the REFERENCING clause.

  The triggered action can be more complex than the example shown.

  • Triggers can be removed with the DROP TRIGGER statement. Any packages that have
    a dependency on the trigger will be marked invalid.

  • Consult the SQL Reference for more details about the variations possible within a
    CREATE TRIGGER statement.
Create Trigger (Cont.)

```
CREATE TRIGGER newproj
  AFTER INSERT ON PROJECT
 REFERENCING NEW AS N_ROW
  FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
  UPDATE DEPARTMENT SET
  PROJ_COUNT = PROJ_COUNT+1
  WHERE DEPTNO=N_ROW.RESP_DEPT;
  CALL
  MY_PROC(N_ROW.RESP_DEPT,N_ROW.PROJNAME);
  UPDATE EMPLOYEE SET PROJ_COUNT=PROJ_COUNT+1
  WHERE EMPNO=N_ROW.RESP_EMP;
END

AS/400: A trigger is created with the
ADDPFTRG (Add Physical File Trigger) CL command.
```

Notes:

- This visual shows how to create a trigger which performs several SQL-statements.
- The significance of the required ATOMIC keyword is that the set of SQL statements is treated as an atomic unit. That is, either all of the statements are executed or none. If, for example, the second UPDATE statement shown in the visual fails, all changes made to the database as part of the triggering operation are backed out.
- REFERENCING is used to correlate the event with the triggered action. In both row and statement triggers it might be necessary to refer to the whole set of affected rows. For example, triggered SQL statements might need to apply aggregations over the set of affected rows (MIN, MAX, or AVG of some column values). A trigger can refer to the set of affected rows by using transition tables that can be specified in the REFERENCES clause of the CREATE TRIGGER statement. The columns in transition tables are referred to using the column names of the triggering table. As with transition variables, there are two kind of transition tables, which are specified as OLD_TABLE and NEW_TABLE together with a table-name. Example:
CREATE TRIGGER LARG_ORD
   AFTER INSERT ON INVOICE
   REFERENCING NEW_TABLE AS N_TABLE
   FOR EACH STATEMENT MODE DB2SQL
   SELECT LARGE_ORDER_ALERT(CUST_NO, TOTAL_PRICE, DELIVERY_DATE)
       FROM N_TABLE WHERE TOTAL_PRICE > 20000

This example shows the use of a transition table in a statement trigger. The UDF named LARGE_ORDER_ALERT will be invoked for each row in the transition table that corresponds to an order worth more than 20000. This example also shows the use of a SELECT statement to invoke a user-defined function for each row in a transition table.

A new transition table always has the full set of updated rows, even if referenced in an after row trigger. That is, all updated or inserted rows are included, even when an after row trigger is activated for the first row.

Transition tables are read-only.
View Considerations

Figure 2-19. View Considerations

Notes:
A view is a named specification of a result table. The specification is an SQL SELECT statement that is effectively executed whenever the view is referenced in an SQL statement.

The data needed to satisfy queries is often stored in many large tables. However, we often need only a small subset of this data in order to create reports. A view can act like a filter so that a report contains only the information we need, rather than a lot of extra detail.

A view can be used to:

- Make a subset of a table’s data available to its users. For example, a view based on an employee table might contain only some of the rows and columns of the table, such as employee number, last name, birth-date, and salary of certain employees in departments A00 and C01.

- Show data that was obtained by joining two or more tables as if it were a single table. A view that is based on a join cannot be updated.
- Display data with different column names and sequences than those which appear in the base tables.

- Display derived data, including the results of any function or operation that you can use in a SELECT statement. For example, a view can be written to display a list of the departments with the average salary of the people in that department without showing the details of the employees in that department.

- Control access to data. Usually, the user will be granted access only to the view. No access to the underlying table(s) is granted in most cases. This has the effect of hiding the other columns and rows from those who do not need to see them.

**Note:** A view does not contain any data. It is simply a *logical table* or *virtual table* that points to rows in real tables. As a result, the data in the view is always current. There is no need to do anything to keep the view synchronized with the table and no extra storage space is needed for the rows in the view.
Sample View

Notes:

- Assume a user needs the last names of the employees in the EMPLOYEE table shown on the visual and, for each employee, also the name of his/her department. However, the user should not be able to see the salaries of the employees. The department name must be obtained by joining to the DEPARTMENT table.

- When a view is created, a name is assigned to the view. The name has the same format as table names. It can be used to refer to the view in other SQL statements.

- The column list, which appears in parentheses after the view name, can be used to give the columns from the base tables different names in the view or to name calculated columns.

- The user of a view needs only the authority to access the view. Access to the original table is not needed. It is possible to give a user SELECT authority on a view without granting any authority on the base table.
The CREATE VIEW statement shown on the visual describes the data that should be made available. It does not make the data available. To see the data described by the view, the statement

```
SELECT * FROM EMP_DEPT
```

can be used which refers to the view rather than to the base tables. It displays the full contents of the view. The join result is now treated as if it were a single table. There is no need to specify the join condition, control the columns or rows returned, or do any other SQL clause except ORDER BY, because the view definition established all of the conditions that were needed.
Notes:

- Views can be based on one or more tables.
- Views can also be based on other views or views joined with tables. The views upon which a view is based must have previously been defined.
- In the example, EMPLOYEE, DEPARTMENT, and PROJECT are tables. PROJ_DEPT is a view based on an inner join of tables PROJECT and DEPARTMENT. It could be created by the following CREATE VIEW statement:

```
CREATE VIEW PROJ_DEPT
    AS SELECT PROJNAME, DEPTNAME, MGRNO
    FROM PROJECT P INNER JOIN DEPARTMENT D
    ON P.DEPTNO = D.DEPTNO
```
PROJ_DEPT_MGR is a view that is based on a left outer join of view PROJ_DEPT to table EMPLOYEE. Its definition could be as follows:

```sql
CREATE VIEW PROJ_DEPT_MGR (PROJECT, DEPARTMENT, MANAGER)
    AS SELECT PROJNAME, DEPTNAME, LASTNAME
    FROM PROJ_DEPT LEFT OUTER JOIN EMPLOYEES
    ON MGRNO = EMPNO
```

- The tables used in the above visual are pure subsets of the DB2 sample tables. Instead of the tables on the visual, views on the DB2 sample tables could be used increasing the number of levels in the view hierarchy.
## Read-Only Views

The EMPLOYEE table contains information about employees, and the DEPARTMENT table contains information about departments. A view can be created by querying these tables. The EMP_DEPT view is a read-only view that joins the EMPLOYEE and DEPARTMENT tables.

**EMPLOYEE Table**

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>LASTNAME</th>
<th>WORKDEPT</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>000010</td>
<td>HAAS</td>
<td>A00</td>
<td>52750.00</td>
</tr>
<tr>
<td>000030</td>
<td>Kwan</td>
<td>C01</td>
<td>38250.00</td>
</tr>
<tr>
<td>000120</td>
<td>O'Connell</td>
<td>A00</td>
<td>29250.00</td>
</tr>
<tr>
<td>000130</td>
<td>Quintana</td>
<td>C01</td>
<td>23800.00</td>
</tr>
<tr>
<td>000140</td>
<td>Nicholls</td>
<td>C01</td>
<td>28420.00</td>
</tr>
</tbody>
</table>

**DEPARTMENT Table**

<table>
<thead>
<tr>
<th>DEPTNO</th>
<th>DEPTNAME</th>
<th>MGRNO</th>
</tr>
</thead>
<tbody>
<tr>
<td>A00</td>
<td>SPIFFY COMPUTER SERVICE DIV.</td>
<td>000010</td>
</tr>
<tr>
<td>C01</td>
<td>INFORMATION CENTER</td>
<td>000030</td>
</tr>
<tr>
<td>D01</td>
<td>DEVELOPMENT CENTER</td>
<td></td>
</tr>
</tbody>
</table>

The EMP_DEPT view contains information about employees and their departments.

**EMP_DEPT View**

<table>
<thead>
<tr>
<th>NAME</th>
<th>DEPARTMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>HAAS</td>
<td>SPIFFY COMPUTER SERVICE DIV.</td>
</tr>
<tr>
<td>O'Connell</td>
<td>SPIFFY COMPUTER SERVICE DIV.</td>
</tr>
<tr>
<td>Kwan</td>
<td>INFORMATION CENTER</td>
</tr>
<tr>
<td>Nicholls</td>
<td>INFORMATION CENTER</td>
</tr>
<tr>
<td>Quintana</td>
<td>INFORMATION CENTER</td>
</tr>
</tbody>
</table>

### Notes:

- A view is read-only if any of the following statements is true:
  - The first FROM clause identifies more than one table or view, that is, the view joins multiple tables or views.
  - The first SELECT clause specifies the keyword DISTINCT, that is, the view eliminates duplicate rows (to be precise: all but one of each set of duplicate rows).
  - The first SELECT clause contains a column function, that is, a column of the logical table described by the view is derived by means of a column function.
  - The outer subselect contains a GROUP BY clause, that is, the rows of the logical table described by the view may be derived from multiple rows of the base tables or views rather than from a single row.
  - The view contains a subquery such that the base object of the outer subselect, and of the subquery, is the same table. This means that the table or view from which the view extracts data is also used to determine the rows being selected.
- The first FROM clause identifies a read-only view, that is, the view is based on a read-only view.

- A read-only view cannot be the object of an INSERT, UPDATE, or DELETE statement.

- The UPDATE statement in the example fails because view EMP_DEPT is based on a join of tables EMPLOYEE and DEPARTMENT and, thus, is read-only.
Notes:

- Not all views are read-only. Views that are not read-only may have some restrictions on which columns of the view may be updated. For example, a view that contains derived data like a person's age cannot be used to update the derived data.

- In addition:
  - You must have the appropriate authorization to insert, update, or delete rows using a view.
  - A view that you can use to update data is subject to the referential constraints and table check constraints for the table upon which the view is ultimately based.
  - When you use a view to insert a row in a table, the view definition must specify all the columns in the base table that are not nullable and do not have a default value. The row being inserted must contain a value for each of those columns.
  - When you use a view to update, you can only update columns that are in the view. The columns of the view form a projection of the base table. You cannot update derived columns, even if they are in the view.
- When you delete with a view, you are deleting one or more entire base table rows (the selection of rows from the base table), and not just the part of the table that is visible through the view. Make sure that the data which is not visible through the view is data that you want to delete before you delete through a view.
**Check Option**

### EMPLOYEE Table

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>LASTNAME</th>
<th>WORKDEPT</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>000010</td>
<td>HAAS</td>
<td>A00</td>
<td>52750.00</td>
</tr>
<tr>
<td>000030</td>
<td>KWAN</td>
<td>C01</td>
<td>38250.00</td>
</tr>
<tr>
<td>000120</td>
<td>O'CONNELL</td>
<td>A00</td>
<td>29250.00</td>
</tr>
<tr>
<td>000130</td>
<td>QUINTANA</td>
<td>C01</td>
<td>23800.00</td>
</tr>
<tr>
<td>000140</td>
<td>NICHOLLS</td>
<td>C01</td>
<td>28420.00</td>
</tr>
</tbody>
</table>

**CREATE VIEW SALARIES_1**

AS

SELECT EMPNO, LASTNAME, SALARY
FROM EMPLOYEE
WHERE SALARY < 40000

**UPDATE SALARIES_1**

SET SALARY = 42000
WHERE EMPNO = '000030'

**CREATE VIEW SALARIES_2**

AS

SELECT EMPNO, LASTNAME, SALARY
FROM EMPLOYEE
WHERE SALARY < 40000

WITH CHECK OPTION

**UPDATE SALARIES_2**

SET SALARY = 42000
WHERE EMPNO = '000030'

**WORKS!**

**FAILS!**

### Notes:

- **WITH CHECK OPTION** (which is identical to WITH CASCADED CHECK OPTION) specifies that every row which is inserted or updated via the view must conform to the view definition. If the row being inserted or updated via the view does not conform to the view definition, the insert or update fails and no rows are inserted or updated.

- If the view is based on another view which has a search condition, this search condition is enforced as well. If that other view is again based on a third view, the search condition of the third view is enforced as well, and so on. With other words, both the search conditions of the view itself and of the views upon which the view is based, directly or indirectly, are enforced.

- **WITH CHECK OPTION** must not be specified if the view is read-only or its search condition includes a subquery, or the search condition of an underlying view includes a subquery.

- If **WITH CHECK OPTION** is specified for an updateable view that does not allow inserts, the constraint applies only to updates.
• WITH CHECK OPTION can even be specified if the view is updateable, but does not have a search condition. In this case, the search conditions of those views are enforced upon which the view is directly or indirectly based.

• If WITH CHECK OPTION is omitted, the search condition of the view is not used in the checking of insert or update operations. The view can then be used to insert rows that do not conform to the search condition of the view and to update rows so that they no longer conform to that search condition. Such rows cannot be retrieved using that view.

If the view depends (directly or indirectly) on a view that has been defined with WITH CHECK OPTION, the search conditions of that view and of all views upon which, in turn, that view depends are still enforced.

• In the first example, view SALARIES_1 is created without CHECK OPTION. An update that raised the yearly salary to 40000 or more, which is beyond the scope of the view, would succeed. After the update, the changed row is no longer visible through the view, but still exists in the base table.

• In the second example, view SALARIES_2 is created. It has the same definition as SALARIES_1 except that CHECK OPTION is specified. This time, an update that tried to raise the yearly salary to 40000 or higher would fail. CHECK OPTION ensures that the update is not permitted if it would remove the changed row from the scope of the view.

• The CHECK OPTION has two forms: cascaded and local. The difference between these two forms is meaningful only when a view is defined on top of another view. If a view called VIEW1 is defined by a query on another view called VIEW2, we will refer to VIEW2 as the underlying view. When VIEW1 is defined with a local check option, operations on VIEW1 must satisfy the definitions of VIEW1 and of all underlaying views that also have a check option; however, they need not satisfy the definitions of underlaying views that do not have a check option. On the other hand, when VIEW1 is defined with cascaded check option, all operations on VIEW1 my satisfy the definitions of VIEW1 and of all underlaying views, whether they have a check option or not. If a CREATE VIEW statement simply specifies WITH CHECK OPTION, the default is cascaded check option.
View Merge

```
CREATE VIEW EMP_SALARY_DEPT (NAME, SALARY, DEPARTMENT) 
  AS SELECT LASTNAME, SALARY, DEPTNAME 
  FROM EMPLOYEE INNER JOIN DEPARTMENT 
  ON WORKDEPT = DEPTNO 
  WHERE DEPTNO = 'C01'
```

```
SELECT NAME, SALARY 
  FROM EMP_SALARY_DEPT 
  WHERE SALARY > 25000
```

```
FUNCTIONS AS
```

```
SELECT LASTNAME AS NAME, SALARY 
  FROM EMPLOYEE INNER JOIN DEPARTMENT 
  ON WORKDEPT = DEPTNO 
  WHERE DEPTNO = 'C01' AND 
  SALARY > 25000
```

Notes:

- In the view merge process, the statement that references the view is combined with the subselect that defined the view. This combination creates a logically equivalent statement. The equivalent statement is executed against the table upon which the view is based.

- In the example, the following happens:
  - The FROM clause of the view definition becomes the FROM clause of the combined statement.
  - The WHERE clause of the defining subselect, WHERE DEPTNO = 'C01', is combined with the WHERE clause of the SELECT, WHERE SALARY > 25000, to give the logically equivalent WHERE DEPTNO = 'C01' AND SALARY > 25000.
  - The column name NAME of the SELECT statement becomes the column name of an AS clause redefining the name of the appropriate base table column: LASTNAME AS NAME.
View Materialization

In some cases, a view cannot be merged with a SELECT statement because the two cannot be combined into a single statement. In these cases, the view is materialized, that is, written to a work file on disk to be processed like a table.

In the example, the view applies the SUM() function to the SALARY column to build a list of the departments and their total salaries. The subsequent SELECT statement determines the largest of the sums returned by the function. The SELECT statement cannot determine the largest sum until all sums have been determined. This requires writing the results of the view to disk so that the SELECT statement can scan the view result to find the largest sum.

The need to write to disk can cause a negative impact on performance. Therefore, view merge is preferred over view materialization.
Checkpoint (1 of 2)

1. Assume that you have created a new table. How could you make sure that you will not have any duplicate rows in the table?

2. T/F. Check constraints can be used to enforce business rules requiring evaluation of one or more columns in a table row.

3. Assume that you want to ensure that all of the foreign key values in a dependent table exist in a key of the parent table. What will you need to define in the parent table? What will you need to define in the dependent table?

4. Assume that you need to insert a record in a table any time an employee receives a salary increase of more than 10 percent. How could you ensure that this happens without writing any application code?

Notes:
Write down your answers here:

1. ____________________________________________

2. ____________________________________________

3. ____________________________________________

4. ____________________________________________
Checkpoint (2 of 2)

5. Which clauses must not be included in a CREATE VIEW statement?

6. T/F. Views can be based on one or more tables, but not on other views.

7. Assume that you have created a view that combines information from the EMPLOYEE and DEPARTMENT tables. Is it possible to change data through this view? Is there some other way to change the data without using a view?

8. Assume that you need to create a view based on the EMPLOYEE table. The view should only provide information about employees having a yearly salary of less than 30000. Users of the view must not update the salary to a value higher than 30000. What should be included in the CREATE VIEW statement to ensure that users cannot use the view to change the salary to a value higher than 30000?

Notes:
Write down your answers here:

5. __________________________________________

6. __________________________________________

7. __________________________________________

8. __________________________________________

You will find an example of the answers in the Appendix.
Summary

Now that you have completed this unit, you should be able to:

- Code statements to:
  - Create tables, indexes and views
  - Alter tables
  - Implement referential integrity (RI)
  - Define triggers and check constraints

- Identify impacts and advantages of referential integrity, including impacts of delete rules

- Identify considerations when using triggers and check constraints

Notes:
Unit 3. Join

What This Unit Is About

This unit reviews inner joins and examines the various outer joins.

What You Should Be Able to Do

After completing this unit, you should be able to:

- Retrieve data from more than one table via inner joins
- Use outer joins

How You Will Check Your Progress

Accountability:

- Unit Checkpoint
- Machine Lab

Labs

- INNER JOIN
- OUTER JOIN

References

DB2 UDB for OS/390 and z/OS V7
SQL Reference (SC26-9944-02)

DB2 Administration Guide (DB2 UDB 7.2 for Windows)
DB2 SQL Reference (DB2 UDB 7.2 for Windows)
Objectives

After completing this unit, you should be able to:

- Use inner joins to retrieve data from more than one table
- Use outer joins complementing inner joins

Notes:

- After reviewing inner joins, we will discuss outer joins. Outer joins allow you to pick up rows whose join-column values do not match any values in the other table, as well as those which have matching values in the join columns.
3.1 Join Review and Outer Join Techniques
Sample Tables for Unit

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>LASTNAME</th>
<th>WORKDEPT</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>000010</td>
<td>HAAS</td>
<td>A00</td>
<td>52750.00</td>
</tr>
<tr>
<td>000030</td>
<td>Kwan</td>
<td>C01</td>
<td>38250.00</td>
</tr>
<tr>
<td>000120</td>
<td>O’Connell</td>
<td>A00</td>
<td>29250.00</td>
</tr>
<tr>
<td>000130</td>
<td>QUINTANA</td>
<td>C01</td>
<td>23800.00</td>
</tr>
<tr>
<td>000140</td>
<td>NICHOLLS</td>
<td>C01</td>
<td>28420.00</td>
</tr>
<tr>
<td>000400</td>
<td>Wilson</td>
<td>-</td>
<td>25400.00</td>
</tr>
</tbody>
</table>

DEPARTMENT
---
<table>
<thead>
<tr>
<th>DEPTNO</th>
<th>DEPTNAME</th>
<th>MGRNO</th>
</tr>
</thead>
<tbody>
<tr>
<td>A00</td>
<td>SPIFFY COMPUTER SERVICE DIV.</td>
<td>000010</td>
</tr>
<tr>
<td>C01</td>
<td>INFORMATION CENTER</td>
<td>000030</td>
</tr>
<tr>
<td>D01</td>
<td>DEVELOPMENT CENTER</td>
<td>-</td>
</tr>
</tbody>
</table>

PROJECT
---
<table>
<thead>
<tr>
<th>PROJNO</th>
<th>PROJNAME</th>
<th>DEPTNO</th>
<th>RESPEMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>AD3100</td>
<td>ADMIN SERVICES</td>
<td>D01</td>
<td>000010</td>
</tr>
<tr>
<td>IF1000</td>
<td>QUERY SERVICES</td>
<td>C01</td>
<td>000030</td>
</tr>
<tr>
<td>IF2000</td>
<td>USER EDUCATION</td>
<td>C01</td>
<td>000030</td>
</tr>
</tbody>
</table>

Notes:
- The visual shows the tables used for the examples of this unit.
- The tables are derived from the DB2 sample tables by:
  - Selecting the columns shown on the visual.
  - Selecting the rows with employee numbers 000010, 000030, 000120, 000130, and 000140 from the sample EMPLOYEE table.
  - Adding the row with employee number 000400 to cater for reasonable outer join examples. Although the definition of column WORKDEPT allows for null values, the original sample tables do not have an employee that has not been assigned to a department.
  - Selecting the rows with department numbers A00, C01, and D01 from the sample DEPARTMENT table.
  - Selecting the rows with project numbers AD3100, IF1000, and IF2000 from the sample PROJECT table.
Join

SELECT EMPNO, LASTNAME, DEPTNO, DEPTNAME
FROM EMPLOYEE, DEPARTMENT
WHERE WORKDEPT = DEPTNO

Notes:

- Sometimes, not all of the information that you want to see is in a single table. To get the report you want, you might need to retrieve some columns from one table and some columns from another table. You can use a SELECT statement to get columns from one or more tables in a single result set.

- If you read from two tables, the rows of the result set will always be a concatenation of rows from the first table with rows from the second table. However, if you do not provide a join condition, DB2 will join each row of the first table with every row of the second table. This is called a Cartesian Product and is usually considered very undesirable since most of the relationships between the rows are false. For example, a Cartesian Product involving the rows of tables EMPLOYEE and DEPARTMENT on the previous visual would include a row with the HAAS row of the EMPLOYEE table concatenated with the D01 row of the DEPARTMENT table, which is clearly not appropriate.

- Normally, you should only concatenate a row in the first table to a row in the second table if the two rows have the same values in some columns. This is accomplished by equating the corresponding columns in the SQL statement by means of a join condition.
The columns based on which the tables are combined (joined) are referred to as join columns.

- In the example on the current visual, the WHERE clause assures that a row in the EMPLOYEE table is only joined to a row in the DEPARTMENT table if the department numbers in both tables are the same, that is, WORKDEPT in table EMPLOYEE has the same value as DEPTNO in the DEPARTMENT table.

- This 'normal' join is also referred to as inner join, see next page.

- Logically additional predicates in the WHERE clause are applied after the join but in practice DB2 applies these so called local predicates as early in the join process as possible for performance reasons.

**Note:** This join syntax is available in all versions of DB2 on every platform.
**Inner Join**

```sql
SELECT EMPNO, LASTNAME, DEPTNO, DEPTNAME
FROM EMPLOYEE
INNER JOIN DEPARTMENT
ON WORKDEPT = DEPTNO
```

**Table:**

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>LASTNAME</th>
<th>DEPTNO</th>
<th>DEPTNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>000010</td>
<td>HAAS</td>
<td>A00</td>
<td>SPIFFY COMPUTER SERVICE DIV.</td>
</tr>
<tr>
<td>000120</td>
<td>O'CONNELL</td>
<td>A00</td>
<td>SPIFFY COMPUTER SERVICE DIV.</td>
</tr>
<tr>
<td>000030</td>
<td>KWAN</td>
<td>C01</td>
<td>INFORMATION CENTER</td>
</tr>
<tr>
<td>000140</td>
<td>NICHOLLS</td>
<td>C01</td>
<td>INFORMATION CENTER</td>
</tr>
<tr>
<td>000130</td>
<td>QUINTANA</td>
<td>C01</td>
<td>INFORMATION CENTER</td>
</tr>
</tbody>
</table>

**Notes:**

- The syntax used in this example means exactly the same as the syntax shown on the previous visual. This new standard syntax for an Inner Join is available on all DB2 platforms. The old Inner Join syntax remains supported on all platforms so you do not have to stop using it.
- In this new syntax, the keywords 'INNER JOIN' replace the comma in the FROM clause and the 'ON' keyword allows you to specify the Join condition as part of the FROM clause, rather than in the WHERE clause. A WHERE clause would follow the ON clause if there was a need for a WHERE, that is, additional conditions.
- Remember that the EMPLOYEE table contains an employee, WILSON, who has not been assigned to a department and the DEPARTMENT table contains a department (D01) that no employee works for. This is an Inner Join so neither of these “orphan” rows appears in the result set.
Right Outer Join

SELECT EMPNO, LASTNAME, DEPTNO, DEPTNAME
FROM EMPLOYEE
RIGHT OUTER JOIN DEPARTMENT
ON WORKDEPT = DEPTNO

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>LASTNAME</th>
<th>DEPTNO</th>
<th>DEPTNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>000010</td>
<td>HAAS</td>
<td>A00</td>
<td>SPIFFY COMPUTER SERVICE DIV.</td>
</tr>
<tr>
<td>000120</td>
<td>O'CONNELL</td>
<td>A00</td>
<td>SPIFFY COMPUTER SERVICE DIV.</td>
</tr>
<tr>
<td>000030</td>
<td>KWAN</td>
<td>C01</td>
<td>INFORMATION CENTER</td>
</tr>
<tr>
<td>000130</td>
<td>QUINTANA</td>
<td>C01</td>
<td>INFORMATION CENTER</td>
</tr>
<tr>
<td>000140</td>
<td>NICHOLLS</td>
<td>C01</td>
<td>INFORMATION CENTER</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>D01</td>
<td>DEVELOPMENT CENTER</td>
</tr>
</tbody>
</table>

Notes:

- An Outer Join gives you all the same rows as an inner join, plus the “orphan” rows that do not have matching values in the other tables.
- There are three types of Outer Joins:
  - RIGHT OUTER JOIN
  - LEFT OUTER JOIN
  - FULL OUTER JOIN
- The result set of a RIGHT OUTER JOIN contains all the rows that would have been created by an Inner Join, plus all “orphan” rows from the right-hand table (the one to the right of the keywords RIGHT OUTER JOIN).
- In the example, we list all employees who have been assigned to departments, plus all departments to which no employees have been assigned. Rows that refer to departments without an employee contain nulls in the columns that originated in the EMPLOYEE table.
**Left Outer Join**

![Diagram of Left Outer Join](image)

```sql
SELECT EMPNO, LASTNAME, DEPTNO, DEPTNAME
FROM EMPLOYEE
LEFT OUTER JOIN DEPARTMENT
ON WORKDEPT = DEPTNO
```

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>LASTNAME</th>
<th>DEPTNO</th>
<th>DEPTNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>000010</td>
<td>HAAS</td>
<td>A00</td>
<td>SPIFFY COMPUTER SERVICE DIV.</td>
</tr>
<tr>
<td>000030</td>
<td>KWAN</td>
<td>C01</td>
<td>INFORMATION CENTER</td>
</tr>
<tr>
<td>000120</td>
<td>O'CONNELL</td>
<td>A00</td>
<td>SPIFFY COMPUTER SERVICE DIV.</td>
</tr>
<tr>
<td>000130</td>
<td>QUINTANA</td>
<td>C01</td>
<td></td>
</tr>
<tr>
<td>000140</td>
<td>NICHOLLS</td>
<td>C01</td>
<td>INFORMATION CENTER</td>
</tr>
<tr>
<td>000400</td>
<td>WILSON</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Notes:**

- The result set of a LEFT OUTER JOIN contains all the rows that would have been created by an Inner Join, plus all “orphan” rows from the left-hand table (the one to the left of the keywords LEFT OUTER JOIN).

- In the example, we list all employees who have been assigned to a department with the names of their departments and all employees that have not been assigned to a department (of course, without a department name). Rows that refer to employees without a department contain nulls in the columns that originated in the DEPARTMENT table.
**Full Outer Join**

![Venn diagram of EMPLOYEE and DEPARTMENT tables demonstrating a Full Outer Join]

**SELECT** EMPNO, LASTNAME, DEPTNO, DEPTNAME  
**FROM** EMPLOYEE  
**FULL OUTER JOIN** DEPARTMENT  
**ON** WORKDEPT = DEPTNO  

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>LASTNAME</th>
<th>DEPTNO</th>
<th>DEPTNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>000120</td>
<td>O’CONNELL</td>
<td>A00</td>
<td>SPIFFY COMPUTER SERVICE DIV.</td>
</tr>
<tr>
<td>000010</td>
<td>HAAS</td>
<td>A00</td>
<td>SPIFFY COMPUTER SERVICE DIV.</td>
</tr>
<tr>
<td>000030</td>
<td>Kwan</td>
<td>C01</td>
<td></td>
</tr>
<tr>
<td>000130</td>
<td>QUINTANA</td>
<td>C01</td>
<td>INFORMATION CENTER</td>
</tr>
<tr>
<td>000140</td>
<td>NICHOLLS</td>
<td>C01</td>
<td>INFORMATION CENTER</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D01</td>
<td>DEVELOPMENT CENTER</td>
</tr>
<tr>
<td>000400</td>
<td>WILSON</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Notes:**
- The result set of a FULL OUTER JOIN contains all the rows that would have been created by an Inner Join, plus all “orphan” rows from both tables.
- In the example, we list all employees and all departments. Rows that refer to departments without an associated employee contain nulls in the columns that originated in the EMPLOYEE table. Rows that refer to employees without a department contain nulls in the columns that originated in the DEPARTMENT table.
Joins of More Than Two Tables

**SELECT**

\[
\begin{align*}
P.PROJNO, & P.PROJNAME, & D.DEPTNO, & D.MGRNO, \\
& E.LASTNAME & \text{AS MGRNAME}
\end{align*}
\]

**FROM**

\[
\begin{align*}
\text{PROJECT P} & \left\langle \text{INNER JOIN DEPARTMENT D ON P.DEPTNO} = \text{D.DEPTNO} \right\rangle \\
\text{LEFT OUTER JOIN EMPLOYEE E ON D.MGRNO} = \text{E.EMPNO}
\end{align*}
\]

**RESULT**

<table>
<thead>
<tr>
<th>PROJNO</th>
<th>PROJNAME</th>
<th>DEPTNO</th>
<th>MGRNO</th>
<th>MGRNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF2000</td>
<td>USER EDUCATION</td>
<td>C01</td>
<td>000030</td>
<td>KWAN</td>
</tr>
<tr>
<td>IF1000</td>
<td>QUERY SERVICES</td>
<td>C01</td>
<td>000030</td>
<td>KWAN</td>
</tr>
<tr>
<td>AD3100</td>
<td>ADMIN SERVICES</td>
<td>D01</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

- Left and right joins are not symmetrical.

**Notes:**

- Many tables can be joined in a single SELECT statement. The maximum number of tables in the join depends on the version of DB2 you are using and the operating system. For example, some of the limits are as follows:
  - For DB2 UDB for UNIX, Windows and OS/2, the maximum number of tables that can be joined depends on the available storage.
  - In DB2 UDB for OS/390, the maximum number of tables that can be joined is 225 or fewer, depending on the complexity of the SELECT statement. (Only 15 tables in each FROM clause.)

- If three or more tables are joined, different types of joins may be used to join the various tables. Of course, the application logic, that is, the desired result, dictates which types of joins must be used.

- If \( n \) tables are joined, there must be at least \( n-1 \) different join conditions and one column from each table must appear in at least one of the join conditions. If this rule is not observed a Cartesian Product will result.
A single join operation combines two tables. If more than two tables are joined, the subsequent join operation combines the intermediate result table of the preceding join operations with the next table.

Your application implies a sequence in which the tables and intermediate result tables should be joined, and, you must express this in the FROM clause. As there may be different ways that you can achieve the desired output, DB2 may decide on joining the tables differently or in a different sequence.

The purpose of the example above is to determine, for each project that has been assigned to a department, the manager of the department performing the project. Projects that have been assigned to departments without manager should be listed as well. You can use different methods to solve this problem:

- The SELECT statement on the visual offers one possibility. You can first join the PROJECT table to the DEPARTMENT table via an Inner Join to determine, for each project, its project number, its project name, the department number of the department performing the project, and the employee number of the manager for the department. Then, you can "Left Outer Join" the intermediate result table to the EMPLOYEE table to add the name of the manager for those projects whose responsible department has a manager.

- As an alternative, you can first determine, for each department, its department number, the employee number of its manager, and, if the department has a manager, his/her name. This can be achieved by a Left Outer Join of the DEPARTMENT table to the EMPLOYEE table. Then, you can perform an Inner Join of the PROJECT table to the intermediate result table to determine, for each project, its project number, its project name, the department number of the department performing the project, the employee number of the manager of the department, and if the department has a manager, his/her name.

You can formulate the appropriate SELECT statement as follows. It provides the same result as the SELECT statement on the visual:

\[
\begin{align*}
\text{SELECT} & \quad PROJNO, \quad PROJNAME, \quad P.DEPTNO, \quad MGRNO, \\
& \quad \text{LASTNAME AS MGRNAME} \\
\text{FROM} & \quad \text{PROJECT P} \\
\text{INNER JOIN} & \quad \text{(DEPARTMENT D} \\
& \quad \text{LEFT OUTER JOIN EMPLOYEE} \\
& \quad \text{ON MGRNO = EMPNO} \\
& \quad \text{ON P.DEPTNO = D.DEPTNO}
\end{align*}
\]

Inner Joins and Full Outer Joins are symmetrical. For them, the order of the two tables being joined does not matter. Accordingly, \text{PROJECT P} and \text{(DEPARTMENT D LEFT OUTER JOIN EMPLOYEE)} in the preceding SELECT statement could be interchanged without changing the result set.
• Left or Right Outer Joins are not symmetrical. Therefore, the order in which the tables are specified is important. If you reverse the order of the tables, a Left Outer Join becomes a Right Outer Join and vice versa. In the preceding SELECT statement, 
(DEPARTMENT D LEFT OUTER JOIN EMPLOYEE) can also be formulated as 
(EMPLOYEE RIGHT OUTER JOIN DEPARTMENT D) without changing the result set.
Joins and Local Predicates (Part 1)

**Figure 3-9. Joins and Local Predicates (Part 1)**

```
SELECT EMPNO, LASTNAME, SALARY, DEPTNO, DEPTNAME
FROM EMPLOYEE
FULL OUTER JOIN DEPARTMENT
ON WORKDEPT = DEPTNO
WHERE SALARY > 30000 AND DEPTNAME LIKE ' %CENTER% '
```

### Notes:

- There are two types of 'predicates' (search conditions): join predicates and local predicates.
  - Join predicates are those specified via the ON keyword in the FROM clause. They tell DB2 which columns in the participating tables of a join operation should be matched to join the rows correctly.
  - Local predicates are those specified via the WHERE clause.
- For the purpose of predicting which rows will be returned from a SELECT statement, assume that the join operation is performed before the other clauses in the statement.
Joins and Local Predicates (Part 2)

SELECT EMPNO, LASTNAME, SALARY, DEPTNO, DEPTNAME
FROM EMPLOYEE
FULL OUTER JOIN DEPARTMENT
ON WORKDEPT = DEPTNO
WHERE SALARY > 30000 AND DEPTNAME LIKE '%CENTER%'

Notes:

- As mentioned on previous page, for the purpose of predicting which rows will be returned from a SELECT statement, assume that the join operation is performed before the other clauses in the statement.

- In the example above, the WHERE clause for the SELECT statement is shown underneath the intermediate result table to emphasize that the predicates of the WHERE clause are applied to the intermediate result table.

The WHERE clause selects rows from the intermediate result table with a salary higher than 30000 (SALARY > 30000) and a department name containing the character string 'CENTER' (DEPTNAME LIKE '%CENTER%'). The only row in the intermediate result table that satisfies both conditions is the row for employee KWAN.

- The result of this Outer Join is equivalent to the result of an Inner Join and, in this case, coding an Inner Join is preferable.

- If the intention was to “Full Outer Join” all employees having a salary higher than 30000 to all departments whose name contains the character string 'CENTER', the query
would have to be formulated differently and would require nested table expressed. They will be discussed in a later unit. As you can easily see, the result would be different since employees KWAN and HAAS both have a salary higher than 30000 and, thus, both would appear in the result set.
Checkpoint

1. Assume that you want to list all employees and their children. You want the report to include employees who have no children. Which technique will you need?

2. T/F. If you specify two or more tables in the FROM clause, but omit all join conditions, the result set will be empty.

3. Assume that you need to list only employees who have no children. Is this possible with a join?

Notes:
Write down your answers here:
1. 

2. 

3. 

You will find an example of the answers in the Appendix.
Summary

Now that you have completed this unit, you should be able to:

- Use inner joins to retrieve data from more than one table
- Use outer joins complementing inner joins

Notes:
Unit 4. CASE, CAST and Summary Tables

What This Unit Is About

This unit provides information on CASE, CAST and Summary Tables.

What You Should Be Able to Do

After completing this unit, you should be able to:

- Identify when CASE expressions can be used
- Code CASE expressions in SELECT list and in the WHERE clause
- Identify when CAST specifications can be used
- Understand the advantages of Summary Tables

How You Will Check Your Progress

Accountability:

- Unit Checkpoint
- Machine Lab

Labs

- Write SQL which uses CASE and CAST

References

DB2 UDB for OS/390 and z/OS V7
   SQL Reference (SC26-9944-02)

DB2 Administration Guide (DB2 UDB 7.2 for Windows)
DB2 SQL Reference (DB2 UDB 7.2 for Windows)
Objectives

After completing this unit, you should be able to:

- Identify when CASE expressions can be useful
- Code CASE expressions in the SELECT list and in the WHERE clause
- Identify when CAST specifications can be used
- Understand the advantages of using Summary Tables

Notes:

- The relatively new SQL, CASE expressions and CAST specifications can be very useful to avoid coding long SQL statements.
- Summary Tables is a feature on DB2 UDB for UNIX, Windows and OS/2 since Version 5.2 and is very useful in OLAP environments.
4.1 CASE, CAST and Summary Tables
CASE Expressions in SELECT

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>LASTNAME</th>
<th>WORKDEPT</th>
<th>SALARY</th>
<th>HIREDATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>00010</td>
<td>HAAS</td>
<td>A00</td>
<td>52750.00</td>
<td>1965-01-01</td>
</tr>
<tr>
<td>00030</td>
<td>KWAN</td>
<td>C01</td>
<td>38250.00</td>
<td>1975-04-05</td>
</tr>
<tr>
<td>00120</td>
<td>O’CONNELL</td>
<td>A00</td>
<td>29250.00</td>
<td>1963-12-05</td>
</tr>
<tr>
<td>00130</td>
<td>QUINTANA</td>
<td>C01</td>
<td>23800.00</td>
<td>1971-07-28</td>
</tr>
<tr>
<td>00140</td>
<td>NICHOLLS</td>
<td>C01</td>
<td>28420.00</td>
<td>1976-12-15</td>
</tr>
</tbody>
</table>

**SELECT EMPNO, LASTNAME, CASE WHEN SALARY < 25000 THEN 'LOW' WHEN SALARY >= 25000 AND SALARY < 40000 THEN 'AVERAGE' ELSE 'HIGH' END AS SALARY_CLASS, CASE SUBSTR(WORKDEPT,1,1) WHEN 'A' THEN 'ADMINISTRATION' WHEN 'C' THEN 'CUSTOMER SERVICE' WHEN 'D' THEN 'DEVELOPMENT' ELSE NULL END AS AREA_TYPE FROM EMPLOYEE |

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>LASTNAME</th>
<th>SALARY_CLASS</th>
<th>AREA_TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>00010</td>
<td>HAAS</td>
<td>HIGH</td>
<td>ADMINISTRATION</td>
</tr>
<tr>
<td>00030</td>
<td>KWAN</td>
<td>AVERAGE</td>
<td>CUSTOMER SERVICE</td>
</tr>
<tr>
<td>00120</td>
<td>O’CONNELL</td>
<td>AVERAGE</td>
<td>ADMINISTRATION</td>
</tr>
<tr>
<td>00130</td>
<td>QUINTANA</td>
<td>LOW</td>
<td>CUSTOMER SERVICE</td>
</tr>
<tr>
<td>00140</td>
<td>NICHOLLS</td>
<td>AVERAGE</td>
<td>CUSTOMER SERVICE</td>
</tr>
</tbody>
</table>

**Notes:**

- CASE expressions allow a user to code IF-THEN-ELSE logic instead of simple values.
- DB2 evaluates the conditions in order from top to bottom. The value returned from the CASE expression is the result of the first true condition. If no case is satisfied, the ELSE value is used. If no ELSE is coded, the value returned is NULL.
- CASE expressions can appear in various places in an SQL statement, including the SELECT clause, the WHERE clause, the GROUP BY clause, and the HAVING clause. It can also appear in IN and VALUES clauses.
- The examples illustrate the use of CASE expressions in the SELECT clause. The first CASE expression returns a salary classification for the employees rather than the salary itself. The salary classification is based on the range of the salary. If the salary is less than 25000, the salary classification is 'LOW'. If it is 'AVERAGE' if the salary is equal to or higher than 25000, but is lower than 40000. Salaries of 40000 and higher are considered 'HIGH' as specified via the ELSE keyword. The appropriate column of the result table is called SALARY_CLASS. This example, which has no expression
between the CASE keyword and the first THEN keyword, is called a **searched WHEN clause**.

- The second example, which has an expression between the CASE keyword and the first THEN keyword, is called a **simple WHEN clause**. The expression uses the SUBSTR() scalar function to determine the first character of the WORKDEPT column and the THEN clauses are evaluated for each expected value of that character. If the value is 'A', 'ADMINISTRATION' is displayed in the result, if the value is 'C', 'CUSTOMER SERVICE' is displayed, and, if the value is 'D', 'DEVELOPMENT' is displayed. Null is displayed for all other values.
### CASE Expression in WHERE Clause

**Figure 4-3. CASE Expression in WHERE Clause**

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>LASTNAME</th>
<th>WORKDEPT</th>
<th>SALARY</th>
<th>COMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>000010</td>
<td>HAAS</td>
<td>A00</td>
<td>52750.00</td>
<td>4220.00</td>
</tr>
<tr>
<td>000030</td>
<td>KWAN</td>
<td>C01</td>
<td>38250.00</td>
<td>3060.00</td>
</tr>
<tr>
<td>000120</td>
<td>O'CONNELL</td>
<td>A00</td>
<td>29250.00</td>
<td>2340.00</td>
</tr>
<tr>
<td>000130</td>
<td>QUINTANA</td>
<td>C01</td>
<td>23800.00</td>
<td>1904.00</td>
</tr>
<tr>
<td>000140</td>
<td>NICHOLLS</td>
<td>C01</td>
<td>28420.00</td>
<td>2274.00</td>
</tr>
</tbody>
</table>

**SELECT** EMPNO, LASTNAME, SALARY+COMM AS TOT_SAL  
**FROM** EMPLOYEE  
**WHERE** (CASE WHEN SALARY = 0 THEN NULL  
ELSE COMM / SALARY  
END ) > 0.08

**Notes:**

- This visual shows an example of CASE statement usage to protect from division by 0 errors.
- It finds the employees who earn more than 8% of their income from commission, but are not fully paid on commission.
CASE Expression in a Function

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>LASTNAME</th>
<th>WORKDEPT</th>
<th>SALARY</th>
<th>COMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>000010</td>
<td>HAAS</td>
<td>A00</td>
<td>52750.00</td>
<td>4220.00</td>
</tr>
<tr>
<td>000030</td>
<td>KWAN</td>
<td>C01</td>
<td>0.00</td>
<td>3060.00</td>
</tr>
<tr>
<td>000120</td>
<td>O'CONNELL</td>
<td>A00</td>
<td>29250.00</td>
<td>2340.00</td>
</tr>
<tr>
<td>000130</td>
<td>QUINTANA</td>
<td>C01</td>
<td>23800.00</td>
<td>1904.00</td>
</tr>
<tr>
<td>000140</td>
<td>NICHOLLS</td>
<td>C01</td>
<td>28420.00</td>
<td>2274.00</td>
</tr>
</tbody>
</table>

SELECT MIN (CASE
WHEN SALARY = 0 THEN NULL
ELSE SALARY END) AS MIN_SAL
FROM EMPLOYEE

Figure 4-4. CASE Expression in a Function

Notes:

- This visual shows an example of CASE in a function.
- It finds the minimum salary for all employees except those with a salary equal to zero.
- The same query could also be written:

```sql
select min(salary)
from employee
where not salary = 0
```
**Nested CASE Expression**

**Notes:**

- This visual shows an example of a nested CASE expression.
- It computes a different salary increase depending on the salary and the employee number.
CAST Specifications

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>LASTNAME</th>
<th>WORKDEPT</th>
<th>SALARY</th>
<th>COMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>000010</td>
<td>HAAS</td>
<td>A00</td>
<td>52750.00</td>
<td>4220.00</td>
</tr>
<tr>
<td>000030</td>
<td>KWAN</td>
<td>C01</td>
<td>38250.00</td>
<td>3060.00</td>
</tr>
<tr>
<td>000120</td>
<td>O'CONNELL</td>
<td>A00</td>
<td>29250.00</td>
<td>2340.00</td>
</tr>
<tr>
<td>000130</td>
<td>QUINTANA</td>
<td>C01</td>
<td>23800.00</td>
<td>1904.00</td>
</tr>
<tr>
<td>000140</td>
<td>NICHOLLS</td>
<td>C01</td>
<td>28420.00</td>
<td>2274.00</td>
</tr>
</tbody>
</table>

**Figure 4-6.** CAST Specifications

```sql
SELECT EMPNO, COMM/SALARY AS COL2,
CAST(COMM/SALARY AS DEC(9,2)) AS COL3
FROM EMPLOYEE
WHERE EMPNO='000140'
```

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>COL2</th>
<th>COL3</th>
</tr>
</thead>
<tbody>
<tr>
<td>000140</td>
<td>0.0976073187895847994370</td>
<td>0.09</td>
</tr>
</tbody>
</table>

**Notes:**

- DB2 supports the CAST syntax defined by SQL92 Standard.
- Casting is often used in programming languages to refer to the process of changing a value from one data type to another. Casting in SQL has the same meaning.
- In the example the division of COMM/SALARY in COL2 gets a default datatype depending on the data types of the columns COMM and SALARY. With CAST, the division can be presented with different data type, in this case DECIMAL(9,2). In practice this limits the number of places to the right of the decimal point as shown in COL3 to two.
- CAST is also useful when a value of a particular data type is needed as the parameter of a function.
- One special use of CAST is to cast a NULL value to a particular data type:

```sql
SELECT EMPNO, CAST(NULL AS SMALLINT)
FROM EMPLOYEE
WHERE EMPNO < '000050'
```
This can be useful to present null values in a result set.

- In the last Unit of the course, Unit 8, we cover UDT, User-Defined Distinct Types. Then you will see which casting functions DB2 UDB automatically creates when UDTs are created.
Automatic Summary Tables

- Materialized views
- Aggregate Aware Optimization
- Contains pre-computed results
- Queries can reuse Summary Tables without accessing base tables
- Improved performance

Notes:

- DB2 UDB on UNIX, Windows and OS/2 supports summary tables since Version 5.2.
- A Summary table is a table whose definition is based on the result of a query. As such, the summary table typically contains pre-computed results based on the data existing in the table or tables on which its definition is based.
- If the optimizer determines that a dynamic query will run more efficiently against a summary table than the base table, the query executes against the summary table, and we should receive the result of the query faster than access directly to the base tables. The query rewrite function of the optimizer will access a summary table if it determines that the query can be answered by using the data in the summary table instead of accessing the base table or tables.
- On DB2 UDB UNIX, Windows and OS/2, you can set the optimization level. For the optimizer to consider the use of summary tables, the optimization level has to be 2 or greater.
- Summary tables are mainly used for dynamic SQL.
Summary Tables - Example

CREATE TABLE DEPT_GROUP
AS ( SELECT WORKDEPT,
      SUM(SALARY) AS SALARY,
      SUM(BONUS) AS BONUS
      FROM EMPLOYEE
      GROUP BY WORKDEPT )
DATA INITIALLY DEFERRED
REFRESH DEFERRED

- **DATA INITIALLY DEFERRED**
  - Data is not inserted into table as part of the CREATE TABLE statement

- **REFRESH DEFERRED**
  - Data in the table can be refreshed at any time using REFRESH TABLE statement

- **REFRESH TABLE statement is the only way to populate the table**

**Notes:**

- REFRESH - Indicates how the data in the table is maintained.

- DEFERRED - The data in the table can be refreshed at any time using the REFRESH TABLE statement. The data in the table only reflects the result of the query at the time of the REFRESH TABLE statement is processed (a snapshot).

- IMMEDIATE - -The changes made to the underlaying tables as part of a DELETE, INSERT or UPDATE are cascaded to the summary table. In this case, the contents of the table, at any point-in-time, are the same as the specified subselect where to be processed.

- When REFRESH DEFERRED or REFRESH IMMEDIATE is specified, the fullselect cannot include:
  - References to a view or a summary table
  - Functions that have external action
  - Table of view references to system objects (explain tables also should not be specified)
• When REFRESH IMMEDIATE is specified,
  - The fullselect must be a subselect and cannot include:
    — Functions that are not deterministic
    — Scalar fullselects
    — Predicates with fullselects
    — Special registers
  - A GROUP BY must be included in the subselect
  - The select list must have a COUNT(*) function and no DISTINCT
  - Only SUM (of not nullable columns), or COUNT column functions are allowed in the select list (without DISTINCT) and the other select list items must be included in the GROUP BY clause
  - All GROUP BY items must be included in the select list
  - A HAVING clause is not allowed
• To exploit Summary table and ensure that we are not using 'old' data from the Summary table: SET CURRENT REFRESH AGE
• See DB2 UDB UNIX, Windows and OS/2 SQL Reference for full syntax and comments.
Summary Tables - Considerations

- If optimizer decides to use a summary table, access to base tables is eliminated.
- You should create non-unique indexes.

- Some of the key restrictions regarding summary tables:
  - You cannot alter a summary table.
  - You cannot alter the length of a column for a base table if that table has a summary table.
  - You cannot import data into a summary table.
  - You cannot create a unique index on a summary table.
  - You cannot create a summary table based on the result of a query that references one or more nicknames.

Notes:

- Check if Summary table is being used by looking at explain.
- Each Summary Table should serve many queries.
- Create indexes on summary tables and update statistics on summary tables.
- Consider physical storage requirements.
Checkpoint

1. T/F. CASE can be coded in both the SELECT-list and in the WHERE clause.

2. T/F. CAST specifications can be used to cast the value NULL to a parameter.

Notes:
Write down your answers here:
1. 
   ________________________________________________________________

2. 
   ________________________________________________________________

You will find an example of the answers in the Appendix.
Summary

Now that you have completed this unit, you should be able to:

- Identify the advantages and use CASE
- Use CAST Expressions to convert between data types
- Understand the advantages of Summary Tables

Notes:
Unit 5. Using Subqueries

What This Unit Is About

This unit provides information on how to use complex subqueries.

What You Should Be Able to Do

After completing this unit, you should be able to:

- Code subqueries using the ALL, ANY/SOME, and EXISTS keywords
- Code correlated subqueries
- Choose the proper type of subquery to use in each case

How You Will Check Your Progress

Accountability:

- Unit Checkpoint
- Machine Lab

Labs
- SUBQUERY using keyword ALL
- SUBQUERY using keyword ANY
- SUBQUERY using keyword EXISTS
- CORRELATED SUBQUERY

References

DB2 UDB for OS/390 and z/OS V7 SQL Reference (SC26-9944-02)

DB2 Administration Guide (DB2 UDB 7.2 for Windows)
DB2 SQL Reference (DB2 UDB 7.2 for Windows)
Objectives

After completing this unit, you should be able to:

- Code subqueries using the ALL, ANY/SOME, and EXISTS keywords
- Code correlated subqueries
- Choose the proper type of subquery to use in each case

Notes:

- It is often useful to imbed one query within another. This enables the inner query to get an answer to a question and then plug that answer into the outer query to control which rows are returned by the outer query. This inner query is called a subquery.
5.1 Using Subqueries
Subquery With Basic Predicate

Notes:

- A basic predicate is one that uses =, <, >, or any combination of these symbols. A subquery in a basic predicate must specify a single result column and cannot return more than one value.

- The result of the predicate can be true, false, or unknown.
  - The predicate result is unknown if:
    - the value of the expression on the left side of the predicate is null. For example, if the predicate is WHERE BONUS + COMM > 50000, and COMM is null in the row being read by the outer query, the expression BONUS + COMM evaluates to null and the value of the entire predicate is unknown for this row of the outer query. In other words, it is not known whether or not the row currently being read by the outer satisfies the predicate.
    - the subquery result set is empty or null. For example, if the predicate is WHERE SALARY > (SELECT SALARY FROM EMPLOYEE WHERE EMPNO = '000405')
and there is no employee 000405, the result of the subquery will be empty and
the entire predicate is unknown for any row of the outer query.

- The predicate result is false if the row in the outer query does not satisfy the
  operator of the subquery. The value returned by the subquery in the example on the
  visual is 52750.00. If the value in the current row of the outer query is 38250.00, the
  predicate value is false.

- In all other cases, the predicate value is true.

The row from the outer query only appears in the final result if the predicate value is true
for that row.

- The subquery in the example determines the largest SALARY value in the EMPLOYEE
  table. The outer query then determines employee number, last name, and salary of the
  persons - or people - whose salary is equal to the one returned by the subquery.

  Note: Several people may earn the same salary. If several people earn the maximum
  salary, the subquery will still return just that one value, but the outer query will return all
  people who earn the maximum salary.
Subquery With IN Predicate

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>LASTNAME</th>
<th>WORKDEPT</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>000010</td>
<td>HAAS</td>
<td>A00</td>
<td>52750.00</td>
</tr>
<tr>
<td>000030</td>
<td>KWAN</td>
<td>C01</td>
<td>38250.00</td>
</tr>
<tr>
<td>000120</td>
<td>O'CONNELL</td>
<td>A00</td>
<td>29250.00</td>
</tr>
<tr>
<td>000130</td>
<td>QUINTANA</td>
<td>C01</td>
<td>23800.00</td>
</tr>
<tr>
<td>000140</td>
<td>NICHOLLS</td>
<td>C01</td>
<td>28420.00</td>
</tr>
</tbody>
</table>

**Notes:**

The IN predicate compares a value or values with a collection of values.

The fullselect must identify a number of columns that is the same as the number of expressions specified to the left of the IN keyword. The fullselect may return any number of rows.

- The IN predicate, when used with a subquery, enables the outer query to compare the values in columns of the outer table with a list of comparable values provided by the subquery.
- For an IN predicate, if any of the non-null values returned by the subquery match any of the non-null values of the column being searched by the outer query, the rows in the outer query that contain matching values will appear in the final result set. Nulls in the subquery result will never match nulls in the outer query result.
- Only SALLY KWAN is currently assigned to a project that belongs to her department.
Subquery With NOT IN Predicate

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>LASTNAME</th>
<th>WORKDEPT</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>000010</td>
<td>HAAS</td>
<td>A00</td>
<td>52750.00</td>
</tr>
<tr>
<td>000030</td>
<td>Kwan</td>
<td>C01</td>
<td>38250.00</td>
</tr>
<tr>
<td>000120</td>
<td>O’Connell</td>
<td>A00</td>
<td>29250.00</td>
</tr>
<tr>
<td>000130</td>
<td>Quintana</td>
<td>C01</td>
<td>23800.00</td>
</tr>
<tr>
<td>000140</td>
<td>Nicholls</td>
<td>C01</td>
<td>28420.00</td>
</tr>
</tbody>
</table>

Which employees are not responsible for projects?

<table>
<thead>
<tr>
<th>PROJNO</th>
<th>PROJNAME</th>
<th>DEPTNO</th>
<th>RESPEMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>AD3100</td>
<td>ADMIN SERVICES</td>
<td>D01</td>
<td>000010</td>
</tr>
<tr>
<td>IF1000</td>
<td>QUERY SERVICES</td>
<td>C01</td>
<td>000030</td>
</tr>
<tr>
<td>IF2000</td>
<td>USER EDUCATION</td>
<td>C01</td>
<td>000030</td>
</tr>
</tbody>
</table>

**SELECT EMPNO, LASTNAME**
**FROM EMPLOYEE**
**WHERE EMPNO NOT IN**

(SELECT RESPEMP FROM PROJECT)

**Notes:**

- The subquery in the example builds a list of the employee numbers of employees that are responsible for projects (RESPEMP in the PROJECT table). The outer query determines the employee number and last name of any employee whose employee number is **NOT** in the list returned by the subquery. In other words, the query is reporting on employees who are not responsible for projects.

- When **IN** is prefaced with **NOT**, as in the example on the visual, the subquery should be written to return only non-null values. Then, all of the rows of the outer query that do not match any of the values returned by the subquery will appear in the result set. If the **NOT IN** subquery returns a null value, the outer query will always produce an empty result set.
NOT IN Predicate for Nullable Column

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>LASTNAME</th>
<th>WORKDEPT</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>000010</td>
<td>HAAS</td>
<td>A00</td>
<td>52750.00</td>
</tr>
<tr>
<td>000030</td>
<td>KWAN</td>
<td>C01</td>
<td>38250.00</td>
</tr>
<tr>
<td>000120</td>
<td>O'CONNELL</td>
<td>A00</td>
<td>29250.00</td>
</tr>
<tr>
<td>000130</td>
<td>QUINTANA</td>
<td>C01</td>
<td>23800.00</td>
</tr>
<tr>
<td>000140</td>
<td>NICOLLAS</td>
<td>C01</td>
<td>28420.00</td>
</tr>
<tr>
<td>000400</td>
<td>WILSON</td>
<td>NULL</td>
<td>25400.00</td>
</tr>
</tbody>
</table>

**Notes:**

- As mentioned in conjunction with the previous visual, it is necessary to be careful with nulls in queries using NOT IN. This is illustrated by the visual.

- For this example, it is assumed that employee WILSON has been added to the EMPLOYEE table with an employee number of 000400, a salary of 25400.00, but without a work department.

- If you want to find all departments for which no employees are working, you might be tempted to use the following query:

  ```sql
  SELECT DEPTNO, DEPTNAME
  FROM DEPARTMENT
  WHERE DEPTNO
  NOT IN (SELECT WORKDEPT
           FROM EMPLOYEE)
  ```

  This would **not** work. The result set would be **EMPTY**. The subquery result set would contain the department numbers A00, C01, A00, C01, C01, and null (for WILSON).
That would make the whole predicate in the outer query evaluate to UNKNOWN, rather than TRUE or FALSE, and no rows in the DEPARTMENT table would satisfy the query.

- To get the desired result, you must change the query to:

```sql
SELECT DEPTNO, DEPTNAME
    FROM DEPARTMENT
    WHERE DEPTNO
      NOT IN (SELECT WORKDEPT
                FROM EMPLOYEE
                WHERE WORKDEPT IS NOT NULL)
```
Subquery With ALL Predicate (Part 1)

Notes:

- The purpose of the query is to determine the department with the highest salary costs. For this department, its department number and salary cost should be listed.
- To solve the posed problem, we formulate the purpose of the query more precisely: Find the department(s) for which the sum of all salaries for employees working for the department(s), that is, the total salary paid by the department(s), is the highest of all departments. Note that there may be more than one department with the highest total salary.
- To find the greatest value out of a set of values, you must determine the value of the set that is greater than or equal to all values of the set including the value itself.
- The requested information can be extracted from the EMPLOYEE table.
- To find the total salaries for the departments, you need to use the SUM() column function and group the salaries by departments (WORKDEPT).
- Since the outer query needs to select departments rather than employees, a HAVING clause is needed rather than a WHERE clause. The HAVING clause must express the
condition that, for a summary row of the outer query to be displayed, the total salary (sum of salaries for the department) must be higher than or equal to the sum of salaries for all departments as illustrated by the visual.
Notes:

- A quantified predicate compares each value in the outer query with the set of values provided by the subquery. A quantified predicate is one that uses one of the following keywords: ALL, ANY, or SOME. As usual, the subselect must specify a single result column. It may return any number of values.

- When the ALL keyword is specified, the result of the predicate is:
  - True if the subquery returns no values or if the specified relationship is true for every value returned by the subquery.
  - False if the specified relationship is false for at least one value returned by the subquery.
  - Unknown if the specified relationship is not false for any values returned by the subquery and at least one comparison is unknown because of a null value. Unknown is treated as false.

- The subquery in the example provides a list of values. Each value is the total salary paid by a department to the employees working for that department. There is one value
for each department. The outer query ensures that a grouped row (by WORKDEPT) of the outer table is only displayed if the appropriate department's total salary is higher than or equal to every value (all values) in the list returned by the subquery. In the example, the total salary of only one of the departments in the outer result is higher than or equal to every value in the subquery result. Thus, that is the department with the highest salary cost.
Subquery With ANY or SOME Predicate (Part 1)

Notes:

- The task is to determine all employees whose salary is higher than the average salary of at least one department. For these employees, their employee number, last name, and salary should be listed.
- We can reformulate the problem as follows which indicates the predicate needed for the subquery:
  Determine all employees whose salary is higher than the average salary of any one of the departments. If you wanted, you could also say: “... higher than the average salary of some of the departments”.
- Note that the salaries of the employees are to be compared with the average salaries of the departments and not with the average salary for all employees. As indicated in the visual, the averages for the departments are 41000.0000000 (A00) and 30156.6666666 (C01), whereas the average salary for all listed employees would be 34494.0000000.
- Because employees have to be determined rather than departments, no grouping is required for the outer query.
Subquery With ANY or SOME Predicate (Part 2)

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>LASTNAME</th>
<th>WORKDEPT</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>000010</td>
<td>HAAS</td>
<td>A00</td>
<td>52750.00</td>
</tr>
<tr>
<td>000030</td>
<td>KWAN</td>
<td>C01</td>
<td>38250.00</td>
</tr>
<tr>
<td>000120</td>
<td>O’CONNELL</td>
<td>A00</td>
<td>29250.00</td>
</tr>
<tr>
<td>000130</td>
<td>QUINTANA</td>
<td>C01</td>
<td>23800.00</td>
</tr>
<tr>
<td>000140</td>
<td>NICHOLLS</td>
<td>C01</td>
<td>28420.00</td>
</tr>
</tbody>
</table>

SELECT EMPNO, LASTNAME, SALARY
FROM EMPLOYEE
WHERE SALARY > ANY (SELECT AVG(SALARY) FROM EMPLOYEE GROUP BY WORKDEPT)

Figure 5-9. Subquery With ANY or SOME Predicate (Part 2)

Notes:

- The predicates ANY and SOME have the same meaning and effect. When ANY or SOME are used, the result of the predicate is:
  - True if the specified relationship is true for at least one value returned by the subquery.
  - False if the subquery returns no value or if the specified relationship is false for every value returned by the subquery.
  - Unknown if the specified relationship is not true for any values returned by the subquery and at least one comparison is unknown because of a null value.

- The subquery in the example provides a list of values. Each value is the average salary paid by one department. To obtain the average salaries of the departments, grouping by WORKDEPT is necessary in the subquery.

The outer query compares the salary of each employee to the average-salary values for the departments in the list returned by the subquery. If the salary for an employee is higher than any of the values returned by the subquery, even the lowest one, the
employee number, last name, and salary for the employee are displayed. In other words, if any employee's individual salary is higher than the average salary for any of the departments, we will report on that individual.

In the example, the salary of employee HAAS is higher than all of the department averages. Employee KWAN qualifies because her salary is higher than the average salary of department C01 (her own), although her salary is not higher than the average salary of department A00.
Subquery With EXISTS Predicate (Part 1)

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>LASTNAME</th>
<th>WORKDEPT</th>
<th>SALARY</th>
<th>HIREDATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>000010</td>
<td>HAAS</td>
<td>A00</td>
<td>52750.00</td>
<td>1965-01-01</td>
</tr>
<tr>
<td>000030</td>
<td>KWAN</td>
<td>C01</td>
<td>38250.00</td>
<td>1975-04-05</td>
</tr>
<tr>
<td>000120</td>
<td>O'CONNELL</td>
<td>A00</td>
<td>29250.00</td>
<td>1963-12-05</td>
</tr>
<tr>
<td>000130</td>
<td>QUINTANA</td>
<td>C01</td>
<td>23800.00</td>
<td>1971-07-28</td>
</tr>
<tr>
<td>000140</td>
<td>NICHOLLS</td>
<td>C01</td>
<td>28420.00</td>
<td>1976-12-15</td>
</tr>
</tbody>
</table>

SELECT EMPNO, LASTNAME, SALARY, HIREDATE
FROM EMPLOYEE
WHERE SALARY > 30000 AND
EXISTS at least one employee earning less than $25,000

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>LASTNAME</th>
<th>WORKDEPT</th>
<th>SALARY</th>
<th>HIREDATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>000130</td>
<td>QUINTANA</td>
<td>C01</td>
<td>23800.00</td>
<td>1971-07-28</td>
</tr>
</tbody>
</table>

Notes:

• In this example, the task is to produce a listing of all employees earning more than 30,000, if there is any employee earning less than 25,000. The listing should contain the employee number, last name, salary, and hiring date.

• Let us reformulate the problem so that we can see more clearly which predicate must be used to solve the problem:

List employee number, last name, salary, and hiring date for all employees if and only if there exists at least one employee who has a salary less than 25000.

• Accordingly, the subquery must return a value of true if the EMPLOYEE table contains at least one employee with a salary less then 25000. The predicate is then true for all rows retrieved by the outer query. The subquery must return a value of false if none such employee exists. In this case, the predicate is false for all rows retrieved by the outer query resulting in no rows being displayed.
Subquery With EXISTS Predicate (Part 2)

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>LASTNAME</th>
<th>WORKDEPT</th>
<th>SALARY</th>
<th>HIREDATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>000010</td>
<td>HAAS</td>
<td>A00</td>
<td>52750.00</td>
<td>1965-01-01</td>
</tr>
<tr>
<td>000030</td>
<td>KWAN</td>
<td>C01</td>
<td>38250.00</td>
<td>1975-04-05</td>
</tr>
<tr>
<td>000120</td>
<td>O’CONNELL</td>
<td>A00</td>
<td>29250.00</td>
<td>1963-12-05</td>
</tr>
<tr>
<td>000130</td>
<td>QUINTANA</td>
<td>C01</td>
<td>23800.00</td>
<td>1971-07-28</td>
</tr>
<tr>
<td>000140</td>
<td>NICHOLLS</td>
<td>C01</td>
<td>28420.00</td>
<td>1976-12-15</td>
</tr>
</tbody>
</table>

**Notes:**

- The EXISTS predicate simply determines if something is true. If the EXISTS predicate is combined with other predicates in the outer query via AND, the outer query will only display a non empty final result if the condition in the subquery is true.
- The result of the EXISTS predicate is true if the result table returned by the subquery contains at least one row. Otherwise, the result is false.
- An EXISTS predicate can be negated by preceding it with the keyword NOT.
- The subquery in the example determines if there are any employees in the EMPLOYEE table that has a salary less than 25000. As indicated below the SQL statement, there is one such row, namely, the row for employee QUINTANA with employee number 000130 and a salary of 23800. The EXISTS predicate does not return this row. It only returns true to indicate that the row exists in table EMPLOYEE. If no row existed in table EMPLOYEE satisfying the WHERE condition of the subquery, false would be returned.

The EXISTS predicate is not the only predicate of the WHERE clause for the outer query, but is in this case true for all rows that qualify the other predicate(s) of the outer
query. The outer query returns employee number, last name, salary, and hiring date of all employees in the EMPLOYEE table that have a salary greater than 30000. The EXISTS effectively acts like an 'IF': a non empty result is only displayed if someone has a salary less than 25000.
Correlated Subquery (Part 1)

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>LASTNAME</th>
<th>WORKDEPT</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>000010</td>
<td>HAAS</td>
<td>A00</td>
<td>52750.00</td>
</tr>
<tr>
<td>000030</td>
<td>KWAN</td>
<td>C01</td>
<td>38250.00</td>
</tr>
<tr>
<td>000120</td>
<td>O'CONNELL</td>
<td>A00</td>
<td>29250.00</td>
</tr>
<tr>
<td>000130</td>
<td>QUINTANA</td>
<td>C01</td>
<td>23800.00</td>
</tr>
<tr>
<td>000140</td>
<td>NICHOLLS</td>
<td>C01</td>
<td>28420.00</td>
</tr>
</tbody>
</table>

Notes:

- In this example, the task is to find the employees who have a salary that is higher than the average salary of their department.

- Again, let us reformulate the problem:

  List employee number, last name, and salary of all employees that have a salary that is higher than the average salary of their corresponding department, that is, the department to which the employee belongs.

In other words, we do not want to find all employees, whose salary is higher than the average salary of any department or of all departments as was the case for the ANY or ALL predicate, respectively. This time, the employee's salaries must be matched with the average salaries of the departments to which the employees belong. That means that there is a correlation between an employee and his/her department that must be expressed in the subquery determining the average salaries.
Correlated Subquery (Part 2)

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>LASTNAME</th>
<th>WORKDEPT</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>000010</td>
<td>HAAS</td>
<td>A00</td>
<td>52750.00</td>
</tr>
<tr>
<td>000030</td>
<td>Kwan</td>
<td>C01</td>
<td>38250.00</td>
</tr>
<tr>
<td>000120</td>
<td>O’Connell</td>
<td>A00</td>
<td>29250.00</td>
</tr>
<tr>
<td>000130</td>
<td>Quintana</td>
<td>C01</td>
<td>23800.00</td>
</tr>
<tr>
<td>000140</td>
<td>Nicholls</td>
<td>C01</td>
<td>28420.00</td>
</tr>
</tbody>
</table>

EMPLOYEE

```
SELECT EMPNO, LASTNAME, SALARY
FROM EMPLOYEE
WHERE SALARY >
(SELECT AVG(SALARY)
FROM EMPLOYEE
WHERE WORKDEPT = WORKDEPT)
```

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>LASTNAME</th>
<th>SALARY</th>
<th>AVG(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>000010</td>
<td>Haas</td>
<td>52750.00</td>
<td>41000.000000</td>
</tr>
<tr>
<td>000030</td>
<td>Kwan</td>
<td>38250.00</td>
<td>30156.666666</td>
</tr>
</tbody>
</table>

Figure 5-13. Correlated Subquery (Part 2)

Notes:

- Noncorrelated subqueries execute the subquery once at the beginning and use the result to control the rows returned by the outer query. A correlated query works differently: a row is read by the outer query, then a value from that row is passed to the subquery and is used to control which rows are returned by the subquery. Then, the predicate in the outer query is used to determine if the outer table row will appear in the result set. This process is repeated for each row of the outer table until each one has been examined and either written to the result set or omitted from it.

- In the example, the outer query reads a row in the EMPLOYEE table and gets the employee number, last name, salary, and department (WORKDEPT). The (work) department for the employee is passed to the subquery. The subquery calculates the average salary for all employees in the work department that was passed from the outer row. Then, the result of the subquery is compared to the individual salary of the employee whose row was read by the outer query. If the individual salary exceeds the result of the subquery, the outer table row is written to the result set; otherwise, it is ignored. This process is repeated for each row of the outer table.
## Correlated Subquery with EXISTS

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>LASTNAME</th>
<th>WORKDEPT</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>000010</td>
<td>HAAS</td>
<td>A00</td>
<td>52750.00</td>
</tr>
<tr>
<td>000030</td>
<td>KWAN</td>
<td>C01</td>
<td>38250.00</td>
</tr>
<tr>
<td>000120</td>
<td>O'CONNELL</td>
<td>A00</td>
<td>29250.00</td>
</tr>
<tr>
<td>000130</td>
<td>QUINTANA</td>
<td>C01</td>
<td>23800.00</td>
</tr>
<tr>
<td>000140</td>
<td>NICHOLLS</td>
<td>C01</td>
<td>28420.00</td>
</tr>
<tr>
<td>000400</td>
<td>WILSON</td>
<td>NULL</td>
<td>25400.00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DEPTNO</th>
<th>DEPTNAME</th>
<th>MGRNO</th>
</tr>
</thead>
<tbody>
<tr>
<td>A00</td>
<td>SPIFFY COMPUTER SERVICE DIV.</td>
<td>000010</td>
</tr>
<tr>
<td>C01</td>
<td>INFORMATION CENTER</td>
<td>000030</td>
</tr>
<tr>
<td>D01</td>
<td>DEVELOPMENT CENTER</td>
<td>-</td>
</tr>
</tbody>
</table>

**SELECT DEPTNO, DEPTNAME FROM DEPARTMENT D WHERE NOT EXISTS**

(SELECT *
 FROM EMPLOYEE
 WHERE WORKDEPT = D.DEPTNO)

---

**Notes:**

- This visual shows use of the True/False indicator EXISTS in a correlated subquery.
  
  This is an alternative way to get the same result as on Figure 5-5.
Checkpoint

1. List three quantified predicates used with subqueries.

2. T/F. The EXISTS predicate can be negated.

3. Assume that you need to list the employees that have the highest salaries in their departments. What kind of subquery will you have to use?

Notes:
Write down your answers here:
1. 

2. 

3. 

You will find an example of the answers in the Appendix.
Summary

Now that you have completed this unit, you should be able to:

- Code subqueries using the ALL, ANY/SOME, and EXISTS keywords
- Code correlated subqueries
- Choose the proper type of subquery to use in each case

Notes:
Unit 6. Scalar Functions

What This Unit Is About

This unit provides information on how to use various common scalar functions.

What You Should Be Able to Do

After completing this unit, you should be able to:

- Code SQL statements that use various scalar functions on columns of the following data types:
  - SMALLINT, INTEGER, DECIMAL, FLOAT, CHAR, VARCHAR, DATE, TIME, TIMESTAMP
- Use the following scalar functions effectively:
  - SUBSTR()
  - COALESCE()
  - VALUE()
  - DECIMAL()
  - DIGITS()
  - CHAR()
  - LENGTH()
  - LTRIM(), RTRIM()
  - DATE(), YEAR(), MONTH(), DAY(), DAYS(), TIME(), HOUR(), MINUTE(), SECOND(), TIMESTAMP(), MICROSECOND()
  - DAYOFYEAR(), DAYOFWEEK(), WEEK(), JULIAN_DAY()
  - RAISE_ERROR(), ROUND()
- Perform calculations involving dates and date/time intervals.

How You Will Check Your Progress

Accountability:

- Unit Checkpoint
- Machine Lab
Labs

- QUERIES using
  — SUBSTR
  — VALUE/COALESCE
  — DECIMAL
  — DIGITS
  — DATE functions

References

DB2 UDB for OS/390 and z/OS V7
  SQL Reference (SC26-9944-02)

DB2 Administration Guide (DB2 UDB 7.2 for Windows)
DB2 SQL Reference (DB2 UDB 7.2 for Windows)
Objectives

After completing this unit, you should be able to:

- Code scalar functions for the data types:
  - SMALLINT, INTEGER, DECIMAL, FLOAT
  - CHAR, VARCHAR
  - DATE, TIME, TIMESTAMP

- Use the following scalar functions effectively:
  - SUBSTR(), COALESCE(), VALUE(), DECIMAL()
  - DIGITS(), CHAR(), LENGTH(), LTRIM(), RTRIM()
  - DATE(), YEAR(), MONTH(), DAY(), DAYS(), WEEK()
  - TIME(), HOUR(), MINUTE(), SECOND()
  - TIMESTAMP(), MICROSECOND()
  - ROUND(), TRUNCATE(), RAISE_ERROR()

- Perform calculations involving dates and date/time intervals

Notes:

- A function is an operation denoted by a function name followed by one or more arguments which are enclosed in parentheses. Most functions have a single argument. The result of a function is a single value. Applying the function to its arguments derives this single value.

- Functions are classified as scalar functions or column functions. The argument of a column function is a set of values from different rows. The argument of a scalar function is a single value.

- In this unit, we will discuss scalar functions. Column functions were discussed in the SQL Basics course and reviewed briefly at the beginning of this course.
6.1 Scalar Functions
Notes:
Scalar functions:

- Are easy to use
- Make it simple to format your output the way you want it
- Make it possible to do calculations involving dates, times, and timestamps
Scalar Function - SUBSTR

The SUBSTR() function is used to extract a portion of a string.

- **string**: identifies the column or literal from which the result is derived. *string* must be a character string or a graphic string. If *string* is a character string, the result of the function is a character string. If it is a graphic string, the result of the function is a graphic string.

- **start**: identifies the starting point at which the function will begin extracting data, that is, the position of the first character extracted. It must be a positive integer that is not beyond the end of *string*.

- **length**: identifies the number of characters that the function must extract from *string*. The value of *length* cannot cause the extraction process to go beyond the end of *string*.

- The example obtains information for departments called a center, that is, which name ends with the character string 'CENTER'. The desired information is the name of the department and the area to which the department belongs. The area is encoded in the first two characters of the department number. Therefore, the SUBSTR() function is
used to extract the leftmost two characters of the department number (\texttt{start=1, length=2}). The appropriate column in the result, i.e., the first column, is called \texttt{AREA}. The second column of the result is the name of the department.
Scalar Function - COALESCE/VALUE (Part 1)

Figure 6-4. Scalar Function - COALESCE/VALUE (Part 1)  

Notes:

- COALESCE() and VALUE() perform exactly the same function. They are synonymous with each other but COALESCE() is part of the ANSI/ISO standard and VALUE() is not. You should prefer to use the COALESCE() function and avoid any confusion with the VALUES keyword.

- The function returns the first argument of its argument list that is not null. If all arguments are null, the function returns a null. For example, if an imaginary table contained three numeric columns named SALARY, COMMISSION, and BONUS, the expressions

  \[
  \text{COALESCE}((\text{SALARY, COMMISSION, BONUS}) \text{ or VALUE}(\text{SALARY, COMMISSION, BONUS})
  \]

would return:

- SALARY if salary is not null
- COMMISSION if salary is null and commission is not null
- BONUS if salary and commission are both null and bonus is not null
- Null if salary, commission and bonus are all null

- The arguments must be compatible. Character string arguments are not compatible with datetime values. Thus, if any argument is a character string, all arguments must be character strings; if any argument is a date, all arguments must be dates; and so forth.

- The example displays the rows of the department table. If a department does not have a manager (MGRNO is null), the character string 'NONE' is displayed. The COALESCE function is used to achieve this. The appropriate column of the result set is named MANAGER.

- COALESCE/VALUE is like a special kind of CASE expression and the same query can be written:

```sql
SELECT DEPTNO, DEPTNAME,
  CASE WHEN MGRNO IS NULL THEN 'NONE'
    ELSE MGRNO
  END
AS MANAGER
FROM DEPT
```
Scalar Function - COALESCE/VALUE (Part 2)

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>LASTNAME</th>
<th>WORKDEPT</th>
<th>SALARY</th>
<th>COMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>000010</td>
<td>HAAS</td>
<td>A00</td>
<td>52750.00</td>
<td>4220.00</td>
</tr>
<tr>
<td>000030</td>
<td>KWAN</td>
<td>C01</td>
<td>38250.00</td>
<td>3060.00</td>
</tr>
<tr>
<td>000120</td>
<td>O'CONNELL</td>
<td>A00</td>
<td>29250.00</td>
<td>2340.00</td>
</tr>
<tr>
<td>000130</td>
<td>QUINTANA</td>
<td>C01</td>
<td>-</td>
<td>1904.00</td>
</tr>
<tr>
<td>000140</td>
<td>NICHOLLS</td>
<td>C01</td>
<td>28420.00</td>
<td>-</td>
</tr>
</tbody>
</table>

**SELECT EMPNO, COALESCE (SALARY+COMM, SALARY, COMM) AS TOT_SALARY FROM EMPLOYEE**

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>TOT SAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>000010</td>
<td>56970.00</td>
</tr>
<tr>
<td>000030</td>
<td>41310.00</td>
</tr>
<tr>
<td>000120</td>
<td>31590.00</td>
</tr>
<tr>
<td>000130</td>
<td>1904.00</td>
</tr>
<tr>
<td>000140</td>
<td>28420.00</td>
</tr>
</tbody>
</table>

- Prefer COALESCE and avoid confusion with the VALUES keyword!

**Notes:**

- This visual shows how the COALESCE/VALUE function can be useful in an addition.
- Observe that EMPNO 000130 has an unknown (NULL) salary and that EMPNO 000140 has an unknown (NULL) commission.
- The same query can also be written this way:

  ```sql
  SELECT EMPNO, COALESCE (SALARY, 0) + COALESCE (COMM, 0)
  FROM EMPLOYEE
  ```
Scalar Function - DECIMAL

The DECIMAL() function returns a decimal representation of the value in its argument in the format DECIMAL(p,s), where p and s are the second and third argument. p is the precision of the decimal number and s the scale, that is, the number of decimal places.

- The first argument must be a character string or a number. The second argument, if specified, must range in value from 1 to 31. The third argument, if specified, must range in value from 1 to p, where p is the value of the second argument.
- Default values for the second argument depend on the data type of the first argument.
- The default value for the third argument is zero.

The example on the visual calculates the average salary of the employees in department C01. The first value displayed (AVERAGE_1) in the result is the average as displayed by DB2 when only the AVG column function is used. The displayed value has scale of 8, that is, 8 decimal places. The second result column (AVERAGE_2) also displays the average value, but after the DECIMAL() function has been used to truncate to two decimal places.
If you do not want to *truncate* the average to two decimal places, but want to *round* it to two decimal places, you must add 0.005 to the average before applying the DECIMAL() function (see AVERAGE_3) or use the ROUND() function, see next page.

- There is also a scalar function called TRUNCATE() or TRUNC(). It returns its first argument to a precision determined by the second argument. When argument 2 is positive, argument 1 is truncated argument 2 places right of decimal point. When argument 2 is negative, argument 1 is truncated argument 2 places to the left of the decimal point.

The first argument can be any built-in numeric data type. The second argument has to be an INTEGER or SMALLINT.

Example:

```
SELECT TRUNCATE(AVG(SALARY),+2) AS TRUN_1, TRUNC(AVG(SALARY),-1) AS TRUN_2
FROM EMPLOYEE
WHERE WORKDEPT='C01'
```

<table>
<thead>
<tr>
<th>TRUN_1</th>
<th>TRUN_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>+3,01566600000000E+004</td>
<td>+3,01500000000000E+004</td>
</tr>
</tbody>
</table>

TRUNCATE() or TRUNC() is available on DB2 UDB for UNIX, Windows and OS/2 and DB2 UDB for OS/390 and z/OS.
Scalar Function - ROUND

ROUND (expression1, expression2)

EMPN0  LASTNAME  WORKDEPT  SALARY  HIREDATE
000010  HAAS      A00       52750.00  1965-01-01
000030  Kwan      C01       38250.00  1975-04-05
000120  O’Connell  A00       29250.00  1963-12-05
000130  Quintana  C01       23800.00  1971-07-28
000140  Nicholls  C01       28420.00  1976-12-15

SELECT AVG(SALARY) AS AVERAGE_1,
ROUND(AVG(SALARY),+2) AS AVERAGE_2,
ROUND(AVG(SALARY),-1) AS AVERAGE_3,
DECIMAL(ROUND(AVG(SALARY),-1),8,2) AS AVERAGE_4
FROM EMPLOYEE
WHERE WORKDEPT = 'C01'

AVERAGE_1   AVERAGE_2   AVERAGE_3   AVERAGE_4
30156.6666666666666666 +3.0156670000000000E+04 +3.0160000000000000E+04 30160.00
30156.6666666666666666 30156.6700000000000000 30160.0000000000 30160.00

Notes:

- The ROUND() function returns expression1 rounded to expression2 places right of the decimal point. If expression2 is negative, expression1 is rounded to the absolute value of expression2 places to the left of the decimal point.

- The result data type depends on the data type of expression1. The result data type is:
  - INTEGER if the first argument is INTEGER or SMALLINT.
  - In DB2 UDB for UNIX, Windows and OS/2: DOUBLE if the first argument is DOUBLE, DECIMAL or REAL.
  - In DB2 UDB for OS/390: DECIMAL if the first argument is DECIMAL.

- The result is null if any argument is null.

- The example on the visual calculates the average salary for the employees in department C01. The first value displayed (AVERAGE_1) in the result is the average as displayed by DB2 when only the AVG column function is used, as discussed on the previous page. The second result column (AVERAGE_2) also displays the average as displayed by DB2 when only the AVG column function is used, as discussed on the previous page.
value, but after the average has been rounded to two decimal places by means of the \texttt{ROUND()} function. The result is in \texttt{DOUBLE}.

The third result column (\texttt{AVERAGE\_3}) also displays the average value, but after it has been rounded to -1 decimal places. The fourth result column (\texttt{AVERAGE\_4}) also displays the average value after it has been rounded to -1 decimal places. This time the function \texttt{DECIMAL} has been used to limit the output from \texttt{ROUND}. 
Scalar Function - DIGITS

The DIGITS() function returns a character string representation of its argument.

The argument must be an integer or a decimal number. If the argument is null, the function returns null.

The result of the function is a fixed-length character string representing the value of the argument without a sign or a decimal point. Leading zeroes will be added if necessary to bring the value up to a minimum length. The length of the string is 5 for small integers, 10 for integers or p for decimal numbers with a precision of p.

The example displays the salary of employee NICHOLLS in two forms:

- The first column of the result (SALARY_1) shows the salary expressed as a character string that contains no decimal point.
- The second column of the result (SALARY_2) shows the integer portion of the salary by extracting the appropriate portion from the value for the first column by means of the SUBSTR() function. This demonstrates that the value returned by the

Notes:

- The DIGITS() function returns a character string representation of its argument.
- The argument must be an integer or a decimal number. If the argument is null, the function returns null.
- The result of the function is a fixed-length character string representing the value of the argument without a sign or a decimal point. Leading zeroes will be added if necessary to bring the value up to a minimum length. The length of the string is 5 for small integers, 10 for integers or p for decimal numbers with a precision of p.
- The example displays the salary of employee NICHOLLS in two forms:
DIGITS() function is indeed a character string as required by the SUBSTR() function.

**Note:** Since the SUBSTR() function only works on character strings, the above example demonstrates a technique frequently used to extract portions of numbers from numeric columns.
Notes:

- The CHAR() function returns a character-string representation of an expression.
- This first visual shows how to convert a datetime value to a different format.
- When the first argument is a date, time or timestamp, the second argument is the name of a datetime format. The valid datetime formats, and an example of a date and a time expressed in each format, are as follows:

<table>
<thead>
<tr>
<th>Format</th>
<th>Sample Date</th>
<th>Sample Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISO</td>
<td>1998-03-27</td>
<td>20.09.47</td>
</tr>
<tr>
<td>USA</td>
<td>03/27/1998</td>
<td>08:09 PM</td>
</tr>
<tr>
<td>EUR</td>
<td>27.03.1998</td>
<td>20.09.47</td>
</tr>
<tr>
<td>JIS</td>
<td>1998-03-27</td>
<td>20:09:47</td>
</tr>
<tr>
<td>Local</td>
<td>depends</td>
<td>depends</td>
</tr>
</tbody>
</table>
• The result of the function is a fixed-length character string.

• If the first argument is a date or a time, the result is the character string representation of that date or time in the format specified by the second argument.

• If the first argument is a timestamp, the second argument is not applicable and must be omitted.

• If the second argument is omitted, the representation of a date or time is in the default format defined during DB2 installation.

• The example displays the hiring date of employee QUINTANA (employee number 000130) in three different formats: the ISO format (ISO_DATE), the US format (USA_DATE), and the European format (EUR_DATE).
Notes:

- This visual shows the syntax for function CHAR() on different data types.
- It returns a FIXED-LENGTH character-string representation of:
  - Character string value if the first argument is any type of character string
  - Integer number if the first argument is a SMALLINT or INTEGER
  - Decimal number if the first argument is a decimal number
  - Double-precision floating-point number if the first argument is a DOUBLE or REAL
  - Datetime value if the first argument is a date, time or timestamp
- If the first argument can be null, the result can be null. If the first argument is null, the result is null.
- The example shows how the decimal-character can be changed in a query to a ',' instead of the default '. '. As shown on the visual, the result includes \( p \) digits, where \( p \) is the precision of the decimal-expression (with a preceding minus sign if the argument is negative).
Scalar Function - LENGTH

Figure 6-11. Scalar Function - LENGTH

Notes:

- The LENGTH() function returns the length of its argument.
- The argument can be a value of any data type.
- The result of the function is an integer, unless the argument is a null, in which case the result is null.
- If the argument is a fixed-length character string, the result is the maximum width of the column as defined when the table was created.
- If the argument is a variable-length character string, the result is the actual length of the specific value being read from the column.
- The example on the visual obtains the lengths of some columns for employee O'CONNELL (employee number 000120). The first column of the result (LENGTH_LASTNAME) displays the length of the employee's last name. The column LASTNAME of the EMPLOYEE table has been defined as VARCHAR. Therefore, the length returned specifies the actual number of characters stored for the last name. If the
last name was stored with trailing blanks, these would be counted as well. Apparently, the name O'CONNELL has been stored without trailing blanks.

The second column of the result (LENGTH_SALARY) shows the internal length of the SALARY column which has been defined as DECIMAL(9,2) in the EMPLOYEE table. Internally, decimal numbers are stored in such a way that two digits occupy a byte. In addition, half a byte is needed for the sign of the decimal number. Of course, the decimal number must occupy a full number of bytes. If \( p \) is the precision of the decimal column, the number of bytes required is \( \text{INTEGER}(p/2)+1 \). For our example, it is \( \text{INTEGER}(9/2)+1 = 5 \), as shown in the result.

The third column of the result (DIGITS_SALARY) shows the number of digits for the SALARY column, that is, its precision. The DIGITS() function is first used to convert the salary of employee O'CONNELL into a character string without sign or decimal point, but with leading zeros to bring the length of the character string to the precision for the column. Thereafter, the LENGTH() function is applied to this character string and returns the precision for the column.
Scalar Functions - LTRIM / RTRIM

The LTRIM (Left TRIM) and RTRIM (Right TRIM) functions remove leading or trailing blanks in an expression.

If the argument is null, the result is null.

The example can be done with the following SQL statement without changing the EMPLOYEE table:

```sql
WITH EMP (EMPNO, LASTNAME, WORKDEPT, SALARY, HIREDATE) AS
  (SELECT TRANSLATE(SUBSTR(EMPNO,1,5),'  ','0')||'0' AS EMPNO, LASTNAME, WORKDEPT, SALARY, HIREDATE FROM EMPLOYEE
   WHERE EMPNO IN ('000010','000030','000120','000130','000140'))
SELECT CHAR(LTRIM(EMPNO),10) AS STRIPPED_EMPNO,
     LENGTH(RTRIM(LASTNAME)) AS LENGTH_NAME
FROM EMP WHERE EMPNO IN ('    30','   120','   140');
```

Notes:

- The LTRIM (Left TRIM) and RTRIM (Right TRIM) functions remove leading or trailing blanks in an expression.
- If the argument is null, the result is null.
- The example can be done with the following SQL statement without changing the EMPLOYEE table:
The example displays information for the employees with the employee numbers 30, 120, and 140. The employee number, EMPNO, has been changed to leading blanks in the EMPLOYEE table. The first column of the result (STRIPPED_EMPNO) contains the employee numbers of the employees where the leading blanks have been removed. The length of result column STRIPPED_EMPNO has been set to 10 in the CHAR() function. Remember that the EMPNO column has been defined as CHAR(6) in the EMPLOYEE table.

The second column contains the lengths of the last names when any trailing blanks have been stripped away.

On DB2 UDB OS/390, there is also a very useful function, STRIP().

- The function STRIP() removes leading, trailing, or both leading and trailing occurrences of a specified strip-character from the value provided by the first argument.

- The first argument must be a string expression.

- The second argument indicates whether characters are removed from the beginning, the end, or both the beginning and the end of the string provided by the first argument. The default for the second argument is to remove the characters from both ends of the string.

- The third argument indicates the character to be removed. The character must be enclosed in apostrophes. The default value of the third argument is the blank (" ") character.

Note: The STRIP() function is only available in DB2 UDB OS/390. If you want to have a function like STRIP() on other platforms, you could write your own user-defined function (UDF) that would behave any way you like.
Date-Related Scalar Functions (Part 1)

- \textbf{DATE}(\textit{expression})
- \textbf{YEAR}(\textit{expression})
- \textbf{MONTH}(\textit{expression})
- \textbf{DAY}(\textit{expression})
- \textbf{DAYS}(\textit{expression})

\textit{Notes:}

- The date-related functions include \textbf{DATE()}, \textbf{YEAR()}, \textbf{MONTH()}, \textbf{DAY()}, \textbf{DAYS()}, \textbf{DAYOFYEAR()}, \textbf{DAYOFMONTH()}, \textbf{DAYOFWEEK} and \textbf{JULIAN\_DAY}. The first four functions extract a portion of their argument, namely, the full date, the year, the month, or the day, respectively. The \textbf{DAYS()} function also works with dates, but in a different way than the other date functions.

\textit{Note:} The \textbf{DAYS()} function is NOT the same as the \textbf{DAY()} function, despite the similarity of the names.

- The \textbf{DAYS()} function determines the difference between the date in the argument and the date of January 1, 0001 and expresses the difference in days.

- We can determine the difference between two dates by subtracting one date from the other date. The difference, which is expressed as a decimal number with no decimal places, represents the number of years, months and days that separate the two dates. If the first date in the expression is later than the second date in the expression, the difference is preceded by a minus sign.
A date in a table can also be subtracted from a literal value representing a valid date or vice versa. The literal need not be the argument of a DATE function, but can be. However, a literal representing a date cannot be subtracted from another literal representing a date.
Date-Related Scalar Functions (Part 2)

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>LASTNAME</th>
<th>WORKDEPT</th>
<th>SALARY</th>
<th>HIREDATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>000010</td>
<td>HAAS</td>
<td>A00</td>
<td>52750.00</td>
<td>1965-01-01</td>
</tr>
<tr>
<td>000030</td>
<td>KWAN</td>
<td>C01</td>
<td>38250.00</td>
<td>1975-04-05</td>
</tr>
<tr>
<td>000120</td>
<td>O'CONNELL</td>
<td>A00</td>
<td>29250.00</td>
<td>1963-12-05</td>
</tr>
<tr>
<td>000130</td>
<td>QUINTANA</td>
<td>C01</td>
<td>23800.00</td>
<td>1971-07-28</td>
</tr>
<tr>
<td>000140</td>
<td>NICHOLLS</td>
<td>C01</td>
<td>28420.00</td>
<td>1976-12-15</td>
</tr>
</tbody>
</table>

SELECT HIREDATE, CHAR(HIREDATE,EUR) AS EUR_DATE,  
DATE(CHAR(HIREDATE,EUR)) AS DATE,  
YEAR(HIREDATE) AS YEAR,  
MONTH(HIREDATE) AS MONTH,  
DAY(HIREDATE) AS DAY,  
DAYS(HIREDATE) AS DAYS,  
DATE('2005-04-04')-HIREDATE AS PERIOD,  
DAYS('2005-04-04')-DAYS(HIREDATE) AS PERIOD IN DAYS,  
HIREDATE+('01/01/2000'-HIREDATE) AS Y2K  
FROM EMPLOYEE  
WHERE EMPNO = '000030'

<table>
<thead>
<tr>
<th>HIREDATE</th>
<th>EUR_DATE</th>
<th>DATE</th>
<th>YEAR</th>
<th>MONTH</th>
<th>DAY</th>
<th>DAYS</th>
<th>PERIOD</th>
<th>PERIOD IN DAYS</th>
<th>Y2K</th>
</tr>
</thead>
<tbody>
<tr>
<td>1975-04-05</td>
<td>05.04.1975</td>
<td>1975</td>
<td>4</td>
<td>5</td>
<td>721083</td>
<td>291129</td>
<td>10957</td>
<td>1999-12-31</td>
<td></td>
</tr>
</tbody>
</table>

Notes:

- The example on the visual uses the hiring date of employee KWAN to demonstrate the date-related scalar functions and perform some calculations with dates.
- The first two columns of the result (HIREDATE and EUR_DATE) show the hiring date when displaying column HIREDATE and when converting it to the European format by means of the CHAR() function. As you can see from the first column, the default format established for the system during DB2 installation is the ISO format.
- The third column of the result (DATE), applies the DATE() function to the character string, the European date, returned by the CHAR() function. It converts the character string again into a date which is displayed in the default date format for the system, that is, the ISO format.
- The fourth (YEAR), fifth (MONTH), and sixth (DAY) columns extract the year, the month, and the day from the hiring date of employee KWAN by means of the YEAR(), MONTH(), or DAY() function, respectively. You should realize that leading zeros are removed.
The seventh column of the result (DAYS) presents the hiring date of employee KWAN as the number of days since January 1, 0001, by using the DAYS() function. The value returned by the DAYS() function is an integer.

The column PERIOD of the result is derived by first converting the character string ‘2005-04-04’ in a date by means of the DATE() function and, then, subtracting the hiring date of the employee. The result is a decimal date duration that specifies the number of years, month, and days between the two dates in this order. The data type of the result is DECIMAL(8,0). Because leading zeros are suppressed in the displayed result, the displayed value has to be interpreted from right to left. A value of ‘608.’ means that the duration is 6 months and 8 days. No years are shown in this case.

The result is a little bit surprising. Because the hiring date of employee KWAN is ‘1975-04-05’, you might have expected a value of ‘291130.’. April 4, 2004, is just one day less than 30 years from April 5, 1975. Since 30 years is equivalent to 29 years, 11 months, and 31 days, 30 years minus one day should be 29 years, 11 months, and 30 days, that is, a decimal date duration of ‘291130.’.

However, this is not the way DB2 calculates the difference between two dates. If DATE1 and DATE2 are two dates which you want to subtract, the following procedural description clarifies the steps involved in the operation RESULT = DATE1-DATE2:

**Step 1**

If DAY(DATE2) <= DAY(DATE1)

then \[ \text{DAY(RESULT)} = \text{DAY(DATE1)} - \text{DAY(DATE2)}. \]

else

\[
\text{DAY(RESULT)} = \text{N+DAY(DATE1)} - \text{DAY(DATE2)} \\
\text{where N = the last day of MONTH(DATE2).} \\
\text{MONTH(DATE2) = MONTH(DATE2)+1.}
\]

end

**Step 2**

If MONTH(DATE2) <= MONTH(DATE1)

then \[ \text{MONTH(RESULT)} = \text{MONTH(DATE1)} - \text{MONTH(DATE2)}. \]

else

\[
\text{MONTH(RESULT)} = 12+\text{MONTH(DATE1)} - \text{MONTH(DATE2)}. \\
\text{YEAR(DATE2) = YEAR(DATE2)+1.}
\]

end

**Step 3**

\[ \text{YEAR(RESULT)} = \text{YEAR(DATE1)} - \text{YEAR(DATE2)}. \]

For our example, DATE1 is ‘2005-04-04’ and DATE2 is ‘1975-04-05’. Since DAY(DATE2) is 5 which is greater than DAY(DATE1), DAY(RESULT) is N+4-5 where N
is the last day of MONTH(DATE2), that is, the last day of April which is 30. Accordingly, the day for the result is 29 as shown on the visual.

This shows that dates should not be subtracted because normal mathematical rules do not apply for date calculations. Rather, we should calculate the difference in days as done for column PERIOD_IN_DAYS of the result.

- The column PERIOD_IN_DAYS shows the number of days you get by first converting '2005-04-04' and the hiring date of employee KWAN into the respective number of days since January 1, 0001, and then subtracting the two integers obtained from the DAYS() function. For this calculation, the normal mathematical rules apply because integers are subtracted.

- The last column of the result (Y2K) adds the decimal date duration, resulting from subtracting the hiring date of the employee from the date '2000-01-01', to the hiring date of employee KWAN. At the first glance, you would expect the result to be the date '2000-01-01' as simple mathematical rules for adding and subtracting numbers suggest. But beware of this trap. You are not performing an integer or decimal calculation, but a date calculation. By subtracting the hiring date of the employee from '2000-01-01', you obtain a decimal date duration for which the algorithm mentioned for the PERIOD column of the result applies. The appropriate decimal date duration is '240826.'. Adding this date duration to the hiring date (1975-04-05) results in date of '1999-12-31', and not '2000-01-01', because DB2 adds the years of the duration to the year of the date, the months of the duration to the month of the date, and the days of the duration to day of the date. (Of course, overflows are taken into account.)

This underlines again how careful you must be with date or date-duration calculations. They just do not follow the normal number arithmetic. To get the expected result, you have to calculate the difference between the two dates in days, by applying the DAYS() function to both dates and subtracting the results, and then add the difference to the hiring date. However, labeled durations are required for this. We will discuss labeled durations later in this unit.
Date-Related Scalar Functions (Part 3)

Notes:

- This visual lists some additional data-related scalar functions. The DAYOFYEAR function returns the day of the year in argument as an integer value in the range 1-366.
- The WEEK() function returns the week of the year of the argument as an integer value in the range 1-54. The week starts with Sunday.
- The DAYOFWEEK() function returns the day of the week in the argument as an integer value in the range 1-7, where 1 represents Sunday.
- The WEEK_ISO() function returns the week of the year of the argument as an integer value in range 1-53. The week starts with Monday. Week 1 is the first week of the year to contain a Thursday, which is equivalent to the first week containing January 4.
- The DAYOFWEEK_ISO() function returns the day of the week in the argument as an integer value in the range 1-7, where 1 represents Monday.
- The JULIAN_DAY() function returns an integer value representing a continuous count of days and fractions since noon Universal Time on January 1, 4713 BCE (on the Julian calendar) to the date value specified in the argument.
• For DAYOFYEAR, DAYOFWEEK, DAYOFWEEK_ISO, WEEK, WEEK_ISO and JULIAN_DAY the argument must be a date, timestamp, or valid character string representation of a date or timestamp that is neither a CLOB nor a LONG VARCHAR. If the argument is null, the result is null.
Figure 6-16. Date-Related Scalar Functions (Part 4)

Notes:

- The example on the visual uses the hiring date of employee KWAN to demonstrate some more date-related scalar functions. Calculations with dates.

- The first column of the result (HIREDATE) shows the hiring date with default format as previously.

- The second column of the result (DAYOFYEAR), applies the function DAYOFYEAR() which returns the day off the year as an integer. HAAS' hiredate 1965-01-01 is the first day of the year.

- The third column of the result (DAYOFWEEK), applies the function DAYOFWEEK() which returns the day of the week in the argument as an integer where 1 represents Sunday. (HAAS was hired on a Friday.)

- The fourth column of the result (DAYOFWEEK_ISO), applies the function DAYOFWEEK_ISO() which returns the day of the week in the argument as an integer where 1 represents Monday. (HAAS was still hired on a Friday.)
- The fifth column of the result (DAYNAME), applies the function DAYNAME() which returns the name of the weekday. (It confirm that HAAS was hired on a Friday.) Note that the scalar function called DAYNAME() is only available on DB2 UDB for UNIX, Windows and OS/2. It returns a mixed case character string containing the name of the day (for example, Friday) for the day portion of the argument.

- The sixth column of the result (WEEK), applies the function WEEK() which returns the week of the year as an integer. The week starts with Sunday. HAAS' hiredate 1965-01-01 is in the first week of the year.

- The seventh column of the result (WEEK_ISO), applies the function WEEK_ISO() which returns the week of the year as an integer. The week starts with Monday. HAAS' hiredate 1965-01-01 is week 53 of the previous year.

- The eighth column of the result (JULIAN_DAY), applies the function JULIAN_DAY() which returns an integer value representing the number of days from January 1, 4713 B.C. (on the Julian calendar).
Time/Timestamp-Related Scalar Functions (Part 1)

Notes:

- The time and timestamp related scalar functions include TIME(), HOUR(), MINUTE(), SECOND(), MICROSECOND(), and TIMESTAMP(). The first five functions extract a portion of their argument, namely, the time, the hour, the minute, the second, or the microseconds, respectively.

  The TIMESTAMP() function returns a timestamp derived from its arguments. The TIMESTAMP() function allows you to specify one or two arguments. If only one argument is specified, this argument must represent a timestamp. If two arguments are specified, the first argument must represent a date and the second a time. The returned timestamp then contains the date specified by the first argument and the time provided by the second argument.

- You can determine the difference between two times or two timestamps by subtracting one from the other. When we subtract one time from another time, a decimal time duration results, which specifies the number of hours, minutes and seconds between the two times. The data type of the result is DECIMAL(6,0). When we subtract one timestamp from another timestamp, a decimal timestamp duration results, which
specifies the number of years, months, days, hours, minutes, seconds and microseconds between the two timestamps. The data type of the result is DECIMAL(20.6). For the special rules that apply when subtracting times or timestamps, see the SQL Reference.

- A time in a table can be subtracted from a literal value representing a valid time or vice versa. The literal need not be the argument of a TIME function, but can be. However, a literal representing a time cannot be subtracted from a literal representing a time. A similar rule applies to timestamps.
Time/Timestamp-Related Scalar Functions (Part 2)

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>LASTNAME</th>
<th>WORKDEPT</th>
<th>SALARY</th>
<th>HIREDATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>000010</td>
<td>HAAS</td>
<td>A00</td>
<td>52750.00</td>
<td>1965-01-01</td>
</tr>
<tr>
<td>000030</td>
<td>KWAN</td>
<td>C01</td>
<td>38250.00</td>
<td>1975-04-05</td>
</tr>
<tr>
<td>000120</td>
<td>O'CONNELL</td>
<td>A00</td>
<td>29250.00</td>
<td>1963-12-05</td>
</tr>
<tr>
<td>000130</td>
<td>QUINTANA</td>
<td>C01</td>
<td>23800.00</td>
<td>1971-07-28</td>
</tr>
<tr>
<td>000140</td>
<td>NICHOLLS</td>
<td>C01</td>
<td>28420.00</td>
<td>1976-12-15</td>
</tr>
</tbody>
</table>

SELECT AC2001010100000000 AS TIMESTAMP, 
    TIME(TIMESTAMP(HIREDATE,'08:15:35')) AS TIME, 
    HOUR(TIMESTAMP(HIREDATE,'08:15:35')) AS HOUR, 
    MINUTE(TIMESTAMP(HIREDATE,'08:15:35')) AS MINUTE, 
    SECOND(TIMESTAMP(HIREDATE,'08:15:35')) AS SECOND, 
    MICROSECOND(TIMESTAMP(HIREDATE,'08:15:35')) AS MICRO, 
    (TIMESTAMP('20000101000000000') - TIMESTAMP(HIREDATE,'08:15:35')) AS UNTIL_Y2K
FROM EMPLOYEE
WHERE EMPNO = '000030'

<table>
<thead>
<tr>
<th>TIMESTAMP</th>
<th>TIME</th>
<th>HOUR</th>
<th>MINUTE</th>
<th>SECOND</th>
<th>MICRO</th>
<th>UNTIL_Y2K</th>
</tr>
</thead>
<tbody>
<tr>
<td>1975-04-05 08.15.35.000000</td>
<td>08.15.35</td>
<td>8</td>
<td>15</td>
<td>35</td>
<td>0</td>
<td>240825154425.000000</td>
</tr>
</tbody>
</table>

Notes:

- The first column of the result (TIMESTAMP) displays the timestamp created by combining the hiring date of employee KWAN with the time '08:15:35' to indicate that the employee was hired on April 5, 1975, at 08:15:35. When using the TIMESTAMP() function with two arguments, the first argument must represent a date and the second a time. Therefore, no microseconds can be provided and the displayed microsecond value is '000000'.

- The columns TIME, HOUR, MINUTE, SECOND, and MICRO extract portions of the timestamp displayed in the first column by using the functions TIME(), HOUR(), MINUTE(), SECOND(), and MICROSECOND(), respectively.

- The last column of the result (UNTIL_Y2K) shows the subtraction of two timestamps. The result is a decimal timestamp duration. Its data type is DECIMAL(20,6). Its external format is yyyymmdhhmmss.nnnnnn expressing, from left to right, years, months, days, hours, minutes, seconds, and microseconds (decimal places). Leading zeros are suppressed. Because a decimal timestamp duration includes a decimal date duration, the same problems exist as described for decimal date durations.
Labeled Durations

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>LASTNAME</th>
<th>WORKDEPT</th>
<th>SALARY</th>
<th>HIREDATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>000010</td>
<td>HAAS</td>
<td>A00</td>
<td>52750.00</td>
<td>1965-01-01</td>
</tr>
<tr>
<td>000030</td>
<td>KWAN</td>
<td>C01</td>
<td>38250.00</td>
<td>1975-04-05</td>
</tr>
<tr>
<td>000120</td>
<td>O'CONNELL</td>
<td>A00</td>
<td>29250.00</td>
<td>1963-12-05</td>
</tr>
<tr>
<td>000130</td>
<td>QUINTANA</td>
<td>C01</td>
<td>23800.00</td>
<td>1971-07-28</td>
</tr>
<tr>
<td>000140</td>
<td>NICOLLS</td>
<td>C01</td>
<td>28420.00</td>
<td>1976-12-15</td>
</tr>
</tbody>
</table>

SELECT HIREDATE,
HIREDATE + 24 YEARS + 11 MONTHS + 30 DAYS AS DATE_1,
HIREDATE - 5 DAYS - 3 MONTHS - 1 YEAR AS DATE_2,
CURRENT DATE AS TODAY
FROM EMPLOYEE
WHERE EMPNO = '000120'

<table>
<thead>
<tr>
<th>HIREDATE</th>
<th>DATE_1</th>
<th>DATE_2</th>
<th>TODAY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1963-12-05</td>
<td>1988-12-05</td>
<td>1962-08-30</td>
<td>1998-11-16</td>
</tr>
</tbody>
</table>

SELECT HIREDATE + (2 YEARS + 3 MONTHS + 5 DAYS)
FROM EMPLOYEE
WHERE EMPNO = '000010'

Notes:

- Datetime values can be added and subtracted. These operations involve decimal numbers called durations. A duration is a number representing an interval of time.

- A duration may be expressed as a decimal, as seen on the previous visuals, or as a labeled duration. '2 YEARS' is an example of a labeled duration.

- A labeled duration represents a specific unit of time as expressed by an integer followed by one of the keywords YEAR(S), MONTH(S), DAY(S), HOUR(S), MINUTE(S), SECOND(S), or MICROSECOND(S). The number specified is converted as if it were assigned to a DECIMAL(15,0) number.

- The sequence of operations is important when adding or subtracting labeled durations. To ensure consistent, predictable results, it is important to follow these rules:
  - When adding labeled durations, add years, then months, then days
  - When subtracting labeled durations, subtract days, then months, then years

- DB2 has some special registers which can be very useful when working with durations. These are CURRENT DATE, CURRENT TIME, CURRENT TIMESTAMP, and
CURRENT TIMEZONE. On some platforms, these registers may be identified as CURRENT_DATE, CURRENT_TIME, etc.

- The first SELECT statement on the visual obtains four columns of date information for employee O’CONNELL. The first column of the result shows the hiring date of the employee. The second column (DATE_1) is derived by adding three labeled durations to the hiring date. The third column (DATE_2) is derived by subtracting three labeled durations from the hiring date. The fourth column displays the current date which is 1998-11-16 because the query was run on November 16, 1998.

- The crossed out SELECT statement on the visual tries to add three labeled durations together, within parentheses, and then add the sum to the hiring date of an employee. A labeled duration can only be added or subtracted from a DATE, TIME, or TIMESTAMP: adding labeled durations together is an error. Therefore, the query fails.
Checkpoint

1. Which function can be used to extract the first three characters of an alphanumerical column?

2. Which function(s) can be used to extract the digits after the decimal point in a column defined as DECIMAL?

3. What is a timestamp?

4. What does the DAYS() function do?

Notes:

Write down your answers here:

1. _______________________________________________________________

2. _______________________________________________________________

3. _______________________________________________________________

4. _______________________________________________________________

You will find an example of the answers in the Appendix.
Summary

Now that you have completed this unit, you should be able to:

- Code scalar functions for the data types:
  - SMALLINT, INTEGER, DECIMAL, FLOAT
  - CHAR, VARCHAR
  - DATE, TIME, TIMESTAMP

- Use the following scalar functions effectively:
  - SUBSTR(), COALESCE(), VALUE(), DECIMAL()
  - DIGITS(), CHAR(), LENGTH(), LTRIM(), RTRIM()
  - DATE(), YEAR(), MONTH(), DAY(), DAYS(), WEEK()
  - TIME(), HOUR(), MINUTE(), SECOND()
  - TIMESTAMP(), MICROSECOND()
  - ROUND(), TRUNCATE(), RAISE_ERROR()

- Perform calculations involving dates and date/time intervals

Notes:
Unit 7. Table Expressions and Recursive SQL

What This Unit Is About

This unit provides information on how to code nested and common table expressions and recursive SQL statements. Table expressions are compared to views, and some coding techniques for avoiding excessive recursion or looping for recursive SQL are discussed.

What You Should Be Able to Do

After completing this unit, you should be able to:

- Identify reasons for using table expressions and recursive SQL
- Use nested and common table expressions
- Identify the difference between views and table expressions
- Code recursive SQL
- Control the depth of recursion when coding recursive SQL

How You Will Check Your Progress

Accountability:

- Unit Checkpoint
- Machine Lab

Labs

- Nested table expressions
- Common table expressions
- Recursive SQL

References

DB2 UDB for OS/390 and z/OS V7
SQL Reference (SC26-9944-02)

DB2 Administration Guide (DB2 UDB 7.2 for Windows)
DB2 SQL Reference (DB2 UDB 7.2 for Windows)
Objectives

After completing this unit, you should be able to:

- Identify reasons for table expressions and recursive SQL
- Use nested and common table expressions
- Identify the difference between views and table expressions
- Code recursive SQL
- Control the depth of recursion when coding recursive SQL

Notes:

- Some business problems that once required complex application programming or the extensive use of views can now be solved with far less effort via table expressions and recursive SQL.
7.1 Table Expressions and Recursive SQL
Nested Table Expressions

Application programmers can use Table Expressions to define and use “pseudo-views” or “inline-views”.

You do not have to create a regular view.

Regular views require you to create them and to control who can use them, which can be seen as negative. However, on the positive side, you can use a view to create a logical table that users cannot change.

You do not create Table Expressions separately and they are not documented in the DB2 Catalog tables. You cannot control access to a table expression itself by grant/revoke, the table expression is just defined in the query which uses it and the access given on the underlaying tables/views is checked.

In the example, everything in the parentheses following the first keyword FROM is a table expression. The name of the table expression is S. The table expression is obtaining data from the EMPLOYEE table. For each employee, it determines the
decade in which the employee was hired and the employee's salary. The result columns for the table expression are called HIREDECADE and SALARY.

The column HIREDECADE contains the decade, during which the employee was hired. For example, 1970 - 9 means that the employee was hired between January 1, 1970 and December 31, 1979. The hiring decade is determined by extracting the first three digits from the ISO format of the hiring date and appending the character string '0 - 9'. The column SALARY is the salary for the employee.

For each hiring decade, the outer query determines the minimum salary paid to employees hired during this decade. The outer query obtains its data from the table expression by grouping the rows returned by the table expression by hiring decade and determining the minimum salary for each distinct hiring decade. The appropriate columns of the final result table are called HIREDECADE and MINIMUM_SALARY.

On DB2 OS/390, the query of the example could not be done by immediately accessing the EMPLOYEE table and not using a table expression. This is because in DB2 OS/390 only column names can be specified in the GROUP BY clause, and not expressions.

- On DB2 UDB for UNIX, Windows and OS/2, an expression or a scalar function can be used in the GROUP BY clause.
- Nested table expressions are available on DB2 UDB UNIX, Windows and OS/2, DB2 UDB AS/400 and DB2 UDB OS/390.

Common tables expressions and recursive SQL are available on DB2 UDB for UNIX, Windows and OS/2.

**Note:** The AS clause is used for two different purposes in this example in order to create temporary names that can be referenced elsewhere in the query: It is used to name the derived column HIREDECADE of the table expression and also to name the table expression itself. The columns of the table expression are referenced by the outer query. The outer query uses the AS clause as well to name the calculated column containing the minimum salary for the hiring decade.
**Notes:**

- An operand of a join can be more complex than the name of a single table. You can use a fullselect in parentheses followed by a correlation name. This is called a nested table expressions.

- In the example on the visual, we want to list, for each employee in the EMPLOYEE table, his/her employee number, last name, hiring decade, salary, and the minimum salary being paid to any employee hired during the same decade. This is illustrated by the result table on the visual.

The result cannot be obtained by only using the EMPLOYEE table. It can be created by means of an inner join of two table expressions. As indicated on the visual, both table expressions must have a column HIREDECADE, and the inner join is performed by using the columns in the ON condition.
One of the table expressions (on the visual, the first one) is called $E$ and must provide employee number, last name, salary, and hiring decade for all employees. The other table expression, called $M$, must provide the minimum salary for each hiring decade. Joining the two table expressions by means of their HIREDECADE columns yields the desired result. The subsequent visual illustrates the result sets of the two table expressions.
Nested Table Expressions in Joins (Part 2)

```
SELECT E.EMPNO, E.LASTNAME, E.HIREDECADE, E.SALARY, M.MINIMUM_SALARY
FROM
  (                          EMPNO  LASTNAME  SALARY  HIREDECADE
    000010  HAAS  52750.00  1960 - 9
    000030  KWAN  38250.00  1970 - 9
    00120   O'CONNELL  29250.00  1960 - 9
    00130   QUINTANA  23800.00  1970 - 9
    000140  NICHOLLS  28420.00  1970 - 9
  ) AS E
INNER JOIN
  (                          HIREDECADE  MINIMUM_SALARY
    1960 - 9  29250.00
    1970 - 9  23800.00
  ) AS M
ON E.HIREDECADE = M.HIREDECADE
```

**Notes:**

- This visual illustrates the temporary result tables to be created by the subselects of the table expressions.
- The first table expression, called \( E \), should provide, for each employee, his/her employee number, last name, salary, and hiring decade (column \( \text{HIREDECADE} \)).
- The second table expression, called \( M \), should supply, for each decade, the minimum salary being paid to employees hired during this decade. This is precisely the same result table as for the example on page.
- By joining the two tables, we will get the desired result.
Notes:

- The visual shows the SELECT statement for retrieving employee number, last name, salary, and hiring decade for all employees.

- The hiring decade is determined in the same way as for the first example of the unit to ensure that the HIREDATE columns of the two table expressions can be used for the inner join operation of the outer query. The hiring decade is derived by appending the character string '0 - 9' to the first three digits of the year when the appropriate employee was hired.

- An AS clause is used to give the derived column the name HIREDECADE.
Nested Table Expressions in Joins (Part 4)

```
SELECT E.EMPNO, E.LASTNAME, E.HIREDECADE, E.SALARY, M.MINIMUM_SALARY
FROM
  (SELECT EMPNO, LASTNAME, SALARY, SUBSTR(CHAR(HIREDATE,ISO),1,3) CONCAT '0 - 9' AS HIREDECADE
   FROM EMPLOYEE
  ) AS E
INNER JOIN
  (SELECT S.HIREDECADE, MIN(S.SALARY) AS MINIMUM_SALARY
   FROM
     (SELECT SUBSTR(CHAR(HIREDATE,ISO),1,3) CONCAT '0 - 9' AS HIREDECADE, SALARY
      FROM EMPLOYEE
     ) AS S
   GROUP BY S.HIREDECADE
  ) AS M
ON E.HIREDECADE = M.HIREDECADE
```

**Notes:**

- Now, we can complete the original problem, that is, determine, for each employee, his/her employee number, last name, hiring decade, salary, and the minimum salary being paid to employees of his/her hiring decade.

- To complete the query, we must replace the boxes labeled *subselect* in the preceding visuals by the SQL statements for the table expressions. The first box must be replaced by the SELECT statement for table expression *E*, shown on the previous visual. The second box must be replaced by the SQL statement for table expression *M*. For each hiring decade, this SQL statement determined the minimum salary paid to people hired during the respective decade.
Common Table Expressions

WITH
  E AS
  (SELECT EMPNO, LASTNAME, SALARY,
      SUBSTR(CHAR(HIREDATE,ISO),1,3) CONCAT '0 - 9'
      AS HIREDECADE
      FROM EMPLOYEE)
M (HIREDECATE, MINIMUM_SALARY) AS
  (SELECT HIREDECATE, MIN(SALARY)
      FROM E
      GROUP BY HIREDECATE)

SELECT E.EMPNO, E.LASTNAME, E.HIREDECADE,
      E.SALARY, M.MINIMUM_SALARY
FROM E INNER JOIN M
      ON E.HIREDECADE = M.HIREDECADE

Notes:

- The table expressions we have seen so far created temporary result tables within the FROM clause of an outer query. These table expressions could not be referenced elsewhere in the query, although their result columns could.

- Common table expressions can be referenced elsewhere in the query, even by other common table expressions within the same query. They are introduced by the keyword WITH and occur at the beginning of the query, not within the FROM of the outer query. They are separated from each other by commas. Every reference to a specific common table expression within the same query uses the same result set.

- The example on the visual reformulates the query that we used in the previous example, that is, the example for nested table expressions. It uses common table expressions which are named $E$ and $M$ as the nested table expressions were before.

  The first common table expression is the SQL statement for table expression $E$ which determines employee number, last name, salary, and hiring decade for all employees of the EMPLOYEE table. The common table expression is again called $E$. The columns of the associated result table are those named in the SELECT statement.
Although the second table expression looks different, it provides the same result as the SQL statement for table expression $M$ of the previous query: for the various decades, it determines the minimum salary being paid to the employees hired during the appropriate decade. The basic differences are:

- The SELECT statement now uses common table expression $E$, defined in front of common table expression $M$, instead of table EMPLOYEE.

- The columns of the common table expression are named by specifying their names in parentheses following the name of the common table expression. This is the same technique as naming the columns of a view. No AS clause is needed for the calculated column in the SELECT statement.

As a main select, the former outer query follows the common table expressions. Since it now can refer to the common table expressions, it becomes very easy.
Recursive SQL

WITH
  RPL (PART, SUBPART, QUANTITY) AS
  (  
    
    **Initialization Select**
    
    SELECT ROOT.PART, ROOT.SUBPART, ROOT.QUANTITY
    FROM PARTLIST ROOT
    WHERE ROOT.PART = '01'
    
    **Iterative Select**
    
    UNION ALL
    
    SELECT CHILD.PART, CHILD.SUBPART, CHILD.QUANTITY
    FROM RPL PARENT, PARTLIST CHILD
    WHERE PARENT.SUBPART = CHILD.PART
  )

**Main Select**

SELECT PART, SUBPART, SUM(QUANTITY) AS QUANTITY
FROM RPL
GROUP BY PART, SUBPART

Notes:

- Recursive SQL is used to work on tables that contain component breakdowns where each component is broken down into subcomponents and each subcomponent is broken down again into sub-subcomponents, etc. Problems involving these kinds of tables are often called "Bill of Materials" problems. A table that represented the parts in a computer would be an example of a Bill of Materials: the major components, the monitor, system unit, and printer, all contain subassemblies like the hard drive, the mother board, and the print head, each of which is composed of other subassemblies, etc., etc.

- Recursive SQL involves defining a common table expression that references itself. The common table expression consists of two distinct components, an initialization select and an iterative select. The initialization select is the first SELECT in the table expression and the iterative select is the second SELECT in the table expression. The iterative select is combined with the initialization select by means of UNION ALL.

The recursive common table expression in the example is named **RPL**. It is defined within the parentheses.
The common table expression in a recursive SQL statement is followed by a **main select**. The main select identifies the columns which are obtained from the result set of the common table expression.

- The example on the visual builds a final result set that identifies all the parts and subparts needed to build Part 01 (WHERE clause of initialization select) in a parts table called PARTLIST. We will see the PARTLIST table when stepping through the various "phases" by means of the subsequent visuals.
Recursive SQL - Initialization Select

```
SELECT ROOT.PART, ROOT.SUBPART, ROOT.QUANTITY
FROM PARTLIST ROOT
WHERE ROOT.PART = '01'
```

<table>
<thead>
<tr>
<th>PART</th>
<th>SUBPART</th>
<th>QUANTITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
<td>5</td>
</tr>
<tr>
<td>00</td>
<td>05</td>
<td>3</td>
</tr>
<tr>
<td>01</td>
<td>02</td>
<td>2</td>
</tr>
<tr>
<td>01</td>
<td>03</td>
<td>3</td>
</tr>
<tr>
<td>01</td>
<td>04</td>
<td>4</td>
</tr>
<tr>
<td>01</td>
<td>06</td>
<td>3</td>
</tr>
<tr>
<td>02</td>
<td>05</td>
<td>7</td>
</tr>
<tr>
<td>02</td>
<td>06</td>
<td>6</td>
</tr>
<tr>
<td>03</td>
<td>07</td>
<td>6</td>
</tr>
<tr>
<td>04</td>
<td>08</td>
<td>10</td>
</tr>
<tr>
<td>04</td>
<td>09</td>
<td>11</td>
</tr>
<tr>
<td>05</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>05</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>06</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>06</td>
<td>13</td>
<td>10</td>
</tr>
<tr>
<td>07</td>
<td>12</td>
<td>8</td>
</tr>
<tr>
<td>07</td>
<td>14</td>
<td>8</td>
</tr>
</tbody>
</table>

Notes:

- The initialization select is executed only once. In the example, it reads the PARTLIST table.
- The WHERE clause of the initialization select controls the starting point of the recursion. In the example, the starting point is all rows with a part number of '01'.
- The right-hand side of the visual displays the four rows placed in the temporary table RPL as the consequence of the initialization select. Parts 02, 03, 04, and 06 are the assemblies that directly make up Part 01. The first column (PART) of the interim result identifies the major part. The second column (SUBPART) identifies the subparts that make up the major part. The third column (QUANTITY) identifies the quantity of the subpart needed to construct one complete major part. For example, it takes three units of Part 06 to construct Part 01.
### Recursive SQL - First Iteration

**Figure 7-10. Recursive SQL - First Iteration**

```
SELECT CHILD.PART, CHILD.SUBPART, CHILD.QUANTITY
FROM RPL PARENT, PARTLIST CHILD
WHERE PARENT.SUBPART = CHILD.PART
```

<table>
<thead>
<tr>
<th>PART</th>
<th>SUBPART</th>
<th>QUANTITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
<td>5</td>
</tr>
<tr>
<td>00</td>
<td>05</td>
<td>3</td>
</tr>
<tr>
<td>01</td>
<td>02</td>
<td>2</td>
</tr>
<tr>
<td>01</td>
<td>03</td>
<td>3</td>
</tr>
<tr>
<td>01</td>
<td>04</td>
<td>4</td>
</tr>
<tr>
<td>01</td>
<td>06</td>
<td>3</td>
</tr>
<tr>
<td>02</td>
<td>05</td>
<td>7</td>
</tr>
<tr>
<td>03</td>
<td>07</td>
<td>6</td>
</tr>
<tr>
<td>03</td>
<td>08</td>
<td>10</td>
</tr>
<tr>
<td>04</td>
<td>09</td>
<td>11</td>
</tr>
<tr>
<td>05</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>05</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>06</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>06</td>
<td>13</td>
<td>10</td>
</tr>
<tr>
<td>07</td>
<td>12</td>
<td>8</td>
</tr>
<tr>
<td>07</td>
<td>14</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PART</th>
<th>SUBPART</th>
<th>QUANTITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>02</td>
<td>2</td>
</tr>
<tr>
<td>01</td>
<td>03</td>
<td>3</td>
</tr>
<tr>
<td>01</td>
<td>04</td>
<td>4</td>
</tr>
<tr>
<td>01</td>
<td>06</td>
<td>3</td>
</tr>
<tr>
<td>02</td>
<td>05</td>
<td>7</td>
</tr>
<tr>
<td>02</td>
<td>06</td>
<td>6</td>
</tr>
<tr>
<td>03</td>
<td>07</td>
<td>6</td>
</tr>
<tr>
<td>04</td>
<td>08</td>
<td>10</td>
</tr>
<tr>
<td>04</td>
<td>09</td>
<td>11</td>
</tr>
<tr>
<td>05</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>06</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>06</td>
<td>13</td>
<td>10</td>
</tr>
</tbody>
</table>

**Notes:**

- Unless it is limited by control variables, the iterative select is executed until all subparts of all parts have been broken down into their subparts, no matter how many repetitions are required. In our example, there are no control variables so the iteration will continue until all parts are completely resolved.

**Note:** It is very easy to write a recursive SQL statement incorrectly and initiate an infinite loop. Control variables are very useful for limiting the number of iterations and are discussed later in this unit.

- The iterative select in the example was the part of the recursive SQL statement between the UNION ALL and the parenthesis that closed the common table expression named RPL. Only the iterative select is repeated on this visual.
During the first iteration, each row from the initialization select is joined to all rows in the PARTLIST table that meet the join criteria. The result rows are added to the temporary table RPL. The rows that are added to RPL indicate that Parts 05 through 09 and 12 through 13 make up the parts returned by the initialization select:

- Part 02 consists of Parts 05 and 06;
- Part 03 consists of Part 07;
- Part 04 consists of Parts 08 and 09;
- and Part 06 consists of Parts 12 and 13.
Recursive SQL - Second Iteration

**Select Statement:**
```
SELECT CHILD.PART, CHILD.SUBPART, CHILD.QUANTITY
FROM RPL PARENT, PARTLIST CHILD
WHERE PARENT.SUBPART = CHILD.PART
```

**Part List Table**
```
<table>
<thead>
<tr>
<th>PART</th>
<th>SUBPART</th>
<th>QUANTITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
<td>5</td>
</tr>
<tr>
<td>00</td>
<td>05</td>
<td>3</td>
</tr>
<tr>
<td>01</td>
<td>02</td>
<td>2</td>
</tr>
<tr>
<td>01</td>
<td>03</td>
<td>3</td>
</tr>
<tr>
<td>01</td>
<td>04</td>
<td>4</td>
</tr>
<tr>
<td>01</td>
<td>06</td>
<td>3</td>
</tr>
<tr>
<td>02</td>
<td>05</td>
<td>7</td>
</tr>
<tr>
<td>02</td>
<td>06</td>
<td>6</td>
</tr>
<tr>
<td>03</td>
<td>07</td>
<td>6</td>
</tr>
<tr>
<td>04</td>
<td>08</td>
<td>10</td>
</tr>
<tr>
<td>04</td>
<td>09</td>
<td>10</td>
</tr>
<tr>
<td>05</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>05</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>06</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>06</td>
<td>13</td>
<td>10</td>
</tr>
<tr>
<td>07</td>
<td>12</td>
<td>8</td>
</tr>
<tr>
<td>07</td>
<td>14</td>
<td>8</td>
</tr>
</tbody>
</table>
```

**RPL Table**
```
<table>
<thead>
<tr>
<th>PART</th>
<th>SUBPART</th>
<th>QUANTITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>02</td>
<td>2</td>
</tr>
<tr>
<td>01</td>
<td>03</td>
<td>3</td>
</tr>
<tr>
<td>01</td>
<td>04</td>
<td>4</td>
</tr>
<tr>
<td>02</td>
<td>05</td>
<td>7</td>
</tr>
<tr>
<td>02</td>
<td>06</td>
<td>6</td>
</tr>
<tr>
<td>04</td>
<td>08</td>
<td>10</td>
</tr>
<tr>
<td>04</td>
<td>09</td>
<td>11</td>
</tr>
<tr>
<td>06</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>06</td>
<td>13</td>
<td>10</td>
</tr>
<tr>
<td>07</td>
<td>12</td>
<td>8</td>
</tr>
<tr>
<td>07</td>
<td>14</td>
<td>8</td>
</tr>
</tbody>
</table>
```

**Notes:**

- The second iteration joins the rows added by the first iteration to the PARTLIST table. The result rows of the second iteration are again added to RPL. The second iteration indicates that Part 05 consists of Parts 10 and 11, Part 06 consists of Parts 12 and 13, and that part 07 consists of parts 12 and 14.

Since there are no correspondences for the subparts of Parts 04 and 06, added by the first iteration, in the PARTLIST table, parts are not added for them to the RPL temporary table.

**Note:** RPL now contains two occurrences each of the rows that define the subparts of Part 06, namely, Parts 12 and 13. The first occurrence of these rows was contributed by the first iteration and the second occurrence of these rows came from the second iteration. The UNION ALL preceding the iterative select prevents the duplicate removal.

- The recursion will not yield additional rows after the second iteration because there are no further subparts for the parts added by the second iteration. However, if the PARTLIST table contained additional levels of subparts, the recursion would continue since the current example does not limit the depth of the recursion.
Recursive SQL - Main Select

```
SELECT PART, SUBPART, SUM(QUANTITY) AS QUANTITY
FROM RPL
GROUP BY PART, SUBPART
```

<table>
<thead>
<tr>
<th>PART</th>
<th>SUBPART</th>
<th>QUANTITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>02</td>
<td>2</td>
</tr>
<tr>
<td>01</td>
<td>03</td>
<td>3</td>
</tr>
<tr>
<td>01</td>
<td>04</td>
<td>4</td>
</tr>
<tr>
<td>01</td>
<td>06</td>
<td>3</td>
</tr>
<tr>
<td>02</td>
<td>05</td>
<td>7</td>
</tr>
<tr>
<td>02</td>
<td>06</td>
<td>6</td>
</tr>
<tr>
<td>03</td>
<td>07</td>
<td>6</td>
</tr>
<tr>
<td>04</td>
<td>08</td>
<td>10</td>
</tr>
<tr>
<td>04</td>
<td>09</td>
<td>11</td>
</tr>
<tr>
<td>06</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>06</td>
<td>13</td>
<td>10</td>
</tr>
<tr>
<td>05</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>05</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>06</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>06</td>
<td>13</td>
<td>10</td>
</tr>
<tr>
<td>07</td>
<td>12</td>
<td>8</td>
</tr>
<tr>
<td>07</td>
<td>14</td>
<td>8</td>
</tr>
</tbody>
</table>

**Notes:**

- After the recursive common table expression has been evaluated completely, the main select is evaluated. The main select references the result of RPL, the common table expression.

- The main select summarizes the total quantity of all parts needed to build Part 01. The grouping and the SUM() function ensure that the quantities of the respective subparts of Part 06 are added together. In other words, the two rows for Part 06, Subpart 12, will be combined to make a single row. So will the two rows for Part 06, Subpart 13. A user who wishes to verify that the warehouse contains enough of each of the components needed to make Part 01 can execute this query and, then, check existing stocks against the result of the query.
Controlling Depth of Recursion (Part 1)

Notes:

- Recursion normally continues until all parts have been resolved into their components. However, the depth of the recursion can be controlled by simulating the addition of a control column to the table.

- This control column is initialized arbitrarily to 0 in the initialization select and increased by one on every execution of the iterative select. A condition in the WHERE clause of the iterative select is used to ensure that the iteration only continues for a fixed number of levels.

- In the example, the control column is named LEVEL. It is set to an integer value of 0 in the initialization select. The WHERE clause of the initialization select determines the value in the PART column with which the table expression begins. This time, we are interested in the breakdown of Part 00, but we could have started with any part number we were interested in. The initial value of LEVEL would still be 0, regardless of the starting part number.
The iterative select increments the LEVEL value by adding 1 on each iteration. The condition

**PARENT.LEVEL < 2**

in the WHERE clause of the iterative select is used to limit the number of iterations: simply set the constant to the number of iterations which are desired.

The main select displays the result of the table expression. The LEVEL column in the final result makes the origin of each result row clear: rows that came from the initialization select have a level of 0, rows from the first iteration have a level of 1, rows from the second iteration have a level of 2, and so on. The ORDER BY puts the result in a convenient sequence.

**Note:** LEVEL is *not* a column of table PARTLIST. It does not have to be added to table PARTLIST via an ALTER TABLE statement. It is a “virtual” column created by the SQL statement.

- The actual result of the recursive SQL statement is illustrated on the next visual.
Controlling Depth of Recursion (Part 2)

<table>
<thead>
<tr>
<th>PART</th>
<th>SUBPART</th>
<th>QUANTITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
<td>5</td>
</tr>
<tr>
<td>00</td>
<td>05</td>
<td>3</td>
</tr>
<tr>
<td>01</td>
<td>02</td>
<td>2</td>
</tr>
<tr>
<td>01</td>
<td>03</td>
<td>3</td>
</tr>
<tr>
<td>01</td>
<td>04</td>
<td>4</td>
</tr>
<tr>
<td>01</td>
<td>06</td>
<td>3</td>
</tr>
<tr>
<td>02</td>
<td>05</td>
<td>7</td>
</tr>
<tr>
<td>02</td>
<td>06</td>
<td>6</td>
</tr>
<tr>
<td>03</td>
<td>07</td>
<td>6</td>
</tr>
<tr>
<td>04</td>
<td>08</td>
<td>10</td>
</tr>
<tr>
<td>04</td>
<td>09</td>
<td>11</td>
</tr>
<tr>
<td>05</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>05</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>06</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>06</td>
<td>13</td>
<td>10</td>
</tr>
<tr>
<td>07</td>
<td>12</td>
<td>8</td>
</tr>
<tr>
<td>07</td>
<td>14</td>
<td>8</td>
</tr>
</tbody>
</table>

**PARTLIST Table**

<table>
<thead>
<tr>
<th>LEVEL</th>
<th>PART</th>
<th>SUBPART</th>
<th>QUANTITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>01</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>05</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>02</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>03</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>04</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>06</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>05</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>05</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>05</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>06</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>03</td>
<td>07</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>04</td>
<td>08</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>04</td>
<td>09</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>06</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>06</td>
<td>13</td>
<td>10</td>
</tr>
</tbody>
</table>

**Final Result Table**

**Notes:**

- This visual displays the result of the breakdown for Part 00 if the recursion is limited to two iterations.
- Subparts 05 and 06 of Part 02 and Subpart 07 of Part 03 could be further decomposed if the number of iterations was not limited to two.
Recursive SQL - Recommendations

- **Desk-check** recursive SQL statement
- **Test** recursive SQL against small test tables
- **Use controls to limit recursion**

**Notes:**

- Recursive SQL is cyclical by definition. This means that it is easy to cause loops if the SQL is coded incorrectly or if the data itself is cyclical. For example, if the join in the iterative select in the earlier examples was coded as

  \[ \text{PARENT.SUBPART} = \text{CHILD.SUBPART} \]

  there would be an infinite recursion if there was even one row where the Part and Subpart values were the same. By the same token, a loop can occur if the data was illogical. For example, if the PARTLIST table had a row where the Part was 05 and the Subpart was 01, a loop would occur. To prevent this sort of problem, desk check all recursive SQL. Also, test it against small tables before implementing it in production.

- Any recursive SQL statement that does not use a control variable will receive an SQL warning (SQL0347W in UDB). Although this is not a serious problem, you can use techniques shown on the preceding visuals to avoid it.
Checkpoint

1. T/F. Table expressions have many similarities to views.

2. List three ways in which table expressions can be more convenient than views.

3. T/F. If your data is invalid and you code a recursive SQL statement, it is possible to start an infinite loop.

4. What key element in the syntax of an SQL statement makes it recursive?

Notes:
Write down your answers here:

1.

_____________________________________________________________

2.

_____________________________________________________________

3.

_____________________________________________________________

4.

_____________________________________________________________

You will find an example of the answers in the Appendix.
Summary

Now that you have completed this unit, you should be able to:

- Identify reasons for table expressions and recursive SQL
- Use nested and common table expressions
- Identify the difference between views and table expressions
- Code recursive SQL
- Control the depth of recursion when coding recursive SQL

Notes:
Unit 8. UDTs/UDFs and Performance

What This Unit Is About

This unit describes the concepts behind User Defined Types (UDTs) and User Defined Functions (UDFs). It also discusses some basic performance considerations.

What You Should Be Able to Do

After completing this unit, you should be able to:

• Describe the concepts behind User-Defined Types and User-Defined Functions.

• Predict when queries will use indexes to get better performance.

How You Will Check Your Progress

Accountability:

• Unit Checkpoint
• Paper Lab

Labs

- Index usage

References

DB2 UDB for OS/390 and z/OS V7
  SQL Reference (SC26-9944-02)

DB2 Administration Guide (DB2 UDB 7.2 for Windows)
DB2 SQL Reference (DB2 UDB 7.2 for Windows)
Objectives

After completing this unit, you should be able to:

- Describe the concepts behind User-Defined Distinct Types and User-Defined Distinct Functions
- Predict when queries will use indexes to get better performance

Notes:
- UDTs and UDFs are available in DB2 UDB for UNIX, Windows and OS/2 since Version 2 and on DB2 for OS/390 from Version 6 and on DB2 UDB for AS/400.
8.1 User-Defined Distinct Types
User-Defined Distinct Types - The Need

- Need to establish context for values
- DB2 enforced typing

Notes:

- Standard data types, like INTEGER and CHAR, are not always adequate for every business need. A company sometimes needs data types that specifically reflect the use of data. DB2 allows users to create their own data types. These are called User-Defined Distinct Types, or UDTs for short.

- UDTs allow DB2 itself, rather than applications, to ensure that only valid comparisons are made. This is especially useful for cases where users are writing ad-hoc queries and may not realize that one column is not logically comparable to another.

- The diagram represents a situation where the statement "SPEED LIMIT 100" is ambiguous until the unit of measure is applied. 100 miles per hour is NOT the same as 100 kilometers per hour. The American motorist who has just crossed the border into Canada has seen a sign that says “SPEED LIMIT 100” and assumes that this means 100 miles per hour. The Canadian police officer pursuing him will ensure that his mistaken impression is soon corrected.
User-Defined Distinct Types - Definition

CREATE DISTINCT TYPE KPH AS INTEGER
WITH COMPARISONS

CREATE DISTINCT TYPE MPH AS INTEGER
WITH COMPARISONS

CREATE TABLE SPEED LIMITS
( ROUTE_NUM SMALLINT NOT NULL,
  CANADA_SL KPH NOT NULL,
  US_SL MPH NOT NULL )

SELECT ROUTE_NUM
FROM SPEED LIMITS
WHERE CANADA_SL > KPH(80)

SELECT ROUTE_NUM
FROM SPEED LIMITS
WHERE CANADA_SL > US_SL

Notes:

- The CREATE DISTINCT TYPE statement allows users to create data types that can be enforced by the database manager.

- The name of the new UDT can be a qualified or an unqualified name. If the qualifier is different from the authorization ID of the statement, the authorization ID must have DBADM authority on the database.

- UDTs are always based on existing built-in data types like CHAR and SMALLINT. UDTs cannot be used as source types for newer UDTs.

- The WITH COMPARISONS clause tells DB2 to generate =, <>, <, <=, >, and >= operators for comparing different instances of the UDT. This clause is required if the source data type supports these comparisons and is prohibited if the source data type does not support comparisons. The source data types which do not support comparisons are BLOB, CLOB, DBCLOB, LONG VARCHAR, and LONG VARGRAPHIC.

- DB2 always generates cast functions to:
- CAST from the UDT to the source type, using the standard name of the source type.
- CAST from the source type to the UDT.

These functions are important for the manipulation of the UDTs in queries.

- The first example in the upper box creates a UDT called KPH, which is based on the INTEGER built-in data type.
- The second example in the upper box creates a UDT called MPH, which is also based on the INTEGER data type.
- The third example in the upper box creates a table named SPEED_LIMITS that contains three columns. The first column, ROUTE_NUM, contains a highway number like 401 or I95. The second column, CANADA_SL, is defined with the new data type KPH. The third column, US_SL, is defined with the new data type MPH. Presumably, each highway in the table, whether Canadian or American, contains the speed limit expressed in both kilometers per hour and miles per hour.
- The first example in the lower box asks for all the highways in the SPEED_LIMITS table that have a CANADA_SL value that is greater than 80. However, the query cannot simply say: 'WHERE CANADA_SL > 80'. The number 80 is an integer, but CANADA_SL is not an integer, it is a KPH. Therefore, the 80 has to be cast to a KPH value via the 'KPH(80)' expression. This CAST function was created automatically by DB2 when the KPH UDT was created.
- The second example in the lower box attempts to get all the rows of the SPEED_LIMITS table where the CANADA_SL value is greater than the US_SL value. KPH and MPH values cannot be compared directly so the statement fails. The WHERE clause in this query would need to be changed to

  WHERE INTEGER(CANADA_SL) > INTEGER(US_SL)

In other words, both columns would need to be explicitly cast back into their source types in order to compare them.

Note: The CAST functions cannot be used to convert a speed limit that is in KPH to its equivalent speed in miles per hour. The CAST functions will not, for example, convert 80 kilometers per hour to 50 miles per hour.
User-Defined Distinct Functions - The Need

**Notes:**

- Although many functions are already built into DB2 and more may emerge over time, many users want more functions. DB2 supports User-Defined Functions, UDFs for short, as a way for users to develop their own functions and have DB2 manage them.

- A UDF can perform many tasks. For example, UDFs can be written to:
  - Convert between different measuring systems, like the metric and US systems
  - Do complex calculations like multivariate analyses
  - Act as methods for UDTs by providing behavior and encapsulation for the distinct types

- UDFs are all registered in DB2. This ensures that they always give consistent answers.

- UDFs can return a single value or a whole table. UDFs can be based on other UDFs that already exist.

- UDFs can for example be written in the following languages: C, Java, and OLE.

- UDFs are normally developed by programmers. The IBM programming classes describe the development process fully and let students develop a UDF.
User-Defined Distinct Functions - Sourced

```
CREATE FUNCTION "+" (MPH,MPH)
RETURNS MPH
SOURCE "+" (INTEGER,INTEGER)

CREATE FUNCTION AVG_US_SPEED (MPH)
RETURNS MPH
SOURCE AVG(INTEGER)
```

Existing Built-in Functions
(or UDFs)
SUM + - / * AVG MIN, etc.

Notes:

- The CREATE FUNCTION statement is used to define UDFs.
- Strong typing defines that data types can only be used as defined by the installation. Therefore, a UDT cannot be used in conjunction with functions provided by the database manager unless usage is defined. To assist the installation in defining common functions, DB2 allows functions to be sourced on previously defined functions. These previously defined functions could be provided by the database manager or user-written.
- It might be advantageous NOT to define a function sourced on every function provided by the database manager for certain UDFs. For example, assume the employee number is defined as a distinct type based on the INTEGER type. The installation would probably not define the functions SUM, AVG, +, -, *, and / on this data type since such manipulations would be invalid.
A function name can also be overloaded, so that the same name can be used several times. For example, consider the function `AVG_US_SPEED` defined in the visual. This function could have been defined as `AVG`, even though such a function name already exists. The database manager will invoke the function that matches the input parameters passed in an overloading situation.
Notes:

- A UDF can accept one or more input parameters, but a single value must be returned. Furthermore, the UDF code cannot contain SQL.

- The UDF is coded in a programming language such as C. This allows the installation to take advantage of functions available in the language. The installation can take advantage of library management of such functions as well. For example, the functions defined in the visual are conversion functions. They reside in a library called convert.

- A UDF can be invoked anywhere in an SQL statement where an expression is currently permitted.

- The optimizer considers UDF during access path selection.

- For an example of usage of the UDF in the visual:

  ```sql
  SELECT ROUTE_NUM FROM SPEED_LIMITS
  WHERE CANADA_SL > MPH_TO_KPH(US_SL)
  ```

- Normally, programmers will develop UDFs. The IBM programming classes describe the development process fully and let students develop a UDF.
8.2 Performance Considerations
Index Concepts

Notes:

- Indexes are used by DB2 in much the same way as indexes in books are used by people. The index contains two major parts - a desired key value and a pointer that identifies where the corresponding row can be found in the table. This identifier is known as the RID, which is an acronym for Row IDentifier.

- The index is structured in strict ascending or descending sequence on the columns in the index. The desired value can be located quickly in the index because of this ascending or descending sequence. This again is similar to searching the index in the back of a book. For example, assume you need to look in the index of a book for any information concerning "configuration". Because the index is ordered, you need not scan through the entire index. You can quickly find any entries that match your search criteria by finding the “C” part of the index and applying your knowledge of the alphabet.

- An index is not essential for finding data: it is essential for finding the data quickly though. It is still possible to find data in a table that has no indexes, but, in this case, every row of the table must be read. Indexes simply reduce the time needed to get the data.
Index Structure

Figure 8-8. Index Structure

Notes:

- An index is structured as a binary tree. This structure enables the database manager to find values in the index rapidly.

- The root node is at the top of this structure. In the graphic, the root node contains an entry for each non-leaf node on the next level down. The entry in the root node consists of the high value contained on the next level and a pointer to this node. Entries are represented by integers in the graphic and pointers are represented by arrows. For example, the '45' entry in the root node in the graphic points down to the second non-leaf node on the next level. There is always just one root node.

- The non-leaf nodes are similar in structure to the root node, except that the range of values addressed is narrower. A non-leaf node contains an entry for each of the nodes on the next level down. (In this case, there is only one non-leaf level, but it is possible to have several non-leaf levels.) The entries in a non-leaf node consist of the high values contained on a node on the next level down and a pointer to this node. The '13' in the first non-leaf node points to the third node on the next level.
- The leaf nodes contain the value/RID pairs themselves. The leaf nodes collectively address the entire table. Each leaf node contains actual key values and the RIDs which identify the location in the data pages of the rows that have those key values.
- The data pages contain the full rows of the table. The RIDs on the leaf nodes point to the location of the row on the data page.
Notes:

- A predicate is a single search condition.
- The first and second predicates in the graphic, 'A = B' and 'C = D', are often referred to as “equal predicates”. The third search criteria, 'J > K', is a “range predicate”.
- The part of DB2 that executes predicates is called the Optimizer.
Predicative Processing

Notes:

- There are three main parts in the Optimizer: the Index Manager, Data Management Services, and Relational Data Services. Each is responsible for different types of predicates.

- Predicates that can be evaluated through the use of an index are evaluated by the Index Manager. These predicates include range-delimiting and index-sargable predicates.

  - A range-delimiting predicate allows the index manager to narrow the search of index pages. If DEPTNO was the leftmost column of an index and the query contains the predicate 'DEPTNO = 10', this predicate is considered range-delimiting. DB2 can use this index to limit the number of index pages it examines so that only pages with a DEPTNO value of 10 are examined. This will most likely save considerable I/O with respect to reading the entire index or table.

  - An index-sargable predicate does not limit the range of pages examined, but it can help find the data faster. If SALARY was a column of an index, but was not one of the leading columns and the values in the leading columns was unknown, the...
The predicate 'SALARY > 50000' would be considered index-sargable. The predicate could not be used to reduce the number of index pages read, but it could still be used to reduce the number of data rows which the query would have to read.

- Predicates that cannot be evaluated by the index manager, but do not require the more complex algorithms of relational data services, are called **data-sargable** predicates. An example of a data-sargable predicate is DEPTNO = 'D10' when DEPTNO is not in an index. These predicates do not reduce the number of index rows read nor the number of data rows read, but they do reduce the number of data rows returned to the result set.

- Predicates that require more complex evaluation require relational data services and are called **residual** (remaining) predicates. In DB2 for OS/390, they are also referred to as **Stage 2** or **non-sargable** predicates. Correlated subqueries, quantified subqueries, and predicates that use LONG VARCHAR columns are all examples of residual predicates. The term residual also infers that these predicates are evaluated after the other predicates have been applied. The presence of a residual predicate is not in itself cause for concern. However, limiting the number of residual predicates can enhance performance.

- Although there are always exceptional cases, the best performance is obtained from range-delimiting predicates and the worst from residual predicates.

- The user or programmer does not need to analyze each predicate manually in order to predict performance. The EXPLAIN facility provided with DB2 will generate an explanation of the access paths that DB2 will use to resolve each part of the SQL statement.
Figure 8-11. Index Utilization

<table>
<thead>
<tr>
<th>LASTNAME</th>
<th>FIRSTNAME</th>
<th>AGE</th>
<th>RID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abell</td>
<td>Jim</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Campbell</td>
<td>Debbie</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cheng</td>
<td>David</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>Free</td>
<td>Ann</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>Free</td>
<td>Bernie</td>
<td>66</td>
<td></td>
</tr>
<tr>
<td>Free</td>
<td>David</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>Free</td>
<td>Debbie</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>Free</td>
<td>Dennis</td>
<td>37</td>
<td></td>
</tr>
<tr>
<td>Free</td>
<td>Dianne</td>
<td>41</td>
<td></td>
</tr>
<tr>
<td>Free</td>
<td>Donald</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>Free</td>
<td>Donna</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td>Free</td>
<td>Doreen</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>Free</td>
<td>Doug</td>
<td>34</td>
<td></td>
</tr>
<tr>
<td>Klonne</td>
<td>Dennis</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. SELECT *
   WHERE LASTNAME = 'Free'

2. SELECT LASTNAME
   WHERE FIRSTNAME = 'David'

3. SELECT *
   WHERE LASTNAME = 'Free'
   AND FIRSTNAME > 'Diane'

4. SELECT *
   WHERE LASTNAME = 'Free'
   AND AGE > 36

5. SELECT ...
   WHERE ...
   GROUP BY LASTNAME
   \-or-
   ORDER BY LASTNAME

Notes:

- The graphic illustrates an index and various predicates that DB2 might encounter. The index is on the combination of LASTNAME, FIRSTNAME, and AGE in an imaginary table. Assume that there are thousands or millions of rows in the imaginary table.

- Given the SQL statements on the right of the graphic, this is the “reasoning” that DB2 would go through to decide whether it was going to use the index.

  1. This statement requests all the columns for all rows for people named 'Free'. Since the predicate applies to the first (leftmost) column in the index, the predicate is range-delimiting. The index only needs to be examined for those entries that have 'Free' in the first part of the index. The RIDs for those entries could then be used to access the data rows of the people named 'Free'.

  2. This statement requests only the last name of people that have a first name of 'David'. The predicate is not range-delimiting, because the binary structure of the index cannot be exploited to limit the search. (Someone with the first name of 'David' could have any last name, and, therefore, appear in any leaf node.) However, the predicate is index-sargable: the column to be evaluated, the first name, does appear...
in the index. DB2 will have to read the entire index, but it will be able to find the people named 'David' without having to go outside of the index. Also, since the last names of the Davids are also in the index, DB2 only needs to find the Davids in the FIRSTNAME column and then look to the left to get the corresponding LASTNAME. The table will not have to be accessed at all. This is called **index-only access** and is very desirable since it performs very well.

Index-sargable predicates can be more efficient than relational scans in cases where:

- The number of data pages is large compared to the number of leaf-node pages
- Other predicates have been evaluated as range-delimiting predicates in the same composite index
- The index itself provides all the necessary information to resolve the SQL statement. This is the case in Example 2.

3. Both of the predicates in this example are range-delimiting predicates. Even though the second predicate is not an equal predicate, it can assist in narrowing the number of leaf pages that have to be scanned in the index. It is not necessary for the index manager to read all entries that have a LASTNAME = 'Free', because the second predicate (FIRSTNAME > 'Diane') provides information that can be exploited to position the search. Certainly, a predicate such as FIRSTNAME = 'Diane' would limit the search even further, but the given predicate does provide useful information.

**Note:** When determining the number of range-delimiting predicates for a multiple-column index, the order of columns in the index is significant. Also, any predicates beyond the first non equal predicate cannot be exploited as range-delimiting, although they would be index-sargable. For example, assume a third predicate (AGE = 37) was added to this example. The Age predicate could not be exploited to reduce the amount of reading that had to be done in the index, since the prior column in the index (FIRSTNAME) is not in an equal predicate.

4. This statement has a range-delimiting predicate and an index-sargable predicate. LASTNAME = 'Free' is a range-delimiting predicate for the reasons discussed in earlier examples. AGE > 36 cannot be used to limit the search since there is an intervening column in the index for which DB2 has not been given a value. However, the AGE could be used for comparison. Therefore, the second predicate is index-sargable. DB2 would find all index entries that satisfy both predicates and then read only the data rows for those index entries.

5. The last statement is provided to highlight a function of the database manager. Data may be accessed via an index to avoid a sort if it is determined to be less costly than other access strategies followed by a sort. The GROUP BY clause and ORDER BY clause are just two SQL constructs that require ordered data; others are documented in the *Administration Guide*. In these cases, DB2 calculates the cost of accessing all of the qualifying rows and then sorting them and compares this with
the cost of reading the qualifying rows in sequence via the index. DB2 will use whichever approach is the cheapest.

- The process of determining the least expensive access path to satisfy an SQL statement is called optimization. This occurs during BIND.

**Note:** When multiple predicates were provided in the examples above, the logical operator between the predicates was the 'AND' operator. This is necessary for the predicate LASTNAME = 'Free' to be considered range-delimiting. If the logical operator was 'OR', isolating individuals with LASTNAME = 'Free' would not be considered range-delimiting.
The Optimizer

- The Optimizer chooses the access path for all SQL statements
- The Optimizer calculates the cost of each possible access path for a given query and uses the path that is the cheapest
  - As a result, the Optimizer may choose not to use a given index as an access path if its calculations show that some other approach is cheaper
- Each code version and operating system for DB2 has its own distinct Optimizer
  - The Optimizers have been evolving steadily since DB2 was first released
- Each Optimizer is capable of rewriting some queries to improve performance
  - The rewrite techniques vary for each Optimizer
- Poorly written SQL can dramatically hurt the performance of a given query
  - The Optimizer cannot always rewrite these queries
- Changes that help a query run better under one Optimizer may not help the same query run better under another Optimizer

Notes:

- The Optimizer is used to calculate access paths for SQL regardless of its source: SPUFI, QMF, the Command Center, the command line, or an application.
- The Optimizer will not use an index if the amount of data is very small: in this case, the Optimizer may calculate that it is cheaper to read all the rows with a table scan and discard rows that do not satisfy the predicates in the query.
- UNIX, Windows and OS/2 versions of DB2 UDB all share the same code base so they also share the same Optimizer. Mainframe versions of the Optimizer have also changed with virtually every release.
- The Optimizer can rewrite queries in a variety of different ways. For example, the Optimizer can:
  - Rewrite subqueries as joins
  - Merge views
  - Eliminate redundant DISTINCTs
  - Convert correlated subqueries to non correlated subqueries
- Convert multiple OR predicates into an IN predicate
- Convert a 'NOT <' predicate to a '>=' predicate

- Some queries provide poor performance and cannot be rewritten. For example, a '<>'-predicate cannot be turned into a positive predicate.

**Note:** The Optimizer is referred to as the 'SQL Compiler' in some of the DB2 manuals.
General Guidelines - Correlated Subqueries

```
SELECT EMPNO, LASTNAME, WORKDEPT, SALARY
FROM EMPLOYEE E
WHERE
  SALARY >
  (SELECT AVG(SALARY)
   FROM EMPLOYEE
   WHERE WORKDEPT = E.WORKDEPT)
```

```
SELECT EMPNO, LASTNAME, E.WORKDEPT, SALARY
FROM EMPLOYEE E INNER JOIN
  (SELECT WORKDEPT,
   AVG(SALARY) AS AVG_SALARY
   FROM EMPLOYEE
   GROUP BY WORKDEPT
  ) AS D
ON E.WORKDEPT = D.WORKDEPT
WHERE SALARY > AVG_SALARY
```

Notes:

- Correlated subqueries tend to perform quite poorly. The first example illustrates why.
- When it encounters an SQL statement that uses a non correlated subquery, DB2 executes the subquery first and passes the result set to the outer query to control the rows which are returned by the outer query. The subquery is performed only once so it tends to be fairly inexpensive, even if it has to read many rows.
- By contrast, a correlated subquery involves repeated executions of the subquery. In the case of the first example, a single row of the EMPLOYEE table is read by the outer query and EMPNO, LASTNAME, WORKDEPT, and SALARY are determined from this row. Then, the WORKDEPT value is passed to the subquery, and the subquery is used to calculate the average salary for all people in that department.

A single value is passed back to the outer query, which then compares the salary of the individual it read earlier with the average salary determined by the subquery. If the individual salary is higher than the average, the row found by the subquery is written to the result set; otherwise it is omitted. This process is followed for each row that satisfies
the outer query. In a worst-case situation, if the EMPLOYEE table contains a million rows, the subquery will be executed a million times.

- Because of the possibility of poor performance, many people avoid correlated subqueries whenever possible. However, this may be too drastic: the Optimizers on many platforms have been improved so their performance is considerably better than the worst-case scenario discussed above. These Optimizers use a scratchpad area to record the subquery results for given values of the correlation value.

For example, the average salary for each department is written to the scratchpad when the average for the department is calculated the first time. When the outer query finds a row with the same correlation value as one that has been saved on the scratchpad, DB2 simply goes to the scratchpad to find the value. That eliminates the need to recalculate the average salary for a given department after it has been done once. This technique improves the performance of the correlated subquery considerably. Given the same million-row EMPLOYEE table and only 100 departments for which the employees work, an Optimizer that uses a scratchpad will only have to perform the subquery 100 times, which is much better than the million times that would have been required in the worst-case scenario.

- The second example produces the same result as the first example, but is not correlated and performs considerably better, even on a platform where scratchpads are used to handle correlated subqueries. In this case, it is cheaper to do the same work as a non correlated subquery.
Notes:

- All of the following may require sorts: ORDER BY, GROUP BY, DISTINCT, UNION, EXCEPT, INTERSECT, and some joins. In some cases, DB2 will read the data in the correct order via the index and avoid the sort, but, in all other cases, DB2 will have to sort. Sorts add to the processing done by the query: They increase CPU requirements and can require additional I/O activity in DB2's sort work area. Sorts are handled by the Relational Data Services component of DB2, the part that handles all the residual predicates. Therefore, performance can suffer if sorts are needed.

- Do not request sorts unless you need sorted data.

- When it makes sense, create indexes on the columns that are used in the ORDER BY, GROUP BY, and other sort-provoking statements. This will enable DB2 to consider using indexes to read the rows via the index and eliminate the sort.
General Guidelines - Use Views

Notes:

**USE VIEWS, BUT DO NOT ABUSE THEM!**

- The view in the example could be easily created by using the two base tables. If users needed both of the columns in the view, the view could save effort among end users. The DBA who created the view could ensure that the join was done correctly so that it avoided Cartesian Products and could even create extra indexes, if needed, to ensure the best possible join performance. In this case, the view could be considered a performance aid.

However, if the users really only needed one of the columns in the view, a view that was based on a join would cause unnecessary processing to join the two tables. In this case, the user should either have a view that was based on a single table to avoid the unnecessary join or, possibly, have access to the base table that contains the needed data.
General Guidelines - Expressions

Given:
- WORKDEPT is a column of table EMPLOYEE
- There is a nonunique index on column WORKDEPT

Example 1:
```
SELECT EMPNO, LASTNAME
FROM EMPLOYEE
WHERE SUBSTR (WORKDEPT, 1, 2) = 'A0'
```

Example 2:
```
SELECT EMPNO, LASTNAME
FROM EMPLOYEE
WHERE WORKDEPT LIKE 'A0%'
```

Avoid expressions in predicates whenever possible

Notes:
- Scalar functions are considered expressions. DB2 will never use an index to satisfy a predicate that contains an expression, unless the expression evaluates to a constant.
- The first example will never use an index because its predicate, "SUBSTR(WORKDEPT,1,2) = 'A0'”, uses the SUBSTR() function, which is considered an expression by DB2.
- The second example does not contain an expression. Therefore, DB2 will consider using an index to satisfy this query.
- Other SQL constructs including arithmetic, the concatenation operator, and casting, are also considered expressions. See the SQL Reference manual for your platform to get a complete list of expressions.
- People writing SQL should investigate alternatives to coding expressions in predicates.
General Guidelines - '<>' Predicates

Given:
- WORKDEPT is a column of table EMPLOYEE
- All employees belong to departments A00 or C01
- There is a nonunique index on column WORKDEPT

Example 1:

SELECT EMPNO, LASTNAME, SALARY
FROM EMPLOYEE
WHERE WORKDEPT <> 'A00'

Example 2:

SELECT EMPNO, LASTNAME, SALARY
FROM EMPLOYEE
WHERE WORKDEPT = 'C01'

Avoid '<>' predicates whenever possible

Notes:

- DB2 UDB for UNIX, Windows and OS/2 uses the index on WORKDEPT to build the result sets for both examples. However, the total cost of the first example is much higher than the total cost of the second example.
- DB2 for OS/390 does not use the index on WORKDEPT to build the result set for the first example, but it does use that index for the second example. The fact that the predicate in the first example uses a not-equal ('<>') operator is the factor that causes the index to be bypassed.
General Guidelines - Arithmetic

Given:
- SALARY is a column of table EMPLOYEE
- There is a nonunique index on column SALARY

Example 1:
```
SELECT EMPNO, LASTNAME
FROM EMPLOYEE
WHERE SALARY+2000 > 40000
```

Example 2:
```
SELECT EMPNO, LASTNAME
FROM EMPLOYEE
WHERE SALARY > 38000
```

Avoid arithmetic in predicates whenever possible

Notes:
- DB2 UDB for UNIX, Windows and OS/2 do not use the index on SALARY for the first example, but uses it for the second example. The total cost of the first example is almost three times as high as that for the second example.
- DB2 for OS/390 does not use the index on SALARY for the first example, but does use it for the second. The fact that the predicate in the first example does arithmetic is the factor that causes the index to be bypassed.
General Guidelines - Search Value Length

Given:
- WORKDEPT is a column of table EMPLOYEE
- WORKDEPT is defined as CHAR(3)
- There is a nonunique index on column WORKDEPT

Example 1:

```
SELECT EMPNO, LASTNAME
FROM EMPLOYEE
WHERE WORKDEPT = 'A00
```

Example 2:

```
SELECT EMPNO, LASTNAME
FROM EMPLOYEE
WHERE WORKDEPT = 'A00'
```

Avoid making search values longer than the searched column whenever possible

Notes:
- DB2 UDB for UNIX, Windows and OS/2 uses the index on WORKDEPT to build the result sets for both examples. The costs of the two examples are identical in this case.
- DB2 for OS/390 does not use the index on WORKDEPT to build the result set for the first example, but it does use it to build the result set for the second example. The fact that the search value is longer than the width of the WORKDEPT column in the first example is the factor that causes the index to be bypassed.
General Guidelines - Conversions

Given:
- EDLEVEL is a column of table EMPLOYEE
- EDLEVEL is defined as SMALLINT
- There is a nonunique index on EDLEVEL

Example 1:
```
SELECT EMPNO, LASTNAME
FROM EMPLOYEE
WHERE EDLEVEL > 15.00
```

Example 2:
```
SELECT EMPNO, LASTNAME
FROM EMPLOYEE
WHERE EDLEVEL > 15
```

Avoid using search values that have different data types from searched column whenever possible

Notes:
- DB2 UDB for UNIX, Windows and OS/2 uses the index on EDLEVEL to build the result sets for both examples. The costs of the two examples are very similar in this case.
- DB2 for OS/390 does not use the index on EDLEVEL to build the result set for the first example, but it does use it to build the result set for the second example. The fact that the search value is a decimal number while EDLEVEL is a small integer is the factor that causes the index to be bypassed in the first example.
General Guidelines - Statements

- Return only the columns that are actually needed
- Use local predicates
- Write joins if equivalent to subquery
- UNION ALL performs better than UNION
- Use Evaluate at Open subqueries when possible

Notes:

- Although it is very convenient to code 'SELECT *', production queries, especially those that access tables with large numbers of rows, should always return only the columns actually needed by the query. Each unneeded column returned by the query adds to the cost of the query. This cost is fairly small when the data is accessed locally, but can be very substantial when accessing the data from a remote database, due to the increased network traffic that occurs when unneeded data is returned.

- Local predicates are predicates that apply to only one table. Any predicate that is not a join predicate is a local predicate. Local predicates can allow the database manager to limit the number of rows that are used in multiple-step access paths. They can be extremely significant in determining which table DB2 accesses first in join processes. Use local predicates to help improve performance.

- In general, joins can outperform subqueries since there are many access strategies available to support the join. If you have the choice of writing a given query as a join or a subquery, choose the join.
**Note:** Most of the DB2 Optimizers will convert subqueries to joins for you whenever possible. DB2 does this “under the covers” (within DB2) so you will not see it happen, but it does occur in many cases.

- UNION ALL is more efficient than UNION. UNION ALL does not require sorting to eliminate duplicate results. Use UNION ALL whenever you can.

- Subqueries that are executed to completion when the outer query is executed are called *Evaluate at Open (EAO)* subqueries. Use EAO subqueries whenever possible. These subqueries are sargable with the exception of “= ANY”, “<> ALL”, and “<> ANY” quantified predicates. Subqueries that are correlated to the outer query cannot be evaluated until a particular row is evaluated in the outer query. Such subqueries are always residual and tend to perform poorly, as discussed earlier.
Checkpoint

1. T/F. User-Defined Types can be compared without restrictions.

2. How much data must DB2 read to satisfy a WHERE condition if no index is available?

3. What kinds of predicates can be evaluated very efficiently?

4. Why will a correlated subquery sometimes perform poorly?

Notes:

Write down your answers here:

1. ________________________________________________________________

2. ________________________________________________________________

3. ________________________________________________________________

4. ________________________________________________________________

You will find an example of the answers in the Appendix.
Summary

Now that you have completed this unit, you should be able to:

- Describe the concepts behind User-Defined Distinct Types and User-Defined Distinct Functions
- Predict when queries will use indexes to get better performance

Notes:
Appendix A. Checkpoint Solutions

Unit 1
1. OS/390, VM, VSE, AIX, OS/2, OS/400, Windows 95, Windows NT, Solaris, HP-UX, Siemens-Nixdorf and Linux
2. True
3. Column functions \texttt{MIN()}, \texttt{MAX()}, and \texttt{AVG()}
4. \texttt{GROUP BY} clause

Unit 2
1. Create a unique index
2. True
3. A primary key on the parent table
   A foreign key on the dependent table
4. Create a trigger
5. \texttt{ORDER BY} and \texttt{UNION}
6. False
   Views can be based on single views or on joins of tables, joins of views, or joins of tables and views.
7. No, the view will be read-only. However, you can update the base tables directly if you have sufficient authority.
8. \texttt{CHECK OPTION}

Unit 3
1. A left or right outer join
2. False
   Each row from the first table will be combined with EVERY row of the second table. The result is called Cartesian Product.
3. No
   You can also use \texttt{EXCEPT} to get the desired result. At present, \texttt{EXCEPT} is only supported on workstation platforms.

Unit 4
1. True
2. True
Unit 5
1. ALL, ANY, and SOME
2. True
3. A correlated subquery

Unit 6
1. The SUBST() function
2. Use the DIGITS() function to convert the value into an alphanumeric format. Then, use the SUBSTR() function to extract the digits to the right of the decimal point.
3. A timestamp consists of a date, a time, and a number of microseconds. It is used to identify exactly when something happened, to one millionth of a second.
4. DAYS() determines the difference in days between the date in the argument of the function and January 1, 0001.

Unit 7
1. True
2. Not necessary to CREATE table expressions.
   Table expressions can use host variables.
   Not necessary to grant access to table expressions.
3. True
4. It contains a common table expression that refers to itself within its definition.
   Other parts of the syntax are also important, such as the UNION ALL, but the provided answer is probably the “key” component.

Unit 8
1. False
2. If no index is available, DB2 has to read every row of the table.
3. Range-delimiting predicates and index-sargable predicates can be evaluated very efficiently in many cases.
4. In the worst cases, a correlated subquery performs the subquery - which may itself read many rows - once for each row which satisfies the WHERE of the outer query. In these cases, I/O is very high and the statement performs poorly.
Bibliography

• Manuals:
  - Cross-Platform Books
    SC26-8416  IBM SQL Reference
  - Platform Specific Books - DB2 UDB Server for OS/390
    SC26-9003  DB2 for OS/390 Version 6 Administration Guide
    SC26-9014  DB2 for OS/390 Version 6 SQL Reference
    SC26-9004  DB2 for OS/390 Version 6 Application Programming and SQL Guide
  - Platform Specific Books - DB2® Universal Database for UNIX, Windows and OS/2 Version 6
    SC09-2839  IBM DB2 Universal Database Administration Guide V6 Vol 1
    SC09-2840  IBM DB2 Universal Database Administration Guide V6 Vol 2
    SC09-2847  IBM DB2 Universal Database V6 SQL Reference Vol 1
    SC09-2848  IBM DB2 Universal Database V6 SQL Reference Vol 2
    SC41-5612  IBM DB2 for AS/400 SQL Reference
  - Miscellaneous
    SC26-3316  Formal Register of Existing Differences in SQL
    ISBN 1-55860-482-0 A complete Guide to DB2 Universal Database by Don Chamberlin

• WEB URLs:
  http://ourworld.compuserve.com/homepages/Graeme_Birchall
    DB2 UDB 6.2 SQL Cookbook

  http://www-4.software.ibm.com/data/db2
    IBM Software - DB2 Product Family
Index

A
adding columns 2-8
ALL 5-12
ALTER TABLE 2-8, 2-24
ANY 5-15
AS 1-11
AVG() 1-11

B
BETWEEN 1-9

C
CASCADE 2-23
catalog 1-7, 1-8
CHAR() 6-18, 6-20
check constraints 2-13
COALESCE() 6-9, 6-11
coding efficient statements 8-32
column constraint 2-14
common table expressions 7-11
correlated subqueries 8-23
correlated subquery 5-21, 5-22
COUNT(*) 1-11
COUNT(DISTINCT) 1-11
CREATE INDEX 2-18
CREATE TABLE 2-8
CREATE TRIGGER 2-29, 2-31

D
database 1-7, 1-8
DATE() 6-25, 6-30
DAY() 6-25, 6-30
DAYS() 6-25, 6-30
DB2 Catalog 1-5
DB2 Family 1-4
DECIMAL() 6-12
delete rules 2-23
dependent table 2-19
DIGITS() 6-16
Duration 6-37

E
EXISTS 5-18

F
foreign key 2-21
FULL OUTER JOIN 3-10

G
GROUP BY 1-13
grouping 1-13
HAVING 1-13
HOUR() 6-34

I
IN 1-9
index concepts 8-12
index structure 8-13
index utilization 8-18
INNER JOIN 3-5, 3-7

J
join 3-5, 3-7, 3-8, 3-9, 3-10, 3-11, 7-6

L
Labeled duration 6-37
LEFT OUTER JOIN 3-9
LENGTH() 6-21
LIKE 1-9
local predicates 3-14

M
MAX() 1-11
MICROSECOND() 6-34
MIN() 1-11
MINUTE() 6-34
MONTH() 6-25, 6-30

N
nested table expressions 7-6
NLEAF 8-13
NLEVELS 8-13
NO ACTION 2-23

O
optimizer 8-21
ORDER BY 1-9
outer join 3-8, 3-9, 3-10, 3-14
overview 1-23

P
parent table 2-19
Performance - General Guidelines  8-25, 8-26, 8-27, 8-28, 8-29, 8-30, 8-31
predicate  8-15
predicate examples  8-18
predicate types  8-16
primary key  2-21

R
read-only views  2-39
recursive SQL  7-13
referential constraint  2-21
Referential Integrity  2-19
RESTRICT  2-23
RIGHT OUTER JOIN  3-8
ROUND()  6-14

S
sargable  8-16
Scalar Functions  8-27
SECOND()  6-34
SELECT  1-9
self-referencing constraints  2-25
SET NULL  2-23
SOME  5-15
sorting - using index to avoid  8-18
subquery  5-12, 5-15, 5-18, 5-21, 5-22
subsystem  1-8
SUM()  1-11

T
table  1-7
table constraint  2-14
table expressions  7-4
TIME()  6-34
TIMESTAMP()  6-34
Topic overview  1-23
triggers  2-27

U
UDFs - the need  8-7
UDTs - defining  8-5
UDTs - the need  8-4
unique key  2-21
user-defined distinct types - defining  8-5
user-defined distinct types - the need  8-4
user-defined functions - the need  8-7

V
VALUE()  6-9, 6-11
view materialization  2-46
view merge  2-45
Views  8-26