IBM

International Systems Centers

IBM DATABASE 2 SQL USAGE GUIDE

GG24-1583

IBM DATABASE 2 SQL Usage Guide

Document Number GG24-1583-00

Per Groth, IBM Denmark

International Systems Center - Santa Teresa
San Jose, California

This publication was produced using the
IBM Document Composition Facility
(Program number 5748-XX9) and
the master was printed on the IBM 3800 Printing Subsystem

First Edition (June 1983)

This edition applies to IBM DATABASE 2 (DB2) Release 1 (Program Number 5740-XYR), Query Management Facility (QMF) Release 1 (5668-972), and Data Extract (DXT) Release 1 (5668-973).

Requests for copies should be made to the IBM branch office that serves you.

Forms for reader's comments are provided at the back of the publication. If the forms have been removed, comments may be addressed to:
IBM Corporation
International Systems Center - Santa Teresa
Department 471
P.O. Box 50020
San Jose, California 95150, U.S.A.
IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

This guide is the result of a residency conducted at the International Systems Center - Santa Teresa.

We would like to acknowledge the excellent work done by the author:

- Per Groth, IBM Denmark

Peter Backlund
Wes Dayton
Colin White

International Systems Center - Santa Teresa
June 1983

This document:

- **GG24-1583 IBM DATABASE 2 SQL Usage Guide**

  demonstrates the power of Structured Query Language (SQL), a data base management language which permits IBM DATABASE 2 (DB2) users to access and manipulate data in relational data bases. The document is intended for DP Professionals who wish to obtain a good functional knowledge of SQL. It covers SQL by using a series of examples starting with the very basic and becoming increasingly complex.

It is one in a series produced by the International Systems Center - Santa Teresa. Other documents in the series are:

- **GG24-1581 IBM DATABASE 2 Relational Concepts**

  which describes the relational approach to data base systems in general and to IBM DATABASE 2 in particular. The relationship between IMS/VS DL/I and DB2 applications is also covered. The intended audience is DP Professionals who wish to understand the relational model of data and how it is implemented in DB2.

- **GG24-1582 IBM DATABASE 2 Concepts and Facilities Guide**

  which gives a functional overview of the IBM DATABASE 2 relational data base management system. It is intended to be read by all DP Professionals who wish to obtain a good functional knowledge of the product.

CHAPTER 1. INTRODUCTION

Structured Query Language (SQL) is a data base management language which permits DB2 users to define, access, and manipulate data stored in large data bases.

The purpose of this guide is to demonstrate the power of SQL. The guide will show that the SQL language is not only a query language, but also a data manipulation language and a data definition and authorization language.

SQL is based on the relational model, where data is stored in tables. Since most people are used to viewing data as tables, this concept is very easy to learn and understand.

The SQL language is a high level language, where the user specifies **what** the result should be, **not how** the result should be obtained. This concept has a great impact on improved user productivity, since the user, in a few statements, can specify how a given problem should be solved.

The guide is divided into three parts:

- **Part A** is an overview of the SQL language. This part will briefly describe the three language functions (e.g. data manipulation, data definition, and data authorization control).

- **Part B** presents examples of using SQL. Examples are shown for each area of SQL covered in part A. The first chapter of this part describes a sample set of tables that will be used in the examples that follow in subsequent chapters.

- **Part C** consists of six appendixes. The first three appendixes show the syntax of the various SQL statements. The last three appendixes are listings of the sample tables used in the guide.

It should be noted that the SQL covered in this guide is restricted to that what can be entered using DB2I or QMF. The special functions of the language designed to cater for imbedding SQL in an application program are not covered.

It is assumed that the user is to some extent familiar with the SQL language. Also note that this guide does not give a complete coverage of SQL. For a more detailed description of the SQL language, please refer to the IBM DATABASE 2 Reference.

This part of the guide will give the reader an overview of the SQL language. The overview is divided into three chapters:

- **Chapter 2. Data Manipulation Language**

  This chapter describes how data in tables can be retrieved using various search conditions. The first part of the chapter shows basic queries, where we are selecting data from one table. The second part of the chapter will familiarize the reader with the advanced features of the SQL. Here we will see how data can be derived from several tables, and how we can insert new data, change existing data, and finally how data can be deleted.

- **Chapter 3. Data Definition Language**

  This chapter describes how tables and other DB2 "objects" can be defined, changed, and deleted using the dynamic SQL data definition language. The chapter will also give an overview of the data types allowed in SQL.

- **Chapter 4. Data Control Language**

  The last chapter in this part is an overview of the authorization mechanism in DB2. We will see how table access can be dynamically granted and revoked.

CHAPTER 2. DATA MANIPULATION LANGUAGE

Structured Query Language (SQL) is a query language that gives a user access to data stored in DB2 tables. The data manipulation part of the language enables the user to make queries without changing the data in the tables, but it also allows the user to modify existing data, to insert new data, and to delete data from the tables. A DB2 table consists of zero or more rows, each row consisting of one or more columns.

Queries are handled with the SELECT statement, modifications with the INSERT, UPDATE and DELETE statements. In this chapter we will discuss:

- Basic Queries using SELECT
- Advanced Queries using SELECT
- INSERT
- UPDATE
- DELETE

BASIC QUERIES USING THE SELECT STATEMENT

The purpose of a query is to retrieve information from DB2 tables. The retrieval is made with a **SELECT** statement. The result of this operation will always be a _table_ consisting of the selected columns and rows. The following must be specified in the SELECT statement:

- What data to retrieve, i.e. the field(s) that should be returned as result of the SELECT.
- The table(s) from which the data is to be retrieved.

Optionally the SELECT can specify:

- The conditions which must be satisfied in order to retrieve the data.

A SELECT statement may also be issued against a **view**. A view is like a window through which the user can access a table, a part of a table, or a combination of two or more tables. Views look like real tables, but they do not contain data. Instead the data is derived from the underlying table(s), on which the view is defined.

The basic form of the **SELECT** statement is:

```
SELECT  the data you want (column name(s))
  FROM  some source (table name(s))
 WHERE  conditions which are to be met (if any)
```

The specified list of data you want can be simple column names; it can be special built-in functions; or it can be arithmetic expressions, like the sum of values from two columns.

Following is an example of a simple SELECT statement:

```
SELECT FIRSTNME, LASTNAME
  FROM TEMPL
```

This statement will retrieve the names of all employees in the TEMPL table[1]. Appendix E in this guide contains listings of the employee and department tables. Appendix F contains listings of four project related tables that will be used later in this guide.

Search Conditions in the WHERE-clause

Using the **WHERE** clause enables selection of only certain employees as the next example shows.

```
SELECT FIRSTNME, LASTNAME
  FROM TEMPL
 WHERE WORKDEPT = 'D11'
```

Here only employees of department 'D11' will be selected.

The columns (fields) will be presented in the same order as they are specified in the SELECT clause. A shorthand notation exists which will present all columns of a table or a view. To use this, the specification of columns is replaced by an '*' as in the following example:

[1] A table name can be prefixed by an authorization-id (or user-id). It is assumed that all tables in this guide are owned and accessed by the user, DSN8, therefore no prefix is needed. If another user (not DSN8) is allowed to access any of these tables, this user must specify the fully qualified table name, e.g. DSN8.TEMPL.

```
SELECT *
  FROM TEMPL
 WHERE WORKDEPT = 'D11'
```

The previous examples will return all rows which satisfy the given WHERE clause (if any). It is possible that several of the selected rows will be identical (i.e. all the specified columns have identical values). To suppress presentation of identical rows, one can use the DISTINCT keyword. This is a positional keyword which must immediately follow the SELECT verb. The alternative to DISTINCT is ALL which is the default value. The following example shows the format:

```
SELECT DISTINCT LASTNAME
  FROM TEMPL
 WHERE LASTNAME = 'BROWN'
```

By using ALL (default) in the above example, the query will return zero, one, or more rows where LASTNAME is 'BROWN'. If DISTINCT is specified, only zero or one row will be returned with LASTNAME equal to 'BROWN'.

The following example will only eliminate duplicate rows having both FIRSTNME and LASTNAME identical. This means that we might retrieve several rows, where the last name is BROWN, but they will all have different FIRSTNME values:

```
SELECT DISTINCT FIRSTNME, LASTNAME
  FROM TEMPL
 WHERE LASTNAME = 'BROWN'
```

In the preceding examples, the constants used in the WHERE clauses have been character strings. When a character string is specified in an SQL statement, it must be enclosed in quotes. When a numeric constant is used, it is specified without quotes. For a description of the data types allowed in SQL, please refer to "Creating a New Table" on page 26.

Combining Search Conditions (AND, OR)

If a select is made on more than one condition, the conditions can be combined by using the **AND** and **OR** keywords. The following example shows a query, where we are selecting employees with a job code of 56 from department D11:

```
SELECT EMPNO, FIRSTNME, LASTNAME
  FROM TEMPL
 WHERE WORKDEPT = 'D11'
   AND JOBCODE  =  56
```

By using the **AND** keyword, the previous example will only select those employees that meet both search conditions. The **OR** keyword will return all rows where at least one of the specified conditions are satisfied. However, if a row satisfies several conditions, it will only be returned once. In the following example, all employees from department D11 as well as all employees having a job code of 56 will be selected:

```
SELECT EMPNO, FIRSTNME, LASTNAME
  FROM TEMPL
 WHERE WORKDEPT = 'D11'
    OR JOBCODE  =  56
```

Several conditions can be combined in the WHERE-clause. Each of these conditions is called a predicate. You may have to use parentheses in order to override precedence rules of operators. This is shown in the following example, where we are selecting employees who either work in department D11 or D21 and who have a job code greater than 54 or an education level greater than 15:

```
SELECT EMPNO, FIRSTNME, LASTNAME
  FROM TEMPL
 WHERE ( WORKDEPT = 'D11' OR WORKDEPT = 'D21' )
   AND ( JOBCODE  >  54   OR EDUCLVL  > 15 )
```

In the last example the greater-than (>) comparison operator was used in the search condition. SQL will allow the following comparison operators to be used in a search condition:

```
=  ¬=  >  >=  ¬>  <  <=  ¬<
```

The special not-operator "¬" can be used to negate a comparison as shown. The **NOT** keyword can be used in a WHERE-clause as a normal Boolean operator. In the following example we are selecting employees who have a job code of 54 and who are working in any department except department D11:

```
SELECT FIRSTNME, LASTNAME, WORKDEPT, EDUCLVL
  FROM TEMPL
 WHERE JOBCODE = 54
   AND NOT WORKDEPT = 'D11'
```

This statement could instead of the NOT keyword have used the ¬= operator in the second search condition:

```
SELECT FIRSTNME, LASTNAME, WORKDEPT, EDUCLVL
  FROM TEMPL
 WHERE JOBCODE = 54
   AND WORKDEPT ¬= 'D11'
```

Search Condition Using BETWEEN

The **BETWEEN** predicate can be used in a WHERE-clause together with other expressions. BETWEEN can be preceded by NOT. You would typically use BETWEEN when you want to select a field within a range of values. In the next example we are selecting all employees having an employee number between 000100 and 000230:

```
SELECT EMPNO, FIRSTNME, LASTNAME, WORKDEPT
  FROM TEMPL
 WHERE EMPNO BETWEEN '000100' AND '000230'
```

Note that employees with the numbers 000100 and 000230 will be included in the result.

Search Condition Using IN

The **IN** predicate makes it possible to compare the contents of a field with a list of values. The predicate is satisfied if the field is equal to any of the constants in the list. If NOT IN is specified, the predicate is satisfied if the field is not equal to any of the listed items. In the following example all employees with an education level of 16, 18, or 20 will be selected:

```
SELECT EMPNO, FIRSTNME, LASTNAME, EDUCLVL
  FROM TEMPL
 WHERE EDUCLVL IN (16, 18, 20)
```

Search Condition Using LIKE

The **LIKE** predicate enables the user to search for character string data which partially matches a specified string. In this string the "%" character represents any string of zero or more characters, the "_" character represents exactly one single character. NOT LIKE is a valid predicate. In the following example we are selecting all employees having a last name starting with a "P" and who are working in a department, where the first character in the department number is "D" and the third character is "1":

```
SELECT EMPNO, FIRSTNME, LASTNAME, WORKDEPT
  FROM TEMPL
 WHERE LASTNAME LIKE 'P%'
   AND WORKDEPT LIKE 'D_1'
```

Search Condition Using NULL

The **NULL** predicate provides a way for the user to explicitly look for null values in the base table, or, if NOT NULL is used, exclude null values from the query result. A null value is a special value that indicates an absence of a value. A null value cannot be used in a comparison, since null is not greater than, smaller than, equal to, or not equal to any other value (including another null value). In the following example all rows containing a null value in the JOBCODE column should be selected:

```
SELECT EMPNO, FIRSTNME, LASTNAME
  FROM TEMPL
 WHERE JOBCODE IS NULL
```

Search Condition With Combined Predicates

The predicates discussed above can be combined in any order in the same WHERE-clause. The following query example illustrates a combination of the predicates explained so far:

```
SELECT LASTNAME, JOBCODE, EDUCLVL, SALARY, WORKDEPT
  FROM TEMPL
 WHERE ( WORKDEPT = 'D11' OR WORKDEPT = 'E21' )
   AND EDUCLVL IN (12, 14, 16, 18)
   AND SALARY BETWEEN 15600 AND 23700
   AND ( LASTNAME NOT LIKE 'P%' OR LASTNAME LIKE '%SON%')
   AND JOBCODE IS NOT NULL
```

Use of Built-In Functions

Five built-in functions can be used in a SELECT-clause. These functions are: **AVG**, **MAX**, **MIN**, **SUM**, and **COUNT**. The argument of a built-in function may be a column name or an expression. An expression is a combination of column names and or constants (e.g. SALARY/12). If duplicate values should be eliminated when using the COUNT built-in function, the column name must be preceded by DISTINCT. COUNT may have a special format - COUNT(*) - which will compute the number of rows satisfying the search condition. In the following example the number of employees in department D11 will be counted. Further, the average, the maximum, the minimum, and the total salary for department D11 will be calculated. Finally, the number of different job codes in that department will be counted:

```
SELECT COUNT(*), AVG(SALARY), MAX(SALARY),
                 MIN(SALARY), SUM(SALARY),
                 COUNT(DISTINCT JOBCODE)
  FROM TEMPL
 WHERE WORKDEPT = 'D11'
```

Ordering of Result Using ORDER BY

The result of a SELECT statement will be presented in a system determined order. By using the **ORDER BY**-clause the user can specify how the result should be ordered. The default ordering is ascending (ASC), but descending order can be specified (DESC). The user can request ordering on one or more items (columns or expressions) in the SELECT-clause by specifying either a column name or an integer number denoting an element in the result list. In the following example we are selecting all employees in departments A00, B01, C01, and D01 ordered on department number. Within a department, the rows will be ordered in descending order using the monthly salary (the 3rd column in the result) as the secondary ordering criteria:

```
SELECT LASTNAME, WORKDEPT, SALARY / 12
  FROM TEMPL
 WHERE WORKDEPT = 'A00' OR WORKDEPT = 'B01'
    OR WORKDEPT = 'C01' OR WORKDEPT = 'D01'
 ORDER BY WORKDEPT, 3 DESC
```

ADVANCED QUERIES USING THE SELECT STATEMENT

All queries discussed so far have been based on only one table, which is specified in the FROM-clause of the query. In the following examples we will combine data from several tables and look into the advanced features of the SELECT statement.

The following features will briefly be covered:

- Join of Several Tables
- Use of Grouping
- Use of Subselects
- Correlation
- Testing for Existence
- Union of Results
- Constants

JOIN of Several Tables

In some cases the data that is needed in a query will not be available in one table only. That means you will have to combine data from two or more tables in order to get the data you need. This process is called **Joining**. The JOIN feature of DB2 permits a query to be written against the combined data in two or more tables. All the tables participating in the join should be listed in the FROM-clause (e.g. FROM TEMPL, TEMPRAC). Conceptually, DB2 forms all possible combinations of rows from the indicated tables, and for each combination tests the condition in the WHERE-clause. The WHERE-clause usually specifies some relationship between the rows to be joined, e.g. their department number values must match:

```
WHERE TEMPL.WORKDEPT = TDEPT.DEPTNO
```

A predicate such as this, which specifies some relationship between two tables, is called a **JOIN**-condition.

The employee table we have been using in the previous examples does not include the department name. If we would like to produce a list containing all employee names, department numbers, and department names, we will have to retrieve the department names from the department table. This can be done by joining the two tables, as the following example shows:

```
SELECT FIRSTNME, LASTNAME, WORKDEPT, DEPTNAME
  FROM TEMPL, TDEPT
 WHERE WORKDEPT = DEPTNO
```

If the WHERE-clause was omitted in the above example, DB2 would return a result containing all possible combinations of rows from the tables in the FROM-clause. This type of result is called a "Cartesian Product". It does not normally produce a meaningful result.

Joining Tables With Identical Column Names

If tables with identical column names are joined, you must qualify the column names with the table names in the SELECT- and WHERE-clauses. If the DEPTNO column in the department table instead was named WORKDEPT, we would have to qualify this column name, in order to do the same select as above:

```
SELECT FIRSTNME, LASTNAME, TDEPT.WORKDEPT, DEPTNAME
  FROM TEMPL, TDEPT
 WHERE TDEPT.WORKDEPT = TEMPL.WORKDEPT
```

Joining a Table To Itself

In some cases it will be necessary to join a table to itself. Another way to view this process is to imagine two (or more) "virtual" tables based on the same base table. The table-name is repeated two or more times in the FROM-clause, indicating that the join consists of combinations of two or more rows from the same table. Since the table name will not be unique, each table name in the FROM-clause must be given a unique label.

In the following example we are selecting pairs of employees, who work in the same department, but have a difference in the job code value of at least 5. Since the employee table is joined to itself, we will apply the labels X and Y to the table name:

```
SELECT X.LASTNAME, X.JOBCODE,
       Y.LASTNAME, Y.JOBCODE
  FROM TEMPL X, TEMPL Y
 WHERE X.WORKDEPT = Y.WORKDEPT
   AND X.JOBCODE >= Y.JOBCODE + 5
```

Using GROUP BY

The **GROUP BY**-clause is used to divide the rows of a table into groups having a matching value in one or more columns. One or more of the built-in functions is then applied to these groups. When the GROUP BY-clause is used in a SELECT statement, only one row is returned for each group. It is therefore not possible - nor meaningful - to include column names not referenced in the GROUP BY-clause in the select list. Not referenced column names can be used in built-in functions only. In the following example, the maximum, the minimum, and the average salary is computed for all male employees in each department:

```
SELECT WORKDEPT, MAX(SALARY), MIN(SALARY), AVG(SALARY)
  FROM TEMPL
 WHERE SEX = 'M'
 GROUP BY WORKDEPT
```

The **HAVING**-clause is used to qualify the grouping. This clause will filter groups so that conditions specified in the HAVING-clause are met. Several group qualifying predicates, connected by ANDs and ORs, may be specified. If the HAVING-clause is used without a GROUP BY-clause, the whole table will be treated as one group. Using the previous example, we will now only list the departments having more than two male employees and only if the maximum salary is more than twice the minimum salary:

```
SELECT WORKDEPT, MAX(SALARY), MIN(SALARY), AVG(SALARY)
  FROM TEMPL
 WHERE SEX = 'M'
 GROUP BY WORKDEPT
HAVING COUNT(*) > 2
   AND MAX(SALARY) > 2 * MIN(SALARY)
```

Using Subselects Within a Query

The SELECT statement may refer to a value or a set of values which will be derived from a subselect. A subselect may also have a subselect. There is no limitation to the number of subselect levels. A subselect will typically be part of the WHERE-clause, but may also appear in the HAVING-clause. If the subselect returns a single value, it can be used on the right side of the comparison operator in any predicate in the WHERE- or HAVING-clause. The subselect must have exactly one expression that describes what is being selected. Let's look at an example, where we are selecting all the female employees, who are younger than the average employee (both female and male):

```
SELECT LASTNAME, FIRSTNME, BRTHDATE
  FROM TEMPL
 WHERE SEX = 'F'
   AND BRTHDATE > ( SELECT AVG(BRTHDATE)
                      FROM TEMPL )
```

Since the subselect will be executed first, a comparison can be made between the average birth date and the birth date found in each row from the outer level query. Note that the subselect must be enclosed in parentheses.

In the above example the subselect only returned one value. In the following examples the subselect might return more than one value. When more than one value can be returned, you must use the ALL, ANY, or IN keyword in the outer level WHERE-clause.

The **ALL** and **ANY** keywords can be used in a comparison when the subselect returns more than one value. If ALL is used in a comparison, the condition is satisfied if the given expression is true for all the values in the returned set. If ANY is used in a comparison, the condition is satisfied if the given expression is true for any of the values in the returned set. In the following example we are selecting all the employees, who have been working longer in the company than all the members of department C01, and who are younger than any member of department E21:

```
SELECT EMPNO, LASTNAME, WORKDEPT
  FROM TEMPL
 WHERE HIREDATE < ALL ( SELECT HIREDATE
                          FROM TEMPL
                         WHERE WORKDEPT = 'C01' )
   AND BRTHDATE > ANY ( SELECT BRTHDATE
                          FROM TEMPL
                         WHERE WORKDEPT = 'E21' )
```

If the subselect returns a set of values, the **IN** predicate can be used in the WHERE-clause. In the following example we are selecting all managers from the employee table. The manager numbers are found in the department table:

```
SELECT EMPNO, LASTNAME, WORKDEPT
  FROM TEMPL
 WHERE EMPNO IN ( SELECT MGRNO
                    FROM TDEPT
                   WHERE MGRNO ¬= '' )
```

Using Correlated Subselects

In the previous examples, all the subselects have been executed once and the resulting value(s) has been substituted into the WHERE-clause. In some queries, however, it is necessary to evaluate the subquery for each row in the outer-level query. This type of query is called a "correlated subselect".

A **correlated subselect** is a subselect that is executed repeatedly, once for each row returned by the outer-level select. A query with a correlated subselect has the same format as an ordinary outer query with a subselect. A "correlation name" is appended to the table name in the FROM-clause and is used to qualify a column name in the subselect with the outer-level table name. This means that the subselect will be re-evaluated for each row of the outer query, if the outer query has different values for a given selected column name.

In the following example we are selecting employees having an education level higher than the average for the department in which the employee is working:

```
SELECT EMPNO, LASTNAME, WORKDEPT  <------┐
  FROM TEMPL X                           |
 WHERE EDUCLVL > ( SELECT AVG(EDUCLVL)   |
                     FROM TEMPL          v
                    WHERE WORKDEPT = X.WORKDEPT )
```

The column name X.WORKDEPT in the inner-level SELECT will correlate to the WORKDEPT field in the outer-level SELECT.

Testing for Existence

The **EXISTS** predicate is used to test for the existence of a row or rows satisfying some condition in a subselect. The EXISTS predicate can be preceded by the NOT keyword. The predicate is used to link a subselect to the outer-level query, but the subselect will not return a value - only a true or false indicator. The true condition is set if the subselect returns one or more rows. If the subselect does not return any rows, the false indicator will be set. In the following example we will use a correlated query in selecting all employees who are not managers (i.e. the employee number should not be found in the department table). The resulting list will be ordered on the last name:

```
SELECT EMPNO, LASTNAME, WORKDEPT
  FROM TEMPL X
 WHERE NOT EXISTS ( SELECT *
                      FROM TDEPT
                     WHERE MGRNO = X.EMPNO )
 ORDER BY LASTNAME
```

The subselect specifies that all columns should be selected (use of the "*"). Since the subselect does not return values to the outer-level query, but only a true or false indicator, any column name(s) could have been specified instead.

Union of SELECT Results

The **UNION** operator is used to combine the results from two or more outer-level queries. The outer level queries are first executed, then the results are combined and duplicate rows are eliminated. In order to be combined, the data types of corresponding items selected by all the SELECT statements must be identical (i.e. have the same defined attributes). If ordering is required, the ORDER BY clause must be written after the last query in the UNION and will apply to the entire result. Since the column names in the outer level queries in most cases will be different, you must specify a column number of the result and not a column name. In the following example we will select the employee numbers of all the employees having a senior level (i.e all managers and all employees having a job code of 56 or more):

```
SELECT MGRNO
  FROM TDEPT
 WHERE MGRNO ¬= ''
 UNION
SELECT EMPNO
  FROM TEMPL
 WHERE JOBCODE >= 56
 ORDER BY 1
```

Use of Constants

A **constant** is a fixed character string that can be included in the select list, just like a column name. To specify a string constant, you must enclose the character string within simple quotes (e.g. 'Text String'). When a constant is specified, the result table will have a column containing the constant; this column will have the VARCHAR attribute (for data types and attributes, please refer to "Creating a New Table" on page 26).

Constants are normally used in connection with the UNION operator in order to add a description to each row in each of the outer-level queries. Constants in two or more SELECT statements - that are combined - _must_ have the same length. Constants can only be combined with columns defined as VARCHAR and the length of the constant _must_ be the same as the maximum length of the column value.

In the following example we will do the same UNION as shown in the previous sub-chapter, but we will now add a description to each row:

```
SELECT MGRNO, 'MANAGER    '
  FROM TDEPT
 WHERE MGRNO ¬=''
 UNION
SELECT EMPNO, 'SENIOR EMP.'
  FROM TEMPL
 WHERE JOBCODE >= 56
 ORDER BY 1
```

As the example shows, the constants in the SELECT statements both have a length of 11 characters.

THE INSERT STATEMENT

The purpose of the **INSERT** statement is to add one or more rows to a table. The data to be inserted can be specified directly in the INSERT statement, or data can be retrieved from an existing table by using a subselect statement.

The two basic forms of the **INSERT** statement are:

```
INSERT INTO    the table (table name)
               (list of column name(s))
      VALUES   (list of values)


INSERT INTO    the table (table name)
               (list of column name(s))
      SELECT   the data you want (column name(s))
        FROM   some source (table name(s))
       WHERE   conditions which are to be met (if any)
```

Simple INSERT of One Row

By using the first format of the INSERT statement, we will now add a new row in the department table. Let's assume that Spiffy Computer Services Division has decided to establish a department for testing. The department number is D41, the name is Systems Test, James Walker has been appointed as manager, and the new department will report to department D01. Since all columns in the table are assigned a value, and since we are inserting columns in correct order, we need not specify the list of column names:

```
INSERT INTO TDEPT
     VALUES ('D41', 'SYSTEMS TEST',
             '000190', 'D01')
```

If not all field values are known at INSERT time, you can either specify in the list of column names only the columns that will receive a value, or you can omit the list of column names, and instead use the NULL keyword for fields not assigned a value.

Let's assume that the Systems Test department has hired two new employees. At this time we only know the names, the work department, the hire dates, and the sex codes. The following example will show the two ways of inserting a simple row in the employee table as discussed in the previous paragraph:

```
INSERT INTO TEMPL
            (EMPNO, FIRSTNME, MIDINIT, LASTNAME,
             WORKDEPT, HIREDATE, SEX)
     VALUES ('000410', 'HOWARD', ' ', 'JENSEN',
             'D41', 820308, 'M')

INSERT INTO TEMPL
     VALUES ('000420', 'CAROLE', 'H', 'GORDON',
             'D41', NULL, 820312, NULL, NULL,
             'F', NULL, NULL)
```

Note that both INSERT statements will insert NULL values in the column names not assigned a value.

Selected INSERT of Multiple Rows

By using the second format of the INSERT statement, multiple rows can be inserted by a single INSERT statement. The subselect statement may contain any of the features described previously, except the UNION operator.

Let's assume that we have created a special employee table for the new Systems Test department. This table is called EMPD41. The following example shows how to insert the two employees and the manager assigned to D41. The data will be derived from our current employee and department table:

```
INSERT INTO EMPD41
            (EMPNO, FIRSTNME, MIDINIT, LASTNAME,
             WORKDEPT, HIREDATE, SEX)
     SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME,
            WORKDEPT, HIREDATE, SEX
       FROM TEMPL
      WHERE EMPNO LIKE '0004%'
         OR EMPNO IN ( SELECT MGRNO
                         FROM TDEPT
                        WHERE DEPTNO = 'D41' )
```

THE UPDATE STATEMENT

The purpose of the **UPDATE** statement is to update the values of one or more columns in one or more rows of a table. The rows to be updated are chosen by the search condition, as described in the SELECT section (see "Basic Queries using the SELECT Statement" on page 5).

The basic form of the **UPDATE** statement is:

```
  UPDATE  the table (table name)
     SET  field name 1 = expression 1,
          field name 2 = expression 2, ...
   WHERE  conditions which are to be met (if any)
```

If no search condition is specified, all rows in the named table will be updated.

In the following example, the job code and the salary will be updated for the two new employees in department D41. Since the two employees have the same job code and the same salary, we can update the two rows in one UPDATE statement. In the same statement, we will also change the hire dates to null values:

```
  UPDATE TEMPL
     SET JOBCODE = 52, SALARY = 18140,
         HIREDATE = NULL
   WHERE EMPNO = '000410'
      OR EMPNO = '000420'
```

In the following example, all employees will be given a 7.5% salary increase. Since this update is for all the rows in the employee table, no search condition is needed:

```
  UPDATE TEMPL
     SET SALARY = 1.075 * SALARY
```

THE DELETE STATEMENT

The purpose of the **DELETE** statement is to delete one or more rows in a table. The rows to be deleted are chosen by the search condition, as described in the SELECT section (see "Basic Queries using the SELECT Statement" on page 5).

The basic form of the **DELETE** statement is:

```
 DELETE
   FROM  some source (table name)
  WHERE  conditions which are to be met (if any)
```

Since the smallest item that can be deleted by the DELETE statement is a <u>row</u>, no column specifications can be given. If only a column value in a row has to be "deleted", you should change this value to NULL by means of the UPDATE statement.

In order to illustrate the use of the DELETE statement, let's assume that Spiffy Computer Services Division decides to abandon the idea of establishing a Systems Test department. The two new employees are offered another job elsewhere in the parent company, and will therefore be deleted in the Spiffy Computer Services Division employee table. The following example shows how the DELETE operation is specified:

```
 DELETE
   FROM TEMPL
  WHERE EMPNO = '000410'
     OR EMPNO = '000420'
```

In the following example we are deleting department D41 in the department table. We will use the employee numbers from our special D41 table in order to find the manager number for the Systems Test department:

```
 DELETE
   FROM TDEPT
  WHERE MGRNO IN ( SELECT EMPNO
                     FROM EMPD41 )
```

The purpose of this example is to illustrate that a DELETE statement can have a subselect. Of course we could have specified the WHERE-clause as: WHERE DEPTNO = D41

Finally, we will delete all rows in the special D41 employee table. After this operation the table will be empty, but will still exist:

```
 DELETE
   FROM EMPD41
```

CHAPTER 3. DATA DEFINITION LANGUAGE

The Data Definition Language is used for data administration. SQL has a set of statements that will allow the authorized user to define, change, and delete objects such as tables, indexes, and views. Through the use of data definition statements, data administration can be performed dynamically while DB2 is executing.

In this chapter, we will discuss how objects are created (CREATE), how we can alter existing object definitions (ALTER), how objects can be dropped (DROP), and finally how an explanatory comment can be added to the DB2 description of a table, view, or column (COMMENT ON).

The following types of DB2 objects can be manipulated by the authorized user:

- Storage Group
- Data Base
- Tablespace
- Table
- Index
- View
- Synonym

A description of the the DB2 objects and the relationship between them can be found in the IBM DATABASE 2 Data Base Planning and Administration Guide. The following figure is an overview of the relationship between Storage Group, Data Base, Table Space, Index Space, Table, and Index in a DB2 system. As the figure shows, the table space TS2 is a partitioned table space spanning two storage groups. Each partition contains a portion of the T21 table. From an operational point of view, the Data Base object is used to start and stop access to tables.

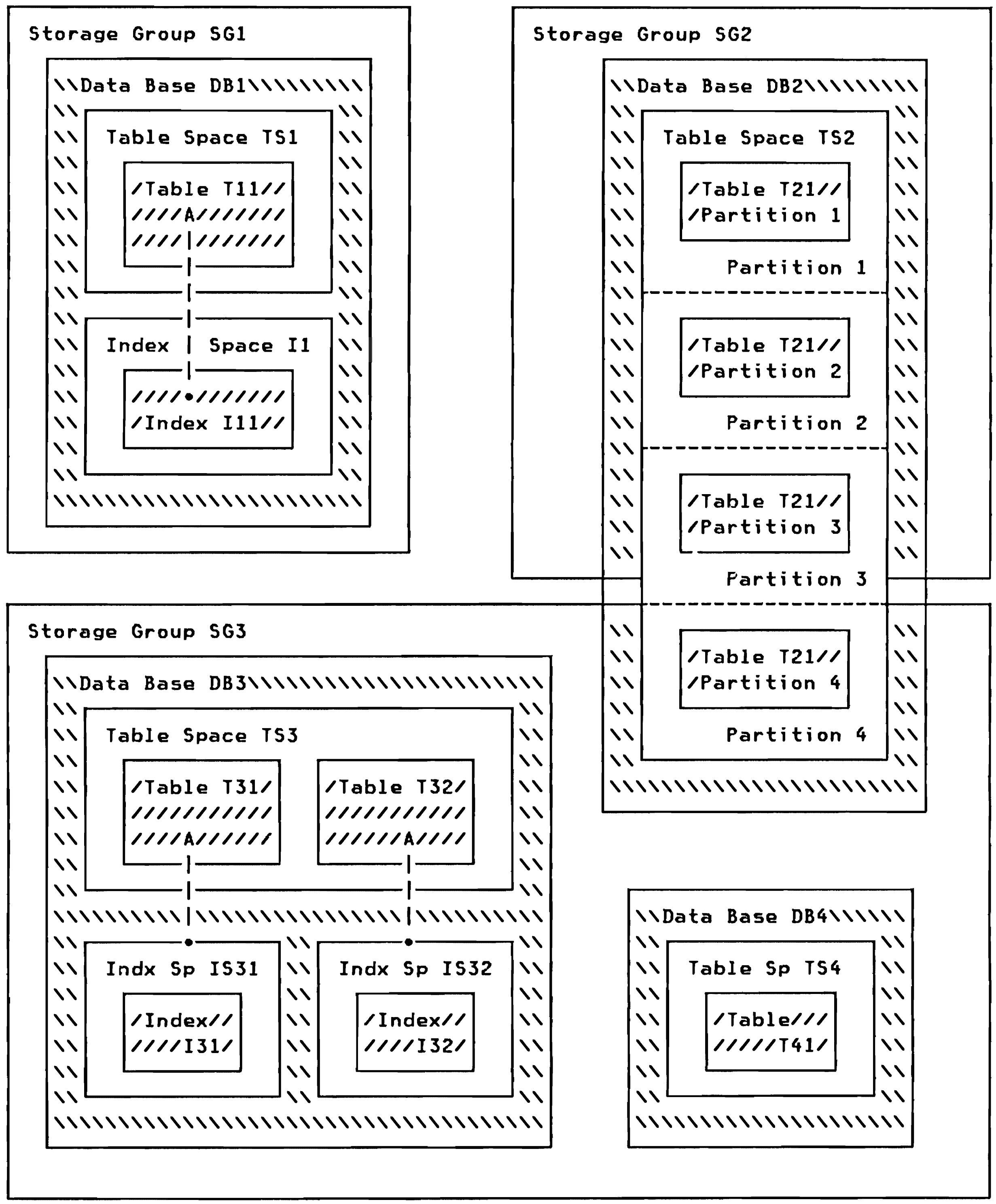

Figure 1. Example of relationships between the DB2 objects

The DB2 definition statements are CREATE, DROP and ALTER. Figure 2 gives an overview of the Data Definition Language (DDL) statements that can be applied to the various DB2 objects.

OBJECT	CREATE	ALTER	DROP
STOGROUP	Y	Y	Y
DATABASE	Y	-	Y
TABLESPACE	Y	Y	Y
TABLE	Y	Y	Y
INDEX	Y	Y	Y
VIEW	Y	-	Y
SYNONYM	Y	-	Y

Figure 2. DDL operations

In the following description of the DDL statements, we will assume that the SQL user is only given authority to issue DDL statements against tables, indexes, views, and synonyms. The use of DDL statements against storage groups, data bases, and table spaces is covered in the <u>IBM DATABASE 2 Data Base Planning and Administration Guide</u>.

THE CREATE STATEMENT

The purpose of the **CREATE** statement is to create an object in DB2. In order to issue the CREATE statement, the user must have the proper authority. In the following we will assume that SYSADM[2] has granted the user authority to create tables, views, and indexes.

Creating a New Table

The **CREATE TABLE** statement creates a new table. The table name and the column names plus their attributes must be specified. Optionally, a table space name and data base name can be specified.

The attributes for every column in the table must be specified in the **CREATE TABLE** statement. These attributes can either specify character data or numeric data:

CHARACTER DATA: The following two data types allow all EBCDIC characters:

CHAR	Fixed length character string (max. 254 characters)
VARCHAR	Variable length character string (max. 32,674 characters). The actual maximum depends on the page size of the table space (which in turn depends on the buffer pool specified); on whether an EDITPROC is specified; and whether the 'NOT NULL' is specified.

NUMERIC DATA: The following four data types are for numeric data only:

INTEGER	Fullword integer numbers in the range -2,147,483,648 to 2,147,483,647
SMALLINT	Halfword integer numbers in the range -32,768 to 32,767
FLOAT	Double word floating point numbers with a size limit of approximatively 10^{75}.
DECIMAL	Fixed point packed decimal numbers with at most 15 digits (999,999,999,999,999).

When a **CREATE TABLE** statement is issued, you may specify the name of a validation procedure. This routine will validate the values before they are inserted. Similarly, you may specify the name of an edit routine. The purpose of this routine is to compact, alter, or encrypt data after it is retrieved and before it is stored.

[2] SYSADM is a user-id of a user with total control over any DB2 resource. SYSADM can grant authority at different levels to other users. This authority may later be changed or revoked by SYSADM.

In a previous example, we inserted rows in a new table - see "Selected INSERT of Multiple Rows" on page 20. This table should only hold information on the names of the employees, the work departments, phone numbers, the hire dates, the sex codes, and the salaries. Let's suppose that all employee numbers must be in the range of 401 to 499, for that purpose we will specify a validation routine by the name of VALIDD41. For security reasons we will further encrypt the data using an edit routine named EDITD41. In the following example, the **CREATE TABLE** statement for the new table, named EMPD41, is shown:

```
CREATE TABLE EMPD41
           (EMPNO    CHAR(6)      NOT NULL,
            FIRSTNME VARCHAR(12)  NOT NULL,
            MIDINIT  CHAR(1)      NOT NULL,
            LASTNAME VARCHAR(15)  NOT NULL,
            WORKDEPT CHAR(3)      NOT NULL,
            PHONENO  CHAR(4),
            HIREDATE DECIMAL(6),
            SEX      CHAR(1),
            SALARY   DECIMAL(8,2) )
           EDITPROC  EDITD41
           VALIDPROC VALIDD41
```

Note that the first five columns cannot contain null values. This means that you <u>must</u> insert values in these columns. In the last four columns a null value will be inserted, if no data is supplied for the columns.

<u>Creating an Index</u>

The **CREATE INDEX** statement creates a new index. If you are creating an index on one of your own tables, you need no special authority. In order to create indexes on other tables, you must have been granted INDEX authority.

In the following example, we are creating an index on the employee number column in our new table EMPD41. Since all employees must have different employee numbers, we will specify that the contents of this column must be unique. Let's assume that this index is the first to be specified for the given table. In that case, the index will be a "clustered" index, i.e. the physical sequence of rows is as much as possible ordered on the employee number. If you have several indexes defined for a table, <u>one</u> index can be defined to be the clustered index (specify the CLUSTER parameter as in the following example). As we normally will list the table in ascending order, the ASC keyword is specified after the EMPNO column name (since ascending order is the default, the ASC keyword could have been left out).

The PAGESIZE parameter specifies the locking granularity of the index. The DSETPASS parameter specifies the VSAM password for the index dataset:

```
CREATE UNIQUE INDEX XEMPD41
                 ON EMPD41
                   (EMPNO  ASC)
                    CLUSTER
                    PAGESIZE .5 KB
                    DSETPASS DSN8
```

Creating a View On a Table

The **CREATE VIEW** statement is used to define a view on a base table. A view is a "virtual" table that is derived from one or more tables, or it may be derived from other views or combinations of tables and views. A view does not contain data, but data will be retrieved from the underlying base tables, when data manipulation is performed.

One of the advantages of views is that subsets of tables can be defined. It is possible to define subsets on rows, on columns and on a combination of rows and columns. A view may include columns which are not part of any underlying table. Such a column could be the sum of values from two other columns, or it could be the average value of a group of rows.

A base table can be updated through the use of a view, if the view is a simple row and column subset of a single base table. When a view is defined, a **CHECK OPTION** parameter can be specified. This parameter will verify all updates made against the view. If a view has defined a subset of rows, this parameter will restrict you from making updates that would cause rows to 'disappear' from the view.

The basic form of the **CREATE VIEW** statement is:

```
CREATE VIEW    a view-name
               (column names in view)
    AS SELECT  the data you want (column name(s))
         FROM  some source (table/view name(s))
        WHERE  certain conditions (if any) are to be met
        [WITH CHECK OPTION]
```

In the following example we will define a simple view on the employee table. This view will contain all the data for employees working in a department, where the department number starts with "D". By applying the **CHECK OPTION**, we can prevent any non D-department data from being created, using the view:

```
CREATE VIEW   D_EMPLOYEES
    AS SELECT *
         FROM TEMPL
        WHERE WORKDEPT LIKE 'D%'
         WITH CHECK OPTION
```

In the following example we will create a view that can be used as a telephone directory. Since this directory should contain the department name, the view will use data from both the employee table and the department table:

```
CREATE VIEW   DIRECTORY
             (LASTNAME, FIRSTNAME, INIT, PHONE,
              DEPARTMENT, DEPTNAME)
    AS SELECT ALL
              LASTNAME, FIRSTNME, MIDINIT, PHONENO,
              DEPTNO, DEPTNAME
         FROM TEMPL, TDEPT
        WHERE WORKDEPT = DEPTNO
```

Note that column names in a view do not have to be the same as in the underlying tables.

Creating a Synonym

The **CREATE SYNONYM** statement is used to define an alternate name for a table or a view. A common use for this is to allow you to refer to a table or view owned by another user without having to enter the fully-qualified name.

Let's assume that a user has been granted select authority to the department table and to the phone directory view. This user now issues the following two **CREATE SYNONYM** statements:

```
CREATE SYNONYM DTABLE FOR DSN8.TDEPT

CREATE SYNONYM DIR    FOR DSH8.DIRECTORY
```

THE ALTER STATEMENT

The purpose of the **ALTER** statement is to change a definition of a DB2 object. In order to issue the **ALTER** statement, the user must have the proper authority, unless the alteration is done on a table or index created by the user, in which case no special authority is needed. In the following we will assume that the user is altering his own objects.

Altering a Table

The **ALTER TABLE** statement is used to add a new column to an existing base table. You must specify the name of the new column and the data type of the column. The new column will always be the "last" column in the table. When a new column is added, all existing rows in the table will contain null values in the new column.

When an **ALTER TABLE** statement is issued, you may specify the name of a validation procedure. This routine could validate the values in the new column, just before they are inserted. Rows existing before the validation procedure was added are not validated.

We have previously created a table for department D41. This table does not contain the job code column. Since we need information about job codes, we will now add a JOBCODE column to EMPD41. In order to verify the job codes of subsequent inserted rows, we will use a new validation routine called VALIDJC; replacing VALIDD41.

```
ALTER TABLE EMPD41
            ADD JOBCODE DECIMAL(3)
            VALIDPROC VALIDJC
```

Altering an Index

The **ALTER INDEX** statement is used to change the description of an index. The statement can be used to change the buffer pool name, the close dataset option, or the password for the dataset.

In "Creating an Index" on page 27, an index on table EMPD41 was created. The password for the dataset was specified to be DSN8. Let's suppose that the VSAM password has been changed to ABC8. In the following example we will change the password in the index specification:

```
ALTER INDEX XEMPD41
            DSETPASS ABC8
```

THE DROP STATEMENT

The purpose of the **DROP** statement is to delete a DB2 object. All objects that are dependent on the object being dropped are also dropped. In order to issue the DROP statement, the user must have the proper authority. In the following we will assume that SYSADM has granted the user authority to drop tables, indexes, and views. A given user may at any time drop DB2 objects created by that user without any specific granted authority.

The basic form of the **DROP** statement is:

```
DROP
  object  object-name
```

In the following example we will drop the index on department table for department D41. We will then drop the actual table (if we had dropped the table first, the index would be dropped automatically). Next, we will drop the view we created on the employee table, and finally we will drop the synonyms created on the department table and on the phone directory view:

```
DROP INDEX   XEMPD41

DROP TABLE   EMPD41

DROP VIEW    D_EMPLOYEES

DROP SYNONYM DTABLE

DROP SYNONYM DIR
```

THE COMMENT ON STATEMENT

The purpose of the **COMMENT ON** statement is to add an explanatory comment to the description of a table, view, or column in the DB2 catalog tables. The maximum length of a comment is 254 characters. To issue this statement you must be the creator of the table, or you must have proper authority.

The basic form of the **COMMENT ON** statement is:

```
COMMENT ON TABLE  table-name or view-name
        IS   'text within quotes'
```

or

```
COMMENT ON COLUMN  table-name.column-name
        IS   'text within quotes'
```

In the following example we will *add* a description to the department table and to the ADMRDEPT column in the same table:

```
COMMENT ON TABLE TDEPT
        IS 'This table contains all DP departments'

COMMENT ON COLUMN TDEPT.ADMRDEPT
        IS 'Id of the department to which the DEPTNO
            department is administratively reporting'
```

One of the major goals of DB2 is to make it easy for individual users to create and drop tables in the data base. The authorization subsystem of DB2 is designed to support this dynamic data base environment. It permits the individual user, who creates a table, to selectively share the use of this table with other users.

When a new table is created, its creator is automatically given full privileges on the table complete with the **GRANT OPTION** on each privilege. The creator of a table can grant these privileges, or any combination of them, to other users with the **GRANT** statement. When a privilege is granted to another user, the **GRANT OPTION** (ability to make further grants of the privilege) may or may not be included. Once granted, a privilege may be "taken away" by the **REVOKE** statement. If a privilege is revoked from user X, it is automatically revoked from all users to whom user X granted it, unless they have another independent source for the same privilege. For a detailed description of the security and authorization mechanism in DB2, please refer to the IBM DATABASE 2 System Planning and Administration Guide.

THE GRANT STATEMENT

The purpose of the **GRANT** statement is to establish controlled access to DB2 objects. A user issuing a **GRANT** statement can specify the privileges that is granted to the other user(s). As an example, a **GRANT** statement may specify SELECT on a table to user JONES. Another **GRANT** statement can specify UPDATE to user SMITH, even indicating which columns may be updated by SMITH. "Appendix B. Data Control Language Syntax" on page 121 shows the format and the capabilities of the **GRANT** statement.

Let's assume that the creator of table TEMPL would like to share access to the table to user JONES. In this example, JONES is only granted authority to read (SELECT) from the table:

```
GRANT SELECT
   ON TABLE TEMPL
   TO JONES
```

This statement will enable JONES to read all data in the table TEMPL.

The creator of the TEMPL table now decides to issue full authorization to the user SMITH; he will even allow SMITH to grant authorization to other users:

```
GRANT ALL PRIVILEGES
   ON TABLE TEMPL
   TO SMITH
   WITH GRANT OPTION
```

THE REVOKE STATEMENT

The purpose of the **REVOKE** statement is to "take away" authorization that previously has been granted to other user(s). When a user issues a **REVOKE** statement, the level of authorization that should be revoked can be specified. If a user was granted both SELECT and UPDATE authorization on a specific table, the owner of this table can revoke the UPDATE authorization, permitting the user to do only SELECTs on the table. Figure 31 on page 123 shows the format of the **REVOKE** statement.

The creator of the table TEMPL decides that the two granted authorizations made on the table should be revoked. The creator will therefore issue the following **REVOKE** statements:

```
REVOKE SELECT
   ON TABLE TEMPL
   FROM JONES

REVOKE ALL
   ON TABLE TEMPL
   FROM SMITH
```

This part of the guide contains examples of simple and advanced SQL queries. For quick reference, each example has a short heading stating the subject that will be explained. The examples are divided into five chapters:

- **Chapter 5. Data Environment**

  All sample queries are based on six sample tables. In this chapter a sample company is described, and the relationships between the data in the six tables are stated.

- **Chapter 6. Data Manipulation - Simple Queries**

  In this chapter we will exploit the power of the SQL language and relational operations through simple queries all based on one table.

  Each example will consists of a stated problem, the SQL statements to solve the problem, the SQL output, and finally a short explanation.

- **Chapter 7. Data Manipulation - Advanced Queries**

  In this chapter we will use the advanced features of the SQL language. We will also show how data can be added to a table, how existing data can be changed, and how we can delete data.

  Each example will consists of a stated problem, the SQL statements to solve the problem, the SQL output, and finally a short explanation.

- **Chapter 8. Data Control Language - Examples**

  This chapter will show how a table owner can give access authority to other users. The chapter describes the various levels of authority that can be granted. Once given, the authority can be revoked. This will also be illustrated in this chapter.

- **Chapter 9. Data Definition Language - Examples**

  The last chapter in this part contains examples of the data definition language. The chapter contains examples showing how DB2 objects are created, altered, and dropped.

CHAPTER 5. DATA ENVIRONMENT

The following chapters of this guide show queries and the resulting output. All of these queries will be issued against a set of tables that have been developed by Spiffy Computer Services Division.

In order to better understand the data in the tables, let's briefly look at Spiffy Computer Services Division's organization and how projects are controlled in the company.

ORGANIZATION

Spiffy Computer Services Division has, like most corporations, a departmental structure. Each department reports to a higher level department. Usually, each department has a department manager and a number of employees assigned to the department. However, in some cases there will be departments without a manager, e.g. when a manager is moved to another department, and a new manager has not yet been appointed. Similarly, there can be departments without employees.

PROJECTS

All activities in the company belong to projects. At any time Spiffy Computer Services Division has several concurrent active projects, and in order to keep track of these projects a unique identification is assigned to each project (project number). A project may be divided into sub-projects, and a project may even be one of several projects in a larger project. Usually a project is assigned to a department, and likewise an employee is assigned to a project as having project responsibility.

When a project is defined, start and end dates as well as estimated average staffing (manpower) are estimated. If the project consists of several activities, each activity will have an estimated start and end date as well as an estimated manpower consumption. Each of these activities are assigned an identification (activity number). It is possible for the same activity to be associated with the same project more than once, provided that there are unique start dates for each association. In order to make a unique identification of an activity, it is therefore in some cases necessary to use both the activity number and the start date.

Employees are assigned to activities within a project. It should be noted that the assignment is to the activity and not to the specific activity estimate. Information maintained about assignments consists of the starting and ending dates of the assignment as well as the fraction of the employee's time to be spent on this activity between the two dates. The same employee may be assigned to the same activity within the same project more than once. In order to make a unique identification of a sub-activity, the start date must be used.

BASE TABLE STRUCTURE

In order to control the projects and the organization information, Spiffy Computer Services Division has developed a set of DB2 tables. Logically, the tables can be divided into two groups, one holding information about departments and employees (**Organization Application**), and one holding information about projects, activities, and manpower allocation (**Project Application**).

Organization Application Tables

This set of tables consists of two base tables with rows having the following contents:

- Department Table (Table name: DSN8.TDEPT)
  - Department Identification (unique)
  - Department Name
  - Department Manager Id
  - Department Id to Report to
- Employee Table (Table name: DSN8.TEMPL)
  - Employee Serial Number (unique)
  - Employee First Name
  - Employee Middle Initial
  - Employee Last Name
  - Employee Department Id
  - Employee Phone Number
  - Employee Hire Date
  - Employee Job Code
  - Employee Education Level
  - Employee Sex Code
  - Employee Birth Date
  - Employee Salary

"Appendix D. Sample Base Table Definition" on page 127 shows the format of the various fields (or columns) in the two base tables. "Appendix E. Organization Application Base Tables" on page 129 shows a list of the contents of the two tables.

Project Application Tables

This set of tables consists of four base tables with rows having the following contents:

- Project Table (Table name: DSN8.TPROJ)
  - Project Identification (unique)
  - Project Name
  - Department Id Associated With Project
  - Id of Responsible Employee
  - Estimated Staffing Requirement
  - Estimated Project Start Date
  - Estimated Project End Date
  - Id of Major Project (if any)
- Activity Type Table (Table name: DSN8.TACTYPE)
  - Activity Identification (unique)
  - Activity Keyword (unique)
  - Activity Description
- Project/Activity Table (Table name: DSN8.TPROJAC)
  - Project Identification
  - Activity Identification
  - Estimated Staffing Requirement
  - Estimated Activity Start Date
  - Estimated Activity End Date

- Employee/Project/Activity Table (Table name: DSN8.TEMPRAC)
  - Employee Identification
  - Project Identification
  - Activity Identification
  - Fraction of Time Allocated
  - Sub-activity Start Date
  - Sub-activity End Date

"Appendix D. Sample Base Table Definition" on page 127 shows the format of the various fields (or columns) in the four base tables. "Appendix F. Project Application Base Tables" on page 133 shows a list of the contents of the four tables.

DB2 IMPLEMENTATION OF SAMPLE TABLES

Figure 3 on page 43 shows the physical implementation of the sample tables in the DB2 data bases. After the creation of a **storage group** (DSN8G000), the SYSADM has created two **data bases**, one for organization and project application data (DSN8DAPP), and one for programming application data (DSN8DPRG).

In the first mentioned data base two **table spaces** are created, one table space for the department table (DSN8SDEP) and one for the employee table (DSN8SEMP). This data base will also hold all the project related tables, but table spaces for these tables will be created dynamically. In the other data base only one table space (DSN8SCOM) is created. This table space holds all the programming related tables (these tables will not be referenced in this guide). Finally, the figure shows that the organization tables will use **buffer pool** BP1, the project tables will use **buffer pool** BP0, and the programming tables will use **buffer pool** BP2.

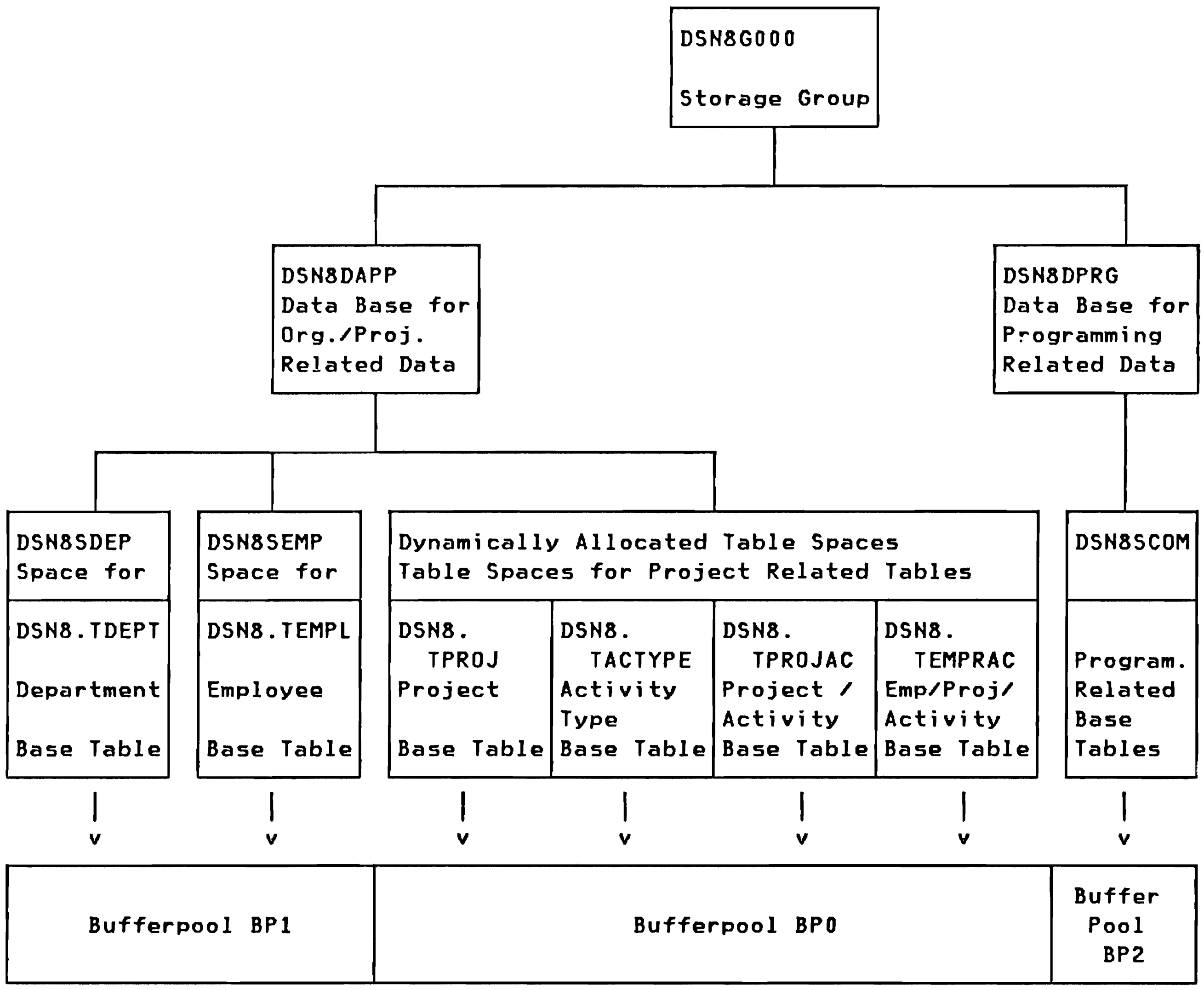

Figure 3. Relationship between the Sample Application DB2 objects

In the preceding chapters we have discussed the format of the SQL statements. In this and the following chapter we will exploit the power of the SQL language and relational operations through various sample queries. This chapter will show simple queries, e.g. the problems can be solved based on data from one table. The following chapter will show queries that make use of the advanced functions of the SQL language.

Each example includes a paragraph stating the problem we are going to solve. The actual SQL statement(s) then follows. The third part of each sample query will show the resulting output. Finally, each example will contain a short explanation of the SQL statement(s) and the result.

OUTPUT FORMAT OF RESULTING TABLES

Column Name: If a column name has been specified in the select list, this name will be printed as the heading in the resulting table. In the following examples, such column headings are printed in capital letters.

If a column is the result of an expression (e.g. SALARY/12), or a built-in function (e.g. AVG(EDUCLVL)), DB2I will return a blank column name. However, in order to explain the result, these blank headings have been substituted with short explanations in the examples. All substituted headings are printed in lower case letters.

Calculated Values: The precision (number of decimal digits) and scale (number of digits to the right of the decimal point) of calculated values are dependent on the attributes of the underlying data and the operation performed. In the following examples we will do addition and subtraction on operands with equal scales. We will see that the result will be presented with the same scale. For a multiplication operation, the precision will be the sum of the precisions of the operands (max. 15). The scale will be the sum of the scales of the operands. The precision of a division is always 15, and the scale will be 15 minus the first operand's precision plus the first operands scale minus the second operand's scale.

The result of the built-in functions will follow the rules stated above. MAX, MIN, and SUM will present the result with the same precision and scale as the underlying data. COUNT will always present the result as an integer value. AVG will present the result with a precision of 15, where the scale will be 15 minus the column's defined precision plus the column's defined scale.

Combined Search Conditions

Simple Select With Combined Conditions

Problem:

Based on the employee table, we want a list of all the female employees, who were hired after January 1, 1980. The list should contain employee number, last and first names, and the date the person was hired.

SQL statement:

```
SELECT EMPNO, LASTNAME, FIRSTNME, HIREDATE
  FROM TEMPL
 WHERE SEX = 'F'
   AND HIREDATE > 800101
```

Result:

EMPNO	LASTNAME	FIRSTNME	HIREDATE
000070	PULASKI	EVA	800930.
000270	PEREZ	MARIA	800930.

Comments:

This simple query uses two combined conditions. Since both conditions must be met, they are combined through the use of the AND keyword. The result shows that both selected employees are female, and both were hired in September 1980.

Simple Select With Ordering

Problem:

Based on the employee table, we want to produce a list of last names, salaries, and department numbers. The list should only include employees earning more than $30,000 a year. The resulting list should be ordered on department number in ascending order (primary), and within each department on salary in descending order (secondary).

SQL statement:

```
SELECT WORKDEPT, LASTNAME, SALARY
  FROM TEMPL
 WHERE SALARY > 30000
 ORDER BY WORKDEPT, SALARY DESC
```

Result:

WORKDEPT	LASTNAME	SALARY
A00	HAAS	52750.00
A00	LUCCHESI	46500.00
B01	THOMPSON	41250.00
C01	KWAN	38250.00
D11	STERN	32250.00
D21	PULASKI	36170.00
E01	GEYER	40175.00

Comments:

As the result shows, all selected employees have salaries exceeding $30,000. The rows are listed in department number order (alphabetic), and, within each department, the rows are ordered on salary, where the employee having the highest salary is listed first.

Simple Select With Grouping

Problem:

Produce a list showing the number of employees working in each department. The list should further contain the calculated average and total salary for each department. The resulting list should be ordered on department number.

SQL statement:

```
SELECT WORKDEPT, COUNT(*), SUM(SALARY), AVG(SALARY)
  FROM TEMPL
 GROUP BY WORKDEPT
 ORDER BY WORKDEPT
```

Result:

WORKDEPT	count(*)	sum(salary)	avg(salary)
A00	3	128500.00	42833.333333333
B01	1	41250.00	41250.000000000
C01	3	90470.00	30156.666666666
D11	9	222100.00	24677.777777777
D21	6	150920.00	25153.333333333
E01	1	40175.00	40175.000000000
E11	5	104990.00	20998.000000000
E21	4	95310.00	23827.500000000

Comments:

The resulting table contains one row for each department. The second column indicates the number of employees in each of these departments. In the third column the sum of salaries within each department is listed. Finally, this sum is divided by the number of employees to express the average salary. Please note that the column names written in lower case, are for information only, DB2 will return "blank" headings.

Simple Select Using Grouping and Having

Problem:

Produce a list showing the average salary for each department. The result should only include employees having a job code less than 55, and all departments with fewer than 3 employees (with jobcode less than 55) should be excluded from the resulting list. Order the result by department number.

SQL statement:

```
SELECT WORKDEPT, AVG(SALARY), COUNT(*)
  FROM TEMPL
 WHERE JOBCODE < 55
 GROUP BY WORKDEPT
HAVING COUNT(*) >= 3
 ORDER BY WORKDEPT
```

Result:

WORKDEPT	avg(salary)	count(*)
D11	21398.000000000	5
D21	19536.666666666	3
E11	18810.000000000	4
E21	23313.333333333	3

Comments:

The result shows a table, where rows have been eliminated in two passes. First all rows having a value of 55 or more in the JOBCODE column are filtered away. On the remaining rows, SQL will do a grouping, and the groups with less than three rows will not be represented in the resulting table.

This logical explanation does not necessarily reflect the actual physical process in SQL.

Simple Select Using Boolean Operators

Problem:

Issue a query that will select employees from the employee table where the following conditions are met:

- The employee must have been hired after August 1, 1974, but before the end of 1979.
- The jobcode must not be 54 or 56.
- The salary must be less than 40,000.
- The education level must be equal to or greater than 12.

For employees meeting these conditions, we will list the last name, the salary, the education level, the job code, and the hire date:

SQL statement:

```
SELECT LASTNAME, SALARY, EDUCLVL, JOBCODE, HIREDATE
  FROM TEMPL
 WHERE (HIREDATE > 740801 AND HIREDATE <= 791231)
   AND (JOBCODE ¬= 54 AND JOBCODE ¬= 56)
   AND SALARY < 40000
   AND EDUCLVL >= 12
```

Result:

LASTNAME	SALARY	EDUCLVL	JOBCODE	HIREDATE
KWAN	38250.00	20	060.	750405.
JONES	18270.00	17	052.	790411.
MARINO	28760.00	17	055.	791205.
JOHNSON	17250.00	16	052.	750911.
LEE	25370.00	14	055.	760223.

Comments:

As can be seen from the resulting table, we have selected five people who all have a hire date later than August 1, 1974, but earlier than January 1, 1980. None of the selected people have a jobcode of 54 or 56. Note that we used the AND keyword, when we specified the two job codes not to be selected, the reason is that we were <u>excluding</u> rows, so both predicates should be true. Finally, the result shows that all salaries are less than 40,000 and all education levels are greater than or equal to 12.

Simple Select Using Expressions

Problem:

The manager of the Personnel department has requested a list of all employees in Spiffy Computer Services Division, who on December 31, 1982 have been working more than 18 years in the company. The list must, apart from the employee numbers and names, include the number of years the selected employees have been working. Further, the list should show the age of the employees when they were hired by Spiffy Computer Services Division. The list should be ordered on the number of years employed:

SQL statement:

```
SELECT EMPNO, LASTNAME, (821231-HIREDATE)/10000,
                     (HIREDATE-BRTHDATE)/10000
  FROM TEMPL
 WHERE (821231 - HIREDATE)/10000 > 18
 ORDER BY 3 DESC
```

Result:

EMPNO	LASTNAME	years employeed	age when hired
000340	GOUNOT	35.072	20.99880000
000050	GEYER	33.041	23.99020000
000110	LUCCHESI	24.071	28.94110000
000120	O'CONNELL	19.002	21.01870000
000310	SETRIGHT	18.031	33.04910000

Comments:

The SELECT statement shows that arithmetic operations can be specified both in the select list and in the WHERE-clause. By subtracting two dates and dividing the result by 10,000, we will obtain the difference expressed in years. In the WHERE-clause the number of years employed is compared to 18, and only rows satisfying this condition are selected. The ordering is done on the third column, and since this column is a result of an expression, we must specify the column number, as opposed to name, in the ORDER BY clause.

Note that the scales in the last two columns are different. The reason is that the third column shows the subtraction between an integer (821231) and a decimal (HIREDATE), compared to the fourth column, where the subtraction is between two decimal numbers. Please refer to "Output Format of Resulting Tables" on page 45.

Simple Select Using Built-In Functions

Problem:

The manager of Administration Systems wants to know to what extent five specific employees are involved in project activities that both start and end during 1982. These five employees have employee numbers 000230 through 000270. To get an indication of this involvment, we will for each employee add the subactivity durations multiplied by the fraction the employee is scheduled to participate in each of the subactivities. This sum is then compared to the total period (i.e. from first subactivity's start date to last subactivity's end date):

SQL statement:

```
SELECT EMPNO, MIN(EMSTDATE), MAX(EMENDATE),
       SUM(EMPTIME * (EMENDATE - EMSTDATE)),
       MAX(EMENDATE) - MIN(EMSTDATE)
  FROM TEMPRAC
 WHERE EMPNO IN ('000230','000240','000250',
                 '000260','000270')
   AND EMSTDATE BETWEEN 820101 AND 821231
   AND EMENDATE BETWEEN 820101 AND 821231
 GROUP BY EMPNO
```

Result:

EMPNO	start	end	calculated alloc.	end-start
000230	820101.	821015.	914.00	914.
000240	820215.	820915.	700.00	700.
000250	820101.	821201.	907.00	1100.
000260	820101.	820701.	593.00	600.
000270	820101.	821015.	914.00	914.

Comments:

As the SQL statement shows, we are only selecting employee numbers between 000230 and 000270. For these employees we are only selecting activities that both start and end during 1982.

The result shows that employee number 000230 is allocated to activities between January 1st until October 15th. By subtracting these two dates, we are assuming that the length of a month is 100 days, the subtraction will therefore result in 914 "days" (last column). You should keep in mind that this artificial length of period is only used to compare the allocation for each employee.

The comparison is done against the fourth column. This column contains the calculated allocation, where the same artificial length of months are used. The fourth column shows that employee number 000230 has a calculated allocation of 914 "days", indicating that in the given period the employee is allocated 100%.

The result further shows that employee number 000250 is allocated 907 "days" in the first 11 months of 1982, equivalent to 1100 "days". This indicates that the average allocation is in the area of 82%.

Use of View

Simple Select Using a View

Problem:

Based on the employee table, the manager of the planning department wants to get a list of all the employees having a jobcode of 52, 54, or 56. Since the planning department is not the owner of the employee table, we will create a view on the employee table under his user-id. This view should only include the employee number, name, work department, and job code.

SQL statements:

```
CREATE VIEW EMPJOBCODES
            (EMP#, NAME, DEPT, JOBCD)
    AS SELECT EMPNO, LASTNAME, WORKDEPT, JOBCODE
         FROM DSN8.TEMPL
        WHERE JOBCODE = 52
           OR JOBCODE = 54
           OR JOBCODE = 56
```

```
SELECT *
  FROM EMPJOBCODES
```

Result:

EMP#	NAME	DEPT	JOBCD
000070	PULASKI	D21	056.
000100	SPENSER	E21	054.
000140	NICHOLLS	C01	056.
000160	PIANKA	D11	054.
000170	YOSHIMURA	D11	054.
000210	JONES	D11	052.
000250	SMITH	D21	052.
000260	JOHNSON	D21	052.
000280	SCHNEIDER	E11	054.
000320	MEHTA	E21	052.
000340	GOUNOT	E21	054.

Comments:

The created view only selects the four specified columns. As the result shows, these columns are given new headings. If a WHERE-clause was specified in the second SELECT statement, the "new" column names defined in the view had to be used. Since the user of the view is using a base table belonging to another user, the FROM clause must specify the fully qualified table name. Since the view only should display rows with specific values in the JOBCODE column, we are using the IN predicate in the WHERE clause.

The SELECT statement following the CREATE statement specifies that all (four) columns should be selected, using the short hand notation "*". Since we are using a view, the FROM clause contains a <u>view name</u> and not a table name.

Simple Select Using LIKE

Problem:

Based on the project table, a list should be produced showing all the projects that have the text string "PROGRAM" somewhere in the project name. In order to further limit the list to relevant projects, only project numbers starting with "AD" or project numbers having "21" as third and fourth character should be selected. List project number, project name, and name of major project, ordered by project number.

SQL statement:

```
SELECT PROJNO, PROJNAME, MAJPROJ
  FROM TPROJ
 WHERE PROJNAME LIKE '%PROGRAM%'
   AND (PROJNO LIKE 'AD%' OR PROJNO LIKE '__21%')
 ORDER BY PROJNO
```

Result:

PROJNO	PROJNAME	MAJPROJ
AD3111	PAYROLL PROGRAMMING	AD3110
AD3112	PERSONNEL PROGRAMMG	AD3110
AD3113	ACCOUNT.PROGRAMMING	AD3110
MA2110	W L PROGRAMMING	MA2100
MA2111	W L PROGRAM DESIGN	MA2110

Comments:

The first LIKE predicate in the WHERE-clause will select all the projects having the character string "PROGRAM" somewhere in the project name, since the pattern is preceded and ended with a "%" character. In the second LIKE predicate we are looking at the project number. We are here selecting all projects having a number starting with the characters "AD" or having the "21" characters as third and fourth characters. Since the "_" character represents exactly one character, we can position the pattern.

CHAPTER 7. DATA MANIPULATION - ADVANCED QUERIES

The SQL language provides several features which enable complex data base queries. Such queries will typically address several tables, and possibly several imbedded selects in one SQL statement. These features, which will be shown in this chapter, may be used in combination with each other, but may also be used with the simpler language features described in the previous chapter.

When a complex - or advanced - SQL query is presented to DB2, DB2 automatically generates an algorithm for processing the query, taking into account the various indexes which are available and the physical clustering of data in the data base. Since this process is completely transparent to the SQL user, it will not be discussed in this guide. Even if we are dealing with advanced queries, the SQL user only has to think about **what** to obtain as a result, **not how** DB2 should produce the result.

Subselect on Same Table

Query With Subselect on Same Table

Problem:

Issue a query that will list all female employees, who earn more than the average salary in Spiffy Computer Services Division (both male and female employees). The resulting table must be ordered on the last name in ascending order. We also want to see the average salary for all employees:

SQL statements:

```
SELECT EMPNO, FIRSTNME, LASTNAME, SALARY
  FROM TEMPL
 WHERE SEX = 'F'
   AND SALARY > ( SELECT AVG(SALARY)
                    FROM TEMPL )
 ORDER BY 3
```

```
SELECT AVG(SALARY) FROM TEMPL
```

Result:

EMPNO	FIRSTNME	LASTNAME	SALARY
000010	CHRISTINE	HAAS	52750.00
000090	EILEEN	HENDERSON	29750.00
000030	SALLY	KWAN	38250.00
000220	JENNIFER	LUTZ	29840.00
000140	HEATHER	NICHOLLS	28420.00
000270	MARIA	PEREZ	27380.00
000070	EVA	PULASKI	36170.00

avg(salary)
27303.593750000

Comments:

SQL will first do the subselect in order to calculate the average salary for <u>all</u> employees. The result is then substituted in the WHERE-clause of the outer-level query, and this query is then executed. The ORDER BY clause specifies that the resulting table will be ordered on the third column, which is the LASTNAME column. We could instead have specified ORDER BY LASTNAME.

The second SELECT statement shows that SQL is a true free form language, the FROM-clause can be specified immediately after the selected list on the same line.

The result shows that seven female employees earn more than the the average salary.

Since the inner-level SELECT statement only returned <u>one</u> value, the SELECT statement is specified directly in the WHERE-clause of the outer-level query. The example "Subselect Returning Set of Values" on page 72 shows how to deal with inner-level SELECTs that return <u>more</u> than one value.

Selecting From Two Tables

Problem:

The secretary in the Support Services department wants a phone directory for all the employees working in departments E01, E11, and E21. This directory must list the last and first names, the names of the departments, and the phone numbers. The resulting list should be ordered on last name and first name:

SQL statement:

```
SELECT LASTNAME, FIRSTNME, DEPTNAME, PHONENO
  FROM TEMPL, TDEPT
 WHERE WORKDEPT = DEPTNO
   AND WORKDEPT IN ('E01', 'E11', 'E21')
 ORDER BY LASTNAME, FIRSTNME
```

Result:

LASTNAME	FIRSTNME	DEPTNAME	PHONENO
GEYER	JOHN	SUPPORT SERVICES	6789
GOUNOT	JASON	SOFTWARE SUPPORT	5698
HENDERSON	EILEEN	OPERATIONS	5498
LEE	WING	SOFTWARE SUPPORT	2103
MEHTA	RAMLAL	SOFTWARE SUPPORT	9990
PARKER	JOHN	OPERATIONS	4502
SCHNEIDER	ETHEL	OPERATIONS	8997
SETRIGHT	MAUDE	OPERATIONS	3332
SMITH	PHILIP	OPERATIONS	2095
SPENSER	THEODORE	SOFTWARE SUPPORT	0972

Comments:

The names and phone numbers can be derived from the employee table (TEMPL). The department name, however, is not in that table and has to be derived from the department table (TDEPT). The FROM-clause specifies both tables. The join between the two tables is done on the department number (WORKDEPT = DEPTNO), since this column exists in both tables (different names, but same data). Conceptually, when a row from the employee table has been selected satisfying the IN predicate in the WHERE-clause, SQL will use the value in the WORKDEPT column to find a matching value in the DEPTNO column in the department table. When the row is found, the department name is taken from the DEPTNAME column in that row.

Comparing Two Rows In the Same Table

Problem:

Compare pairs of employees having the same job code. If one of the employees in a pair was hired more than 9 years before the other, but has a lower salary than the other, the names and salaries of the two employees plus the difference in hire dates (in terms of years) must be shown. List the result in descending order on the difference in hire dates:

SQL statement:

```
SELECT EMPOLD.LASTNAME, EMPOLD.SALARY,
       EMPNEW.LASTNAME, EMPNEW.SALARY,
      (EMPNEW.HIREDATE-EMPOLD.HIREDATE)/10000
  FROM TEMPL EMPOLD, TEMPL EMPNEW
 WHERE EMPOLD.JOBCODE  = EMPNEW.JOBCODE
   AND EMPNEW.HIREDATE - EMPOLD.HIREDATE > 90000
   AND EMPOLD.SALARY   < EMPNEW.SALARY
 ORDER BY 5 DESC
```

Result:

LASTNAME	SALARY	LASTNAME	SALARY	years diff.
GOUNOT	23840.00	SPENSER	26150.00	33.01140000
GOUNOT	23840.00	YOSHIMURA	24680.00	31.04100000
GOUNOT	23840.00	SCHNEIDER	26250.00	19.98190000
BROWN	27740.00	MARINO	28760.00	13.09020000
QUINTANA	23800.00	PEREZ	27380.00	9.02020000

<———— EMPOLD ————> <———— EMPNEW ————>

Comments:

This query shows how two rows in the same table can be compared by joining copies of the same table. The two "virtual" tables have given the arbitrary labels EMPOLD and EMPNEW in order to distinguish between them. These qualifying labels must be used when the columns are referenced in order to make the column names unique.

In concept, SQL is selecting one row in the EMPOLD table. This row is then compared to all the rows in the EMPNEW table. SQL will then select the next row in the EMPOLD table and compare that row to the rows in the EMPNEW table, and so on.

Queries Combined With UNION

Problem:

Produce a list containing department numbers, department names, and manager names. If a department is without a manager the character string "** UNKNOWN **" should be written in the manager name column. Present the result in department number order:

SQL statement:

```
SELECT DEPTNO, DEPTNAME, LASTNAME
  FROM TDEPT, TEMPL
 WHERE MGRNO = EMPNO
UNION
SELECT DEPTNO, DEPTNAME, '** UNKNOWN **  '
  FROM TDEPT
 WHERE MGRNO NOT IN ( SELECT EMPNO
                        FROM TEMPL )
 ORDER BY 1
```

Result:

DEPTNO	DEPTNAME	LASTNAME
A00	SPIFFY COMPUTER SERVICE DIV.	HAAS
B01	PLANNING	THOMPSON
C01	INFORMATION CENTER	KWAN
D01	DEVELOPMENT CENTER	** UNKNOWN **
D11	MANUFACTURING SYSTEMS	STERN
D21	ADMINISTRATION SYSTEMS	PULASKI
E01	SUPPORT SERVICES	GEYER
E11	OPERATIONS	HENDERSON
E21	SOFTWARE SUPPORT	SPENSER

Comments:

The UNION operator is used to combine the result of the two SELECT statements. The first SELECT statement will find all the departments in the department table, where the manager number also can be found in the employee table.

The second SELECT statement will find the departments in the department table that have manager numbers not contained in the employee table. Consequently, we can not provide a name for these managers, and instead we will use a constant indicating that the name is unknown. Since the data types of corresponding items selected by the two SELECT statements must be identical, the literal has been specified as exactly 15 characters matching the definition of LASTNAME (VARCHAR(15)).

Note that the ORDER BY clause <u>must</u> be written after the last SELECT statement in the union, and that only column numbers can be specified in the clause.

Query Testing For Existence

Problem:

Find all activity types that have been defined in the Project-Activity base table. Then list each activity type having a defined staffing estimate of more than 1. The resulting table should be ordered by activity number:

SQL statement:

```
SELECT ACTNO, ACTDESC
  FROM TACTYPE TAC
 WHERE EXISTS ( SELECT *
                  FROM TPROJAC
                 WHERE ACSTAFF > 1
                   AND ACTNO = TAC.ACTNO )
 ORDER BY 1
```

Result:

ACTNO	ACTDESC
60	DESCRIBE LOGIC
70	CODE PROGRAMS
80	TEST PROGRAMS
130	OPER COMPUTER SYS

Comments:

This correlated query first selects a row in the TACTYPE table. The activity number from the selected row is then substituted into the inner-level query's WHERE-clause (TAC.ACTNO). If the inner-level query finds at least one row in the TPROJAC table, a true condition is set, which means that the EXISTS predicate in the outer-level query is satisfied.

Instead of using the EXISTS predicate, we could have solved the stated problem with the following query by joining two tables:

```
SELECT DISTINCT TACTYPE.ACTNO, ACTDESC
  FROM TACTYPE, TPROJAC
 WHERE ACSTAFF > 1
   AND TACTYPE.ACTNO = TPROJAC.ACTNO
 ORDER BY 1
```

Query Testing For Non-Existence

Problem:

Before a new project can be committed, the manager of the Planning department wants to have a list of all the employees, who are not currently assigned to any activity:

SQL statement:

```
SELECT EMPNO, LASTNAME, FIRSTNME
  FROM TEMPL EMP
 WHERE NOT EXISTS (SELECT *
                     FROM TEMPRAC
                    WHERE EMPNO = EMP.EMPNO)
 ORDER BY 1
```

Result:

EMPNO	LASTNAME	FIRSTNME
000060	STERN	IRVING
000120	O'CONNELL	SEAN

Comments:

In the outer-level query, SQL will retrieve an employee number from the employee table (TEMPL). This employee number is then substituted into the WHERE-clause of the inner-level query (correlated sub-query). If the inner-level query finds a row in the Employee Project Activity base table, the given employee is assigned to an activity, and should not be included in the the resulting table. Since we are using the NOT EXISTS predicate, only employee numbers not found in the correlated sub-query will satisfy the WHERE-clause of the outer-level query.

Table Joined to Itself With Subselect

Problem:

Three people in Spiffy Computer Services Division have done an outstanding job. The management has decided to give these three employees a salary raise of 50%. After the salaries have been updated in the employee table, we will select all the employees (if any), who have a salary exceeding their respective manager's salary:

SQL statements:

```
UPDATE TEMPL
   SET SALARY = 1.5*SALARY
 WHERE EMPNO = '000200'
    OR EMPNO = '000240'
    OR EMPNO = '000320'
```

```
SELECT E.EMPNO, E.LASTNAME, E.SALARY,
       M.LASTNAME, M.SALARY
  FROM TEMPL E, TEMPL M
 WHERE E.SALARY > M.SALARY
   AND E.WORKDEPT = M.WORKDEPT
   AND M.EMPNO IN
         ( SELECT MGRNO
             FROM TDEPT )
```

Result:

EMPNO	LASTNAME	SALARY	LASTNAME	SALARY
000200	BROWN	41610.00	STERN	32250.00
000240	MARINO	43140.00	PULASKI	36170.00
000320	MEHTA	29925.00	SPENSER	26150.00

<——— Employee Data ———> <——— Manager Data ———>

Comments:

The salaries are updated using the UPDATE statement. For the three specified employee numbers, SQL will multiply the current salary fields with 1.5, thus increasing the salaries by 50%.

The SELECT statement joins the employee table to itself. The reason is that we are comparing the salaries for two employees working in the same department. The two "logical" tables are assigned the identification labels E (employee) and M (manager). Since the employees in the "M" table must be managers, we are using a sub-select in the outer-level query. This sub-select will retrieve all manager numbers from the department table and through the IN predicate eliminate all non-managers in the "M" table.

The result shows that only these three people to whom we just gave the salary raise earn more than their managers.

This query assumes that the MGRNO (manager number) for a DEPTNO (department number) in the TDEPT table agrees with the WORKDEPT (work department number) for his/her EMPNO (employee number) in the TEMPL table. It is also limited to non-manager employees and their managers, and does not consider higher level managers and their salaries.

Correlated Subselect

Problem:

This problem is the same as outlined in "Table Joined to Itself With Subselect" on page 66, but we will now use a correlated subselect to select all the employees (if any), who have a salary exceeding their respective manager's salary:

SQL statement:

```
SELECT EMPNO, LASTNAME, SALARY, M.MGRNO
  FROM TEMPL E, TDEPT M
 WHERE WORKDEPT = DEPTNO
   AND SALARY > (SELECT SALARY
                   FROM TEMPL
                  WHERE EMPNO = M.MGRNO)
```

Result:

EMPNO	LASTNAME	SALARY	MGRNO
000200	BROWN	41610.00	000060
000240	MARINO	43140.00	000070
000320	MEHTA	29925.00	000100

Comments:

The salaries have been updated using the UPDATE statement from the previous example.

The outer-level query will select an employee from the employee table and that employee's manager from the department table. When the work department number is equal to the department number in the department table, the correlated subselect will be executed to verify if the salary for the selected employee is greater than the salary found in the subselect, where the employee number is equal to the manager number in the given department. As written, this query cannot list the names of the managers, since we are not joining the employee table to itself, and consequently only the employee information can be derived from the employee table. The manager number is derived from the department table.

As expected, the result is the same as in the previous example.

Alternative SELECT Statement:

The following query will show a third way to solve the given problem. This query uses nested subselects in two levels without combined predicates in the WHERE-clauses. Because of the construction of the query, only data from the employee table can be listed in the output (i.e. the manager number will not be listed):

SQL statement:

```
SELECT EMPNO, LASTNAME, SALARY
  FROM TEMPL E
 WHERE E.SALARY >
       (SELECT SALARY
          FROM TEMPL M
         WHERE M.EMPNO =
               (SELECT MGRNO
                  FROM TDEPT
                 WHERE DEPTNO = E.WORKDEPT) )
```

Result:

EMPNO	LASTNAME	SALARY
000200	BROWN	41610.00
000240	MARINO	43140.00
000320	MEHTA	29925.00

Combining Views and Unions

Problem:

The manager of Spiffy Computer Services Division has requested a list of all the departments. This list must contain department numbers, department names, manager numbers, and manager names. Since this list will be requested frequently, it has been decided that a view producing this list should be created. The name of the view will be VDEPTMGR.

Using this view, we will now produce a list containing department numbers, manager numbers, and manager names. The list should be a combination of the following three situations:

1. All departments having a manager, where the manager also exists in the employee table.
2. All departments not having a manager assigned yet. For these departments, the manager name columns must show that no manager has been assigned yet.
3. All departments having a manager, but where the manager does not exist in the employee table.

The combined list should be ordered by department number:

SQL statements:

```
CREATE VIEW VDEPTMGR
           ( DEPTNO, DEPTNAME, MGRNO, FIRSTNME,
             MIDINIT, LASTNAME )
    AS SELECT DEPTNO, DEPTNAME, EMPNO, FIRSTNME,
             MIDINIT, LASTNAME
         FROM TDEPT, TEMPL
        WHERE MGRNO = EMPNO
```

```
SELECT DEPTNO, MGRNO, M.FIRSTNME, M.LASTNAME
  FROM VDEPTMGR M, TEMPL E
 WHERE E.WORKDEPT = DEPTNO
UNION
SELECT DEPTNO, MGRNO, '* NO MANAGER', 'ASSIGNED YET * '
  FROM TDEPT
 WHERE MGRNO = '      '
UNION
SELECT DEPTNO, MGRNO, '* INVALID   ', 'MANAGER NO.  * '
  FROM TDEPT M
 WHERE MGRNO ¬= '      '
   AND NOT EXISTS ( SELECT EMPNO FROM TEMPL
                    WHERE M.DEPTNO = WORKDEPT )
 ORDER BY 1 ASC
```

Result:

DEPTNO	MGRNO	FIRSTNME	LASTNAME
A00	000010	CHRISTINE	HAAS
B01	000020	MICHAEL	THOMPSON
C01	000030	SALLY	KWAN
D01		* NO MANAGER	ASSIGNED YET *
D11	000060	IRVING	STERN
D21	000070	EVA	PULASKI
E01	000050	JOHN	GEYER
E11	000090	EILEEN	HENDERSON
E21	000100	THEODORE	SPENSER

Comments:

The VDEPTMGR view joins the department table (TDEPT) and the employee table (TEMPL) on the column names MGRNO and EMPNO. The result is a department table containing department information and manager information.

The first SELECT statement will produce one row for each department having a manager, where the manager also exists in the employee table.

The second SELECT statement will produce a row for each department not having a manager assigned. Instead of name values we are here specifying constants. Note that the first constant must have a length of 12 characters to match FIRSTNME, the second constant must be 15 characters to match LASTNAME.

The third SELECT statement will produce a row for each department having a manager, but where the manager does not exist in the employee table. The NOT EXISTS predicate in the WHERE-clause will only select those department numbers (DEPTNO) in the department table, where there is no match on the WORKDEPT column in the employee table. Note that the use of the correlation variable M in the sub-query limits the search to the department numbers (DEPTNO) being considered in the outer-level SELECT. Without the use of this correlation variable, the NOT EXISTS predicate would always return a "false" indicator and consequently no rows would qualify. The department table does not contain a row satisfying this select statement.

Finally, the results of the three SELECT statements are combined, using UNION, and the combined list is ordered in ascending order on department number (column number one).

Subselect Returning Set of Values

Problem:

The Personnel department is currently doing an analysis on employee data. For this purpose, we will select all employees, who have been working longer in the company than all the members of department C01, and who are younger than any employee having a job code greater than or equal to 44 and less than or equal to 50. The result should include the employee number, name, work department number, and department name:

SQL statement:

```
SELECT EMPNO, LASTNAME, WORKDEPT, DEPTNAME
  FROM TEMPL, TDEPT
 WHERE HIREDATE < ALL (SELECT HIREDATE
                         FROM TEMPL
                        WHERE WORKDEPT = 'C01')
   AND BRTHDATE > ANY (SELECT BRTHDATE
                         FROM TEMPL
                        WHERE JOBCODE BETWEEN 44 AND 50)
   AND WORKDEPT = DEPTNO
 ORDER BY WORKDEPT, LASTNAME
```

Result:

EMPNO	LASTNAME	WORKDEPT	DEPTNAME
000120	O'CONNELL	A00	SPIFFY COMPUTER SERVICE DIV.
000200	BROWN	D11	MANUFACTURING SYSTEMS
000220	LUTZ	D11	MANUFACTURING SYSTEMS
000250	SMITH	D21	ADMINISTRATION SYSTEMS
000090	HENDERSON	E11	OPERATIONS

Comments:

The two subselects could both return a set of values (more than one row). The two sets of values are substituted into the WHERE-clause. The first part of the WHERE-clause in the outer-level query is only satisfied, if the hire date of the selected row is less than all the hire dates in the subset. Similarly, the second part of the WHERE-clause is only satisfied, if the birth date value of the selected row is greater than any of the values in the subset of birth dates. The third part of the WHERE-clause is used to retrieve the name of the department from the department table using a join.

INSERT of a Single Row

Problem:

Department D21 will be involved in new projects and has therefore hired two new employees. These two new employees have not yet started working, so phone numbers and job codes are not known. We will now insert the available data for the two employees, leaving the unknown data as null values. After the INSERT operation, we will list all employees in the D21 employee table, where the job code has a null value:

SQL statements:

```
INSERT INTO TEMPLD21
              (EMP#, LASTNAME, FIRSTNAME, SALARY)
       VALUES ('000272','PETERSEN','LARRY',18385)

INSERT INTO TEMPLD21
       VALUES ('000274','JOHNSON','ELISABETH',
                 NULL,NULL,18740)
```

```
SELECT *
  FROM TEMPLD21
 WHERE JOBCD IS NULL
```

Result:

EMP#	LASTNAME	FIRSTNAME	PHONE	JOBCD	SALARY
000272	PETERSEN	LARRY	-	-	18385.00
000274	JOHNSON	ELISABETH	-	-	18740.00

Comments:

The first INSERT statement uses a list of column names. Column names not included in this list will receive a null value, when the row is inserted. Note that the values list only has data items for the specified column names. The second INSERT statement has no list of column names. In this case, we must supply a data item for each column in the defined table. Since we do not know the phone number and job code, we have specified the data items as NULLs.

The SELECT statement will only list employees where the job code is a null value. As the resulting table shows, null values are presented as hyphens ("-").

INSERT of Rows From Existing Table

Problem:

The manager of department D21 has decided to have a special employee table created. This table should only contain data for people working in department D21. Apart from employee numbers, first and last names, the table should contain phone numbers, jobcodes, and salaries. As first step, the system administrator is requested to create a table named TEMPLD21. As second step all employees working in D21 - except the manager - should be inserted into the new table. Finally, we will produce a department telephone directory showing last name, first name, employee number, and telephone number:

SQL statements:

```
CREATE TABLE DSN8.TEMPLD21
                 (EMP#      CHAR(6),
                  LASTNAME  VARCHAR(15),
                  FIRSTNAME VARCHAR(12),
                  PHONE     CHAR(4),
                  JOBCD     DECIMAL(3),
                  SALARY    DECIMAL(8,2) )
```

```
INSERT INTO TEMPLD21
            SELECT EMPNO, LASTNAME, FIRSTNME,
                   PHONENO, JOBCODE, SALARY
              FROM TEMPL
             WHERE WORKDEPT = 'D21'
               AND EMPNO NOT IN (SELECT MGRNO
                                   FROM TDEPT
                                  WHERE DEPTNO = 'D21')
```

```
SELECT LASTNAME, FIRSTNAME, EMP#, PHONE
  FROM TEMPLD21
 ORDER BY 1, 2
```

Result:

LASTNAME	FIRSTNAME	EMP#	PHONE
JEFFERSON	JAMES	000230	2094
JOHNSON	SYBIL	000260	8953
MARINO	SALVATORE	000240	3780
PEREZ	MARIA	000270	9001
SMITH	DANIEL	000250	0961

Comments:

The first SQL statement will create a table, where the columns will have the specified attributes (for a discussion of data types, please refer to "Creating a New Table" on page 26).

The second SQL statement is an INSERT statement with an imbedded SELECT statement containing a subselect. The subselect will select the manager's employee number. This number is used to prevent the outer-level select from including the manager's data. The select list of the outer-level query only contains the columns that should be inserted into the new D21 employee table. Note that these names are the column names of source table (TEMPL).

The third SQL statement is a simple SELECT on the TEMPLD21 table producing a telephone directory for department D21.

UPDATE of a Single Row

Problem:

One of the newly hired employees - Larry Petersen - has now started working in department D21. He has been assigned a telephone number and a job code, and we will therefore update this data in the D21 employee table. After the update, we will produce a listing of all the D21 employees having valid job codes (i.e. not null values):

SQL statements:

```
UPDATE TEMPLD21
   SET PHONE = '4176',
       JOBCD = 52
 WHERE EMP# = '000272'
```

```
SELECT *
  FROM TEMPLD21
 WHERE JOBCD IS NOT NULL
 ORDER BY EMP#
```

Result:

EMP#	LASTNAME	FIRSTNAME	PHONE	JOBCD	SALARY
000230	JEFFERSON	JAMES	2094	053.	22180.00
000240	MARINO	SALVATORE	3780	055.	28760.00
000250	SMITH	DANIEL	0961	052.	19180.00
000260	JOHNSON	SYBIL	8953	052.	17250.00
000270	PEREZ	MARIA	9001	055.	27380.00
000272	PETERSEN	LARRY	4176	052.	18385.00

Comments:

The UPDATE statement selects one row via the WHERE-clause. For this row, the column values of PHONE and JOBCD will be set to the specified values regardless of the previous contents.

UPDATE of Multiple Rows

Problem:

The manager of department D21 has decided to give all employees with a job code of 55 a salary increase of 9%. A new job code system has been adopted by Spiffy Computer Services Division. This system is based on a three digit number, where the first digit is a "2", and the last two digits are the previous job code (e.g. 55). All jobcodes should be updated from 0xx to 2xx. After the updating has been done, the manager wants a list showing the names, job codes, and salaries for the employees having a job code of 255 (previously 55):

SQL statements:

```
UPDATE TEMPLD21
   SET SALARY = SALARY + 0.09 * SALARY
 WHERE JOBCD = 55

UPDATE TEMPLD21
   SET JOBCD = JOBCD + 200
```

```
SELECT LASTNAME, FIRSTNAME, JOBCD, SALARY
  FROM TEMPLD21
 WHERE JOBCD = 255
```

Result:

LASTNAME	FIRSTNAME	JOBCD	SALARY
PEREZ	MARIA	255.	29844.20
MARINO	SALVATORE	255.	31348.40

Comments:

The first UPDATE statement will only select the rows where the job code is 55. For these rows the SALARY values will be reset to the computed value (i.e. 9% is added to the "old" salary). All other rows will not be affected. The second UPDATE statement does not have a WHERE-clause, and consequently all rows in the table will be updated. The result of the SELECT statement shows that two employees have a jobcode of 255 (previously 55) and that these two employees now have a salary that is 9% higher than shown in the previous example.

DELETE of a Single Row

Problem:

Before the other newly hired employee - Elisabeth Johnson - was scheduled to start in department D21, she decided to accept a job in another company. We will therefore delete her data from the D21 employee table. After the delete has taken place, we will produce a new listing of all the employees in department D21, this list should be ordered on employee number:

SQL statements:

```
DELETE
   FROM TEMPLD21
 WHERE LASTNAME = 'JOHNSON'
   AND FIRSTNAME= 'ELISABETH'
```

```
SELECT *
  FROM TEMPLD21
 ORDER BY 1
```

Result:

EMP#	LASTNAME	FIRSTNAME	PHONE	JOBCD	SALARY
000230	JEFFERSON	JAMES	2094	253.	22180.00
000240	MARINO	SALVATORE	3780	255.	31348.40
000250	SMITH	DANIEL	0961	252.	19180.00
000260	JOHNSON	SYBIL	8953	252.	17250.00
000270	PEREZ	MARIA	9001	255.	29844.20
000272	PETERSEN	LARRY	4176	252.	18385.00

Comments:

The DELETE statement will select the row where the specified first and last names match. When the row is found, it will be deleted from the table. Note that a DELETE statement without a WHERE-clause will delete all rows in the table.

The following SELECT statement lists all rows in the D21 employee table (no WHERE-clause). As the result shows, the employee number 000274 is no longer in the table.

DELETE of Multiple Rows

Problem:

A new education program has been started in Spiffy Computer Services Division. In the first phase of this program, all employees with an education level of 15 are enrolled in a training program lasting seven months. During this period the selected employees will report to the education department. We will now update the employee table for department D21, and delete employees with an education level of 15. After the delete has taken place, we will print the names and employee numbers of those remaining in the table:

SQL statements:

```
DELETE
   FROM TEMPLD21
 WHERE EMP# IN ( SELECT EMPNO
                   FROM TEMPL
                  WHERE WORKDEPT = 'D21'
                    AND EDUCLVL  =  15   )
```

```
SELECT EMP#, LASTNAME, FIRSTNAME
  FROM TEMPLD21
 ORDER BY EMP#
```

Result:

EMP#	LASTNAME	FIRSTNAME
000230	JEFFERSON	JAMES
000240	MARINO	SALVATORE
000260	JOHNSON	SYBIL
000272	PETERSEN	LARRY

Comments:

The D21 employee table does not hold any information on education levels. We have therefore issued a sub-query on the TEMPL table, selecting all employee numbers from department D21, where education level is 15. This list of numbers is substituted into the WHERE-clause of the DELETE statement, and the appropriate rows are deleteed from the D21 table. As the result of the SELECT shows, two rows were deleted from the table.

In the preceding chapters, we have discussed how a user can define DB2 objects using the SQL Data Definition Language, and how data can be inserted and manipulated in these objects through the use of the SQL Data Manipulation Language. In this chapter we will discuss the security aspects of DB2 objects. We will, through examples, show how users are authorized to access and modify data in tables, how the user can give authorization to other users, so tables can be shared between users, and how only partial authorization can be issued. We will further show how authorization, once given, can be "taken away" again.

For a complete description of the Data Control Language, please refer to the IBM DATABASE 2 Reference and the IBM DATABASE 2 System Planning and Administration Guide.

SQL USERS IN THE SAMPLE COMPANY

The storage groups, data bases, and table spaces for the Organization Application Tables and the Project Application Tables have been created by the system administrator. In the following examples we will assume that the user-id of the system administrator is SYSADM. This user-id has full authorization to all DB2 objects. We will further assume that several employees in Spiffy Computer Services Division are using SQL on a regular basis. In each of the four departments (D11, D21, E11, and E21) a user responsible for SQL has been appointed. Finally, two employees in department D21 have each been assigned a SQL user-id. Following is a list of valid user-id's:

- SYSADM - system administrator
- USERD11 - user responsible for SQL in department D11
- USERD21 - user responsible for SQL in department D21
  - USERD21A - Daniel Smith
  - USERD21B - Maria Perez
- USERE11 - user responsible for SQL in department E11
- USERE21 - user responsible for SQL in department E21

Grant SELECT Authority On a Table

After loading the Project Application Tables, the system administrator is going to **GRANT** SELECT authority to USERD11, USERD21, USERE11, and USERE21. All four users are allowed to do SELECTs on all rows and all columns, but they are not allowed to grant authority to other users. The system administrator will therefore issue the following GRANT statement:

```
GRANT SELECT
   ON TABLE DSN8.TPROJ,
            DSN8.TPROJAC,
            DSN8.TEMPRAC,
            DSN8.TACTYPE
   TO USERD11, USERD21, USERE11, USERE21
```

The example shows that several tables (and/or views) can be specified in one GRANT statement. In the example we are assuming that the tables were loaded by user DSN8, we must therefore use the qualified table name. The example also shows that the same authorization can be granted to several users in one statement.

Grant SELECT Authority To PUBLIC

The department table does not contain any sensitive information and should be available to all SQL users in Spiffy Computer Services Division. The following GRANT statement will therefore be issued:

```
GRANT SELECT
   ON TABLE DSN8.TDEPT
   TO PUBLIC
```

Instead of user-id(s), PUBLIC can be specified. This indicates that we are granting the specified privilege to <u>all</u> user-ids.

Grant UPDATE Authority On a Table

It has been decided that the Project base table will be maintained by the SQL-responsible employee in department D21 (USERD21). The system administrator will therefore issue a GRANT statement giving USERD21 full update capability on the Project table:

```
GRANT UPDATE
   ON TABLE DSN8.TPROJ
   TO USERD21
```

This statement will enable the user to update rows, but does not permit the user to delete or insert rows.

Note that before any updates can take place, SELECT authority on the table must also be established. This is assumed to be done in all the following examples.

Grant UPDATE Authority On Columns

The SQL-responsible users in department D11 and E11 want to be able to update the project start and end dates in the project-activity table (DSN8.TPROJAC). The system administrator will therefore issue an update authority on the ACSTDATE and ACENDATE columns for USERD11 and USERD21:

```
    GRANT UPDATE(ACSTDATE, ACENDATE)
       ON TABLE DSN8.TPROJAC
       TO USERD11, USERE11
```

The GRANT statement shows that the two specified users now can update the two mentioned columns, but not any other columns.

Grant INSERT and DELETE Authority On a Table

In "Grant UPDATE Authority On a Table" on page 82 the employee with user-id USERD21 was granted update authority on the Project table. This employee now has a need for inserting new rows and deleting some existing rows. It has therefore been decided that USERD21 also must have INSERT and DELETE authority:

```
    GRANT INSERT, DELETE
       ON TABLE DSN8.TPROJ
       TO USERD21
```

This example shows that several privileges can be specified in one GRANT statement.

Grant INDEX and ALTER Authority On a Table

In order to be able to maintain the Project table, USERD21 must be able to create indexes on the table and to add columns to the table (use of ALTER). The system administrator will therefore issue a GRANT statement that will provide the specified authority:

```
GRANT INDEX, ALTER
   ON TABLE DSN8.TPROJ
   TO USERD21
```

After this GRANT statement has been issued, USERD21 can create indexes and issue ALTER statements on the Project table (TPROJ).

Grant Authority With GRANT Option

The SQL-responsible person in department D21 (USERD21) wants to be able to delegate some of the maintenance work on the Project table to Daniel Smith (user-id: USERD21A). The system administrator will therefore reissue a GRANT statement with UPDATE privilege to USERD21, but this time with the **GRANT OPTION**:

```
GRANT UPDATE
   ON TABLE DSN8.TPROJ
   TO USERD21
 WITH GRANT OPTION
```

The GRANT OPTION indicates that USERD21 now has the authority to pass the UPDATE privilege on the Project table along to other users, in this case to USERD21A.

Note that a statement that grants TO PUBLIC WITH GRANT OPTION is not allowed in SQL. If such a statement is encountered by the system, a warming message is issued and the GRANT is made <u>without</u> GRANT OPTION.

USERD21 will now allow Daniel Smith (USERD21A) to make updates on the PRSTAFF, PRSTDATE, and PRENDATE columns of the Project table. We will assume that USERD21A and USERD21B already are granted SELECT authority on the Project table. USERD21 will therefore issue the following GRANT statement:

```
GRANT UPDATE(PRSTAFF, PRSTDATE, PRENDATE)
   ON TABLE DSN8.TPROJ
   TO USERD21A
```

Note that this statement is issued by USERD21, and not by the system administrator. Daniel Smith can only update the three specified columns, and is not allowed to insert or delete rows.

Let's assume that the last GRANT statement was also issued with the GRANT OPTION. In that case, Daniel Smith would be able to delegate UPDATE authority to another user, like Maria Perez

(USERD21B). In the following example, Daniel Smith grants update authority on the PRSTAFF column of the Project table to USERD21B:

```
    GRANT UPDATE(PRSTAFF)
       ON TABLE DSN8.TPROJ
       TO USERD21B
```

Grant Authority On Selected Rows

The SQL-responsible person in department E21 (user-id: USERE21) has been assigned to do a salary analysis. This analysis should not include managers, and consequently, USERE21 should not be allowed to "see" rows with manager information. Since we cannot specify row values in a GRANT statement, we will create a view on the employee table, and then grant select authority using this view. Using this technique we can grant authority on a field value level. The system administrator will create the following view which will include all employees found in the department table except managers:

```
    CREATE VIEW SALARY
        AS SELECT *
             FROM DSN8.TEMPL
            WHERE EMPNO NOT IN ( SELECT MGRNO
                                   FROM DSN8.TDEPT )
```

After this view has been created, the system administrator can issue a GRANT statement that will allow USERE21 to select all non-manager rows in the employee table:

```
    GRANT SELECT
       ON SALARY
       TO USERE21
```

Note that the keyword TABLE in front of the table or view name can be omitted.

Grant Authority On Selected Rows and Columns

The SQL-responsible employee in department D11 has been assigned to update information on employees in department D11 in the Employee table. For the selected rows only the names, phone numbers, job codes, and education levels should be possible to update. The system administrator will create a view that will select all rows having a work department value of D11:

```
    CREATE VIEW DEP_D11
        AS SELECT *
             FROM DSN8.TEMPL
            WHERE WORKDEPT = 'D11'
```

Following this view creation, the system administrator issues a GRANT statement that will allow USERD11 to issue updates to the specified columns in the DEP_D11 view:

```
    GRANT UPDATE(FIRSTNME, MIDINIT, LASTNAME,
                 PHONENO, JOBCODE, EDUCLVL )
       ON DEP_D11
       TO USERD11
```

List Granted Authority From System Catalog Tables

In order to verify the issued GRANT statements, the system administrator will now do a SELECT on one of the system catalog tables[3]. This system catalog table (SYSIBM.SYSTABAUTH) contains information on SELECT and UPDATE authority on the table level. We will now select all the rows, where the grantee column value starts with the character string 'USERD' or 'USERE'. The following SELECT statement is issued:

```
    SELECT DISTINCT GRANTOR, GRANTEE, TTNAME,
                    SELECTAUTH, UPDATEAUTH
      FROM SYSIBM.SYSTABAUTH
    WHERE (GRANTEE LIKE 'USERD%'
       OR  GRANTEE LIKE 'USERE%')
      AND  TCREATOR = 'DSN8'
```

The result of this statement will be a list, showing who (GRANTOR) gave authority to whom (GRANTEE) on given tables or views (TTNAME). The last two columns will indicate if select authority (SELECTAUTH) or update authority (UPDATEAUTH) was granted:

[3] For a description of the System Catalog Tables, please refer to the IBM DATABASE 2 Reference.

GRANTOR	GRANTEE	TTNAME	SELECTAUTH	UPDATEAUTH
SYSADM	USERD11	DEP_D11		Y
SYSADM	USERD11	TACTYPE	Y	
SYSADM	USERD11	TEMPRAC	Y	
SYSADM	USERD11	TPROJ	Y	
SYSADM	USERD11	TPROJAC		Y
SYSADM	USERD11	TPROJAC	Y	
SYSADM	USERD21	TACTYPE	Y	
SYSADM	USERD21	TEMPRAC	Y	
SYSADM	USERD21	TPROJ		
SYSADM	USERD21	TPROJ		Y
SYSADM	USERD21	TPROJ	Y	
SYSADM	USERD21	TPROJAC	Y	
SYSADM	USERE11	TACTYPE	Y	
SYSADM	USERE11	TEMPRAC	Y	
SYSADM	USERE11	TPROJ	Y	
SYSADM	USERE11	TPROJAC		Y
SYSADM	USERE11	TPROJAC	Y	
SYSADM	USERE21	SALARY	Y	
SYSADM	USERE21	TACTYPE	Y	
SYSADM	USERE21	TEMPRAC	Y	
SYSADM	USERE21	TPROJ	Y	
SYSADM	USERE21	TPROJAC	Y	
SYSADM	USERD21	TPROJ		G
USERD21	USERD21A	TPROJ		G
USERD21A	USERD21B	TPROJ		Y

The last three rows in the table show that the system administrator (SYSADM) has granted USERD21 update authority on the TPROJ table with GRANT OPTION (G). USERD21 has then granted USERD21A update authority on the same table also with GRANT OPTION (G). Finally, USERD21A has granted USERD21B update authority without GRANT OPTION (Y).
SYSIBM.SYSTABAUTH does not indicate which specific columns can be updated. If all columns can be updated, UPDATECOLS of the SYSTABAUTH table contains a blank. If only specific columns can be updated, UPDATECOLS contains an '*' and detailed information can be found in another system catalog table - SYSIBM.SYSCOLAUTH.

We will now do a similar select on this system catalog table in order to verify if the granted UPDATE authorities on the column level were specified correctly.

The following SELECT statement was issued by the system administrator:

```
    SELECT DISTINCT GRANTOR, GRANTEE, TNAME, COLNAME
      FROM SYSIBM.SYSCOLAUTH
    WHERE (GRANTEE LIKE 'USERD%'
       OR  GRANTEE LIKE 'USERE%')
      AND  CREATOR = 'DSN8'
```

The following result shows that the UPDATE authorities on the indicated columns are in accordance with GRANT statements issued above:

GRANTOR	GRANTEE	TNAME	COLNAME
SYSADM	USERD11	DEP_D11	EDUCLVL
SYSADM	USERD11	DEP_D11	JOBCODE
SYSADM	USERD11	DEP_D11	PHONENO
SYSADM	USERD11	DEP_D11	LASTNAME
SYSADM	USERD11	DEP_D11	MIDINIT
SYSADM	USERD11	DEP_D11	FIRSTNME
USERD21A	USERD21B	TPROJ	PRSTAFF
USERD21	USERD21A	TPROJ	PRENDATE
USERD21	USERD21A	TPROJ	PRSTDATE
USERD21	USERD21A	TPROJ	PRSTAFF
SYSADM	USERE11	TPROJAC	ACENDATE
SYSADM	USERE11	TPROJAC	ACSTDATE
SYSADM	USERD11	TPROJAC	ACENDATE
SYSADM	USERD11	TPROJAC	ACSTDATE

Note: The SYSIBM.SYSCOLAUTH table records the UPDATE privileges held by users on individual columns.
The SYSIBM.SYSTABAUTH table records <u>all</u> privileges held by users on tables and views.

Revoke SELECT Authority On a Table

In the example shown in "Grant SELECT Authority On a Table" on page 82 the user-id USERE21 was granted SELECT authority on the TEMPRAC and TPROJAC tables. This authorization is now no longer needed, so it has been decided to **REVOKE** the authorization. The system administrator will therefore issue the following REVOKE statement:

```
REVOKE SELECT
    ON DSN8.TEMPRAC, DSN8.TPROJAC
  FROM USERD21
```

If the same authorization had to be revoked from several users, a list of user-ids could have been specified.

Revoke SELECT Authority From PUBLIC

Let's assume that the system administrator previously had granted SELECT authority to all users (PUBLIC) on the Project table (TPROJ). We will now revoke this authorization by the following statement:

```
REVOKE SELECT
    ON DSN8.TPROJ
  FROM PUBLIC
```

Revoke Authority Granted With GRANT OPTION

In "Grant Authority With GRANT Option" on page 84 the following chain of UPDATE authorization on table TPROJ was issued:

```
SYSADM ——> USERD21 ——> USERD21A ——> USERD21B
```

After an auditing of the authorization system catalog tables, the manager of department D21 has decided that Maria Perez (USERD21B) must not be able to do the update on the Project table (TPROJ), granted by USERD21A. In order to revoke this authorization, USERD21A could issue a REVOKE statement, but instead the system administrator is requested to issue the necessary revoke operation:

```
REVOKE UPDATE
    ON DSN8.TPROJ
  FROM USERD21B
    BY USERD21A
```

Note that the BY user-id clause only will revoke the authorization granted by USERD21A. If Maria Perez was granted update authorization on the Project table by other users, these authorizations will not be affected. Only the system administrator can issue a REVOKE statement with a BY user-id clause.

Revoke Authority With Cascading Effect

Until now USERD21 has been responsible for maintaining the Project table (TPROJ). USERD21 has now been involved in a new project, and the maintenance of the Project table has been taken over by another employee. The system administrator will therefore issue the following REVOKE statement:

```
     REVOKE UPDATE
         ON DSN8.TPROJ
       FROM USERD21
```

The statement shown has a cascading effect on revoking authorization. Daniel Smith (USERD21A) was granted limited UPDATE authority by USERD21. However, since USERD21's authorization is revoked, USERD21A's update authorization is automatically revoked too. If USERD21A has granted other users update authority on the Project table, these authorizations would also automatically be revoked.

CHAPTER 9. DATA DEFINITION LANGUAGE - EXAMPLES

In the previous chapters we assumed that storage groups, data bases, table spaces, and tables already were defined. In this chapter we will discuss the use of the SQL Data Definition Language. The discussion will be based on examples, where we will create and do other definitions on the tables we used in the Data Manipulation Language and the Data Control Language examples.

The Data Definition Language consists of four statements:

- CREATE
- DROP
- ALTER
- COMMENT ON

Since SQL does not provide any output when an SQL Data Definition Language statement is executed (except for an acknowledgement), we will only show various statements and discuss the parameters used.

For a detailed description of the Data Definition Language, please refer to the IBM DATABASE 2 Reference.

THE CREATE STATEMENT

The following figures show the syntax of CREATE statements for the various DB2 objects.

The CREATE STOGROUP statement

Purpose: The CREATE STOGROUP statement is used to define a set of volumes controlled by a VSAM catalog. The storage group is subsequently used to allocate DB2 table spaces and indexes.

Example 1: Create a storage group named DSN8G000

```
CREATE STOGROUP DSN8G000
 VOLUMES (DSNV01)          <———   device serial number(s)
 VCAT  DSNCAT              <———   VSAM catalog alias name
 PASSWORD DSNDEFPW         <———   VSAM control level password
```

Other significant operands are:

- There are no other operands

Comments:

- All volumes in the group must be of the same device type
- The same volume can be used in multiple storage groups
- If no password is specified, DB2 will access the VSAM catalog without a password

Figure 4. CREATE STOGROUP Statement

The CREATE DATABASE statement

Purpose: The CREATE DATABASE statement is used to define a data base which will subsequently be used to collectively describe a group of DB2 table spaces and indexes.

Example 1: Create a data base named DSN8DAPP

```
CREATE DATABASE DSN8DAPP
 STOGROUP DSN8G000         <——  default storage group
 BUFFERPOOL BP1            <——  default buffer pool
```

Other significant operands are:

- There are no other operands

Comments:

- If a subsequent tablespace or index definition does not contain a buffer pool specification then the default buffer pool specification will be used.
- If a subsequent table definition does not contain a tablespace specification then a tablespace will automatically be created in the default storage group defined in the associated data base definition

Figure 5. CREATE DATABASE Statement

The CREATE TABLESPACE statement

Purpose: The CREATE TABLESPACE statement is used to allocate a table space which will subsequently contain DB2 tables.

Example 1: Create a table space named DSN8SDEP

```
CREATE TABLESPACE DSN8SDEP
 IN DSN8DAPP                  <----  data base name
 USING STOGROUP DSN8G000      <----  use existing storage group
       PRIQTY 24              <----  primary allocation in K bytes
       SECQTY 8               <----  secondary alloc. in K bytes
       ERASE NO               <----  do not erase dropped data set
 LOCKSIZE PAGE                <----  locking on page level
 BUFFERPOOL BP1               <----  buffer pool for table space
 CLOSE YES                    <----  close data set, if not in use
 DSETPASS DSN8                <----  VSAM password
```

Figure 6. CREATE TABLESPACE Statement - part 1

The CREATE TABLESPACE statement

Example 2: Create a partitioned table space named DSN8SEMP

```
CREATE TABLESPACE DSN8SEMP
 IN DSN8DAPP
 USING STOGROUP DSN8G000
       PRIQTY 24
       SECQTY 12
       ERASE NO
 NUMPARTS 4                        <——  number of partitions
  (PART 1                          <——  partition number 1
    USING STOGROUP DSN8G000
          PRIQTY 12
          SECQTY 4,
   PART 3                          <——  partition number 3
    USING STOGROUP DSN8G000
          PRIQTY 12
          SECQTY 4)
 LOCKSIZE PAGE
 BUFFERPOOL BP1
 CLOSE YES
 DSETPASS DSN8
```

Figure 7. CREATE TABLESPACE Statement - part 2

Other significant operands are:

- VCAT <—— reference VSAM catalog for space

Comments:

- The CREATE TABLESPACE statement allows you to allocate and format a table space. A table space is a unit of data storage used to contain one or more tables. The maximum addressable range is 64 Gigabytes.
- The primary space allocation (PRIQTY) is specified in Kilobytes. The minimum (and default) specification is 12 Kilobytes. The secondary space allocation (SECQTY) is also specified in Kilobytes. The minimum specification is 4 Kilobytes, the default specification is 12 Kilobytes.
- If a partitioned table space is requested, the NUMPART parameter specifies the number of partitions. This number also implicitly determines the maximum partition size (e.g. if the number of partitions is between 1 and 16, the maximum partition size is 4 Gigabytes).
- The PART parameter specifies the partition number to which the following space specification applies. Partitions for which you do not explicitly specify ~~a~~ a space allocation are assigned space as indicated by the USING parameter that applies to the whole table space.

Figure 8. CREATE TABLESPACE Statement - part 3

The CREATE TABLE statement

Purpose: The CREATE TABLE statement defines a DB2 table. Table name and column names must be specified. Attributes of the columns must also be specified.

Example 1: Create a TABLE named DSN8.TEMPL

```
CREATE TABLE DSN8.TEMPL
 (EMPNO    CHAR(6)              <——  unique column name
                NOT NULL,        <——  null values not allowed
  FIRSTNME VARCHAR(12)          <——  variable length character
                NOT NULL,                string
  MIDINIT  CHAR(1)              <——  fixed length character
                NOT NULL,                string
  LASTNAME VARCHAR(15)
                NOT NULL,
  WORKDEPT CHAR(3)
                NOT NULL,
  PHONENO  CHAR(4)      ,
  HIREDATE DECIMAL(6)   ,       <——  packed decimal format
  JOBCODE  DECIMAL(3)   ,
  EDUCLVL  SMALLINT     ,       <——  15 bit signed integer
  SEX      CHAR(1)      ,
  BRTHDATE DECIMAL(6)   ,
  SALARY   DECIMAL(8,2) )
 EDITPROC DSN8EAE1              <——  name of edit routine
 IN DSN8DAPP.DSN8SEMP           <——  DB and tablespace names
```

Figure 9. CREATE TABLE Statement - part 1

The CREATE TABLE statement

Example 2: Create a TABLE with the name DSN8.TPROJ

```
CREATE TABLE DSN8.TPROJ
  (PROJNO   CHAR(6)
                NOT NULL,
   PROJNAME VARCHAR(24)
                NOT NULL,
   DEPTNO   CHAR(3)
                NOT NULL,
   RESPEMP  CHAR(6)
                NOT NULL,
   PRSTAFF  DECIMAL(5,2),
   PRSTDATE DECIMAL(6)  ,
   PRENDATE DECIMAL(6)  ,
   MAJPROJ  CHAR(6)
                NOT NULL)
  VALIDPROC DSN8EAV1       <---- data validation routine
  IN DATABASE DSN8DAPP     <---- DB name (table space will
                                   automatically be created)
```

Figure 10. CREATE TABLE Statement - part 2

Other significant operands are:

• INTEGER	<——	31 bit signed integer
• FLOAT	<——	63 bit floating point number
• LONG VARCHAR	<——	similar to VARCHAR

Comments:

- If a column is defined as DECIMAL, you can specify the precision and scale. The first specification (precision) specifies the number of decimal digits to be stored, and must be from 1 to 15. The second specification (scale) is the number of digits to the right of the decimal point, this number must be less than or equal to the precision.

- If NOT NULL is specified, this column cannot contain null values. An attempt to insert a null value (or update to a null value) will fail.

- An edit routine can be used to compact, alter, or encrypt data. The edit routine will be executed just after the record is retrieved and just before it is stored.

- A validation routine is used to validate data (e.g. the routine may verify that a value falls within a specified range). The routine is executed just before the row is inserted or updated.

- When the DATABASE parameter is used, a table space will automatically be created. This table space will have the default table space attributes.
  The name of the table space will be derived from the table name.

- In SQL/DS a column can be specified as LONG VARCHAR. For compatibility reasons this specification can be used, but DB2 will treat it as a VARCHAR column with an internally determined length.
  Note that this precludes future use of the ALTER statement against this table. *

Figure 11. CREATE TABLE Statement - part 3

The CREATE INDEX statement

Purpose: The CREATE INDEX statement is used to create an index on a DB2 table

Example 1: Create a unique index named DSN8.XEMPL

```
CREATE UNIQUE                  <----    column contains unique values
 INDEX DSN8.XEMPL
 ON DSN8.TEMPL                 <----    name of base table
    (EMPNO    ASC)             <----    column name and ordering
 USING STOGROUP DSN8G000       <----    storage group to hold index
       PRIQTY 12               <----    primary allocation in K bytes
       ERASE NO                <----    do not erase dropped dataset
 SUBPAGES 8                    <----    number of subpages
 BUFFERPOOL BP0                <----    buffer pool for index
 CLOSE YES                     <----    close dataset, if not in use
 DSETPASS DSN8                 <----    VSAM password
```

Figure 12. CREATE INDEX Statement - part 1

The CREATE INDEX statement

Example 2: Create an index with the name DSN8.XDEPT

```
CREATE UNIQUE
 INDEX DSN8.XDEPT
 ON DSN8.TDEPT
    (DEPTNO ASC)
 USING STOGROUP DSN8G000
       PRIQTY 24
       ERASE NO
 SUBPAGES 4
 CLUSTER                         <——   create clustered index
  (PART 1 VALUES('A99'),         <——   key ranges for partitions
   PART 2 VALUES('B99'),
   PART 3 VALUES('C99'),
   PART 4 VALUES('999'))
 BUFFERPOOL BP1
 CLOSE YES
 DSETPASS DSN8
```

Figure 13. CREATE INDEX Statement - part 2

Other significant operands are:

- VCAT <—— reference VSAM catalog to get space

Comments:

- If UNIQUE is specified before any rows are inserted into the table, uniqueness of keys is guaranteed. If it is specified for a table containing rows with duplicate keys, an error message will be issued, when the index is built. If UNIQUE is not specified, duplicate key values are allowed.
- The primary space allocation (PRIQTY) is specified in Kilobytes. The minimum (and default) specification is 12 Kilobytes. The secondary space allocation (SECQTY) is also specified in Kilobytes. The minimum specification is 4 Kilobytes, the default specification is 12 Kilobytes.
- The CLUSTER parameter will result in a clustering index. You cannot specify this parameter, if you already have built another clustering index on the table.
- The PART parameter specifies ranges for the partitions of a partitioned table space. If the table space is partitioned, you <u>must</u> specify this parameter, preceded by CLUSTER.
- The VALUES parameter specifies the <u>highest</u> key value to be stored in the partition.

Figure 14. CREATE INDEX Statement - part 3

The CREATE VIEW statement

Purpose: The CREATE VIEW statement is used to define the application view of a DB2 table.

Example 1: Create a view named DSN8.VPHONE

```
CREATE VIEW DSN8.VPHONE
 (LASTNAME,                   <———  column names in view
  FIRSTNME,
  MIDDLEINITIAL,
  PHONENUMBER,
  EMPLOYEENUMBER,
  DEPTNUMBER, DEPTNAME)
 AS SELECT                    <———  underlying column(s)
  LASTNAME,
  FIRSTNME,
  MIDINIT ,
  PHONENO ,
  EMPNO,
  DEPTNO, DEPTNAME
 FROM DSN8.TEMPL,DSN8.TDEPT   <———  base tables or views
 WHERE WORKDEPT = DEPTNO      <———  join condition
```

Other significant operands are:

- WITH CHECK OPTION <——— validates INSERTs and UPDATEs

Comments:

- A view based on more than one table cannot be updated.
- A UNION operator cannot be used in a view definition.
- A view cannot have more than 16 underlying base tables.
- A view cannot be altered. If an additional column has to be added, the view must be redefined.

Figure 15. CREATE VIEW Statement

The CREATE SYNONYM statement

Purpose: The CREATE SYNONYM statement is used to define an alternate name for a DB2 table or view.

Example 1: Create a synonym named VPHONE

```
CREATE SYNONYM VPHONE        <——   synonym (alternate) name
 FOR DSN8.VPHONE             <——   table or view name
```

Other significant operands are:

- There are no other operands

Comments:

- This statement allows you to refer to a table or view owned by another user without having to enter the table's or view's fully-qualified name.
- No authorization is required to issue this statement.

Figure 16. CREATE SYNONYM Statement

THE DROP STATEMENT

The following figures show the syntax of DROP statements for the various DB2 objects.

The DROP statement

Purpose: The DROP statement is used to drop DB2 objects.

Example 1: Drop the synonym named VPHONE

```
DROP SYNONYM VPHONE          <——   drop synonym name
```

Other significant operands are:

• TABLE	<——	drop a table
• VIEW	<——	drop a view
• INDEX	<——	drop an index
• STOGROUP	<——	drop a storage group
• DATABASE	<——	drop a data base
• TABLESPACE	<——	drop a table space

Comments:

- DROP will permanently remove the specified DB2 object and its description from the system catalog
- All other DB2 objects (and their descriptions) that are directly or indirectly dependent on the object being dropped are also dropped

Figure 17. DROP Statement

THE ALTER STATEMENT

The following figures show the syntax of ALTER statements for the various DB2 objects.

The ALTER STOGROUP statement

Purpose: The ALTER STOGROUP is used to modify a previously defined DB2 storage group.

Example 1: Alter storage group DSN8G000 to add a new volume to the group

```
ALTER STOGROUP DSN8G000
 ADD VOLUMES ( DSNV02)       <——   add volume DSNV02 to storage
                                              group DSN8G000
```

Other significant operands are:

```
•   PASSWORD         <——   VSAM password

•   REMOVE VOLUMES   <——   Remove volume(s)
```

Comments:

- Removing a volume from a storage group will not affect existing data
- The password must be the VSAM control-level password that will be used to protect the VSAM catalog.

Figure 18. ALTER STOGROUP Statement

The ALTER TABLESPACE statement

Purpose: The ALTER TABLESPACE statement is used to modify a previously defined DB2 table space.

Example 1: Alter the VSAM password for table space DSN8SEMP

```
ALTER TABLESPACE DSN8SEMP
 DSETPASS DSN8A             <——  change password for table
                                        space DSN8SEMP
```

Other significant operands are:

- BUFFERPOOL <—— change buffer pool (BP0/BP1/BP2)
- LOCKSIZE <—— change locksize level
- CLOSE <—— change close parameter

Comments:

- If LOCKSIZE is specified, a level of PAGE, TABLESPACE, or ANY can be given

Figure 19. ALTER TABLESPACE Statement

The ALTER TABLE statement

Purpose: The ALTER TABLE statement is used to modify a previously defined DB2 table.

Example 1: Add a new column to table DSN8.TEMPL

```
ALTER TABLE DSN8.TEMPL
 ADD ADDRESS VARCHAR(60)     <——  add a column to a table
```

Other significant operands are:

- VALIDPROC <—— add or change a validation routine

Comments:

- When a column is added, all values of the column are NULLs.
- The added column will be the "last" column in the table
- If a validation routine is added, only <u>new</u> rows will be validated. Existing rows are not validated.
- If VALIDPROC NULL is specified, the existing validation routine is "disconnected" from the table.

Figure 20. ALTER TABLE Statement

The ALTER INDEX statement

Purpose: The ALTER INDEX statement is used to modify a previously defined DB2 index.

Example 1: Alter the buffer pool associated with index DSN8.XEMPL1

```
ALTER INDEX DSN8.XEMPL1
 BUFFERPOOL BP2            <——  change from BP0 to BP2
```

Other significant operands are:

- DSETPASS <—— change VSAM password
- CLOSE <—— change close parameter

Comments:

- This statement only changes the descriptors of an index. If the index is not open, the changes take effect immediately otherwise changes take effect next time the index is opened

Figure 21. ALTER INDEX Statement

THE COMMENT ON STATEMENT

The following figures show the syntax and examples of COMMENT ON statement.

The COMMENT ON statement

Purpose: The COMMENT ON statement is used to add a comment to the DB2 catalog tables.

Example 1: Add a comment to the catalog for table DSN8.TEMPL

```
COMMENT ON TABLE DSN8.TEMPL
 IS 'EMPLOYEE TABLE'        <----   text string (max. 254 char.)
```

Other significant operands are:

- COLUMN <---- column comment

Comments:

- If TABLE is specified, the comment will be placed in the REMARKS column of the SYSTABLES catalog table
- If COLUMN is specified, the comment will be placed in the REMARKS column of the SYSCOLUMNS catalog table

Figure 22. COMMENT ON Statement

This part of the guide consists of six appendixes showing the syntax of the SQL language and listings of the sample tables.

The first three are syntax descriptions intended to describe SQL as used in this document. Therefore they are not complete or rigorous in definition, but are supplied as a summary of the preceding chapters. A complete description of the syntax of the entire SQL language can be found in IBM DATABASE 2 Reference.

- **Appendix A. Data Manipulation Language Syntax**

  In this appendix the syntax of the SELECT, INSERT, UPDATE, and DELETE statements is shown.

- **Appendix B. Data Control Language Syntax**

  This appendix shows the syntax of the GRANT and REVOKE statements.

- **Appendix C. Data Definition Language Syntax**

  This appendix is an overview of the syntax for the CREATE, ALTER, DROP, and COMMENT ON statements.

The last three appendixes summarizes the Sample as used in this document. A complete description can be found in IBM DATABASE 2 Sample Application Guide.

- **Appendix D. Sample Base Table Definition**

  This appendix lists the table definitions of the Organization Application tables and the Project Application tables.

- **Appendix E. Organization Application Base Tables**

  This appendix lists the sample employee base table and the sample department base table.

- **Appendix F. Project Application Base Tables**

  This appendix lists the contents of the project base table, the project activity base table, the employee project activity base table, and the activity type base table.

THE SELECT STATEMENT

The SELECT statement - part 1

Purpose: The SELECT statement returns one or more rows from a specified table or tables. Only row(s) satisfying a given search condition are returned.

Format:

```
SELECT

    [ALL|DISTINCT]
     select-list|*
     FROM table-specification
    [WHERE search-condition]
    [GROUP BY column-name]
    [HAVING search-condition]
    [ORDER BY order-specification]
```

ALL|DISTINCT

- ALL (which is the default value) returns all rows which satisfy the search condition.
- DISTINCT will eliminate duplicate rows (i.e. rows where <u>all</u> specified column values are identical).

... continue

Figure 23. SELECT Statement Syntax - Part 1

The SELECT statement - part 2

select-list

Specifies the data which should be retrieved.
The list is composed of one or more elements, where each individual element can be:

- A column name
- A literal constant
- A SQL function
- An arithmetic expression combined from any of the above elements

The order of the elements can be arbitrarily specified.

When the FROM-clause specifies more than one table (**join**) and a column name thus becomes ambiguous, it is neccessary to qualify the column name. This can be made either by prefixing the column name with the table name (table-name.column-name) or with a correlation name (see FROM-clause).

*****

The '*' is a shorthand notation used to specify the selection of all column names in the table(s) specified in the FROM clause. The order of the colums is based upon the definition of the table(s). In the same way as column names can be qualified (see above), the '*' can be qualified with a table name (table-name.*) or a correlation name.

...continue

Figure 24. SELECT Statement Syntax - Part 2

The SELECT statement - part 3

FROM table-specification

The FROM-clause specifies the DB2 table or tables from which data is to be retrieved. If data is retrived from multiple tables, all table names must be specified in the FROM-clause separated by a ",". If one table name is specified two or more times in the same FROM-clause (indicating that the table is being joined to itself), each table name **must** be given a unique label so that column references can be made unambiguous (e.g. FROM TABLE1 X, TABLE1 Y).

WHERE search-condition

The WHERE-clause contains a search condition (either simple or compound) that is used to determine which rows will be retrieved from the table(s). Several conditions can be combined by AND'ing and OR'ing conditions.

GROUP BY column-name

The GROUP BY feature of DB2 permits a table to be divided into groups of rows with matching values in one or more columns. When using GROUP BY, SQL will only return one result row for each group. A grouping query may have a standard WHERE-clause which serves as a "filter", keeping only those rows which satisfy the search condition.

HAVING search-condition

The HAVING feature of DB2 is used to apply a condition to groups, causing SQL to return a result only for those groups which satisfy the condition. The HAVING-clause **must** be written after the GROUP BY-clause. The HAVING-clause may contain one or more group-qualifying predicates, connected by AND's and OR's.

ORDER BY order-specification

The ORDER BY-clause determines the order in which the query result is presented. Ordering can either be ascending (default) or descending. Ordering may be requested by one or more items (column names or expressions) of the SELECT statement. These items are specified either by column name or column number.

Figure 25. SELECT Statement Syntax - Part 3

The INSERT statement - part 1

Purpose:

The INSERT statement is used to insert a single row or multiple rows into an existing table in the DB2 data base. Two basic formats exist. Format 1 will add one new row to the table specified.

Format 1: INSERT

```
INTO table-specification
[(column-name-list)]
VALUES (list-of-values)
```

INTO table-specification

Specifies the name of the table in which the data should be added.

column-name-list

This list specifies the columns into which values are inserted. All fields not included in the list will be given a NULL value. If data is inserted in all columns no list need to be specified.

VALUES

Values are specified in column order separated by commas. A fixed length character type element will be padded with blanks on the right, if the given length is less than the defined length.
* | No padding is done to variable length character type elements.

Figure 26. INSERT Statement Syntax - Part 1

The INSERT statement - part 2

Purpose: Format 2 of the INSERT statement is used to insert into an existing table rows which are selected or computed from the DB2 data base by a SELECT statement.

Format 2:
```
INSERT

    INTO table-specification
    [(column-name-list)]
    SELECT column-names
      FROM table-specification
     WHERE search-condition
```

INTO

Same as for Format 1.

column-name-list

Same as for Format 1.

SELECT column-names

This list specifies the names of the columns from which data is retrieved. The number of column names in the list must agree with the number of columns specified in the list of column names in the INTO-clause.

FROM table-specification

The FROM-clause specifies the DB2 table or tables from which data is being retrieved. Refer to the SELECT statement for further explanation.

WHERE search-condition

The WHERE-clause contains a search condition, which will determine the rows selected from the specified table(s). Refer to the SELECT statement for further explanation.

Figure 27. INSERT Statement Syntax - Part 2

The UPDATE statement

Purpose:

The UPDATE statement is used to update the values of one or more columns in one or more rows of a DB2 table. The rows to be updated are chosen by the search condition. If no search condition is given, all rows in the specified table are updated.

Format:

```
UPDATE table-specification

       SET  column-name-1 = expression-1
          [,column-name-2 = expression-2, ...]
    [WHERE search-condition]
```

table-specification

Specifies the name of the table in which the data should be updated.

SET

The SET-clause specifies one or more new column values. These values can be constants, expressions or NULL. An expression may contain constants, column names, and the arithmetic operators +, -, *, and /.

WHERE search-condition

The WHERE-clause specifies the condition(s) that have to be met in order to do the update operation. For a detailed explanation of the WHERE-clause, please refer to the SELECT statement syntax description.

Figure 28. UPDATE Statement Syntax

The DELETE statement

Purpose:

The DELETE statement is used to delete one or more rows in a table. The rows to be deleted are chosen by the search condition. If no search condition is given, all rows in the specified table are deleted.

Format:

```
DELETE

      FROM table-specification
    [WHERE search-condition]
```

FROM table-specification

Specifies the name of the table from which the data should be deleted.

WHERE search-condition

The WHERE-clause specifies the condition(s) that have to be met for a row to be deleted. For a detailed explanation of the WHERE-clause, please refer to the SELECT statement syntax description.

Figure 29. DELETE Statement Syntax

THE GRANT STATEMENT

The following figure shows the syntax of the GRANT statement for various DB2 objects. It should be noted that there are four other formats of the GRANT statement. These formats are aimed at SYSADM, DBADM, etc. For a complete description, please refer to the IBM DATABASE 2 System Planning and Administration Guide.

The GRANT statement

Purpose: The GRANT statement is used to give authorization to other users on various DB2 objects. Only the creator of an object, a user given authorization with the GRANT OPTION, and a user with SYSADM authorization can GRANT authorization.

```
GRANT      | ALL [PRIVILEGES]    <----   same as following list
           |
           | privilege list:
               [ALTER,]          <----   table may be altered
               [DELETE,]         <----   rows may be deleted
               [INDEX,]          <----   INDEX may be created
               [INSERT,]         <----   rows may be inserted
               [SELECT,]         <----   rows may be selected
               [UPDATE  [(column-list)]]      <----   Note 1.
             ON [TABLE] table-name-list       <----   Note 2.
             TO         auth-id-list | PUBLIC <----   Note 3.
             [WITH GRANT OPTION]              <----   Note 4.
```

Note 1: Column values within the specified tables may be UPDATEd. If a a list of column names is given, then only specified columns may be UPDATEd. If no list is specified, all columns may be updated.

Note 2: The table-name-list is a list of table names or view names, or a combination of the two.

Note 3: A list of user ids can be specified if several users are granted authorization. If the specific authorization is to be granted to all users, PUBLIC can be specified.

Note 4: The granted user can GRANT the specified access to other users. The GRANT option is not allowed when granting to PUBLIC.

Figure 30. GRANT Statement Syntax

THE REVOKE STATEMENT

The following figure shows the syntax of the REVOKE statement for various DB2 objects. It should be noted that there are four other formats of the REVOKE statement. These formats are aimed at SYSADM, DBADM, etc. For a complete description, please refer to the IBM DATABASE 2 System Planning and Administration Guide.

The REVOKE statement

Purpose: The REVOKE statement is used to "take away" authorization previously given through the GRANT statement. If authorization has been GRANTed to a third user, this authorization will also be REVOKEd.

```
REVOKE    | ALL [PRIVILEGES]     <----   same as following list
          |
          | privilege list:
              [ALTER,]         <----   revoke
              [DELETE,]        <----     the indicated
              [INDEX,]         <----     privileges
              [INSERT,]        <----     ...
              [SELECT,]        <----     ...
              [UPDATE]         <----     ...
           ON [TABLE]  table-name-list          <----   Note 1.
           FROM        auth-id-list | PUBLIC    <----   Note 2.
           [BY         ALL | auth-id-list]      <----   Note 3.
```

Note 1: The table-name-list is a list of table names or view names, or a combination of the two.

Note 2: A list of user ids from which authorization is "taken away".

Note 3: The use of the BY clause is limited to those authorization ids that hold the SYSADM authorization.

Figure 31. REVOKE Statement Syntax

The following figure shows the four Data Definition statements and the DB2 objects that can be specified in these statements.

Since all the parameters of the Data Definition Language have been discussed previously ("Chapter 9. Data Definition Language - Examples" on page 91), we will not show the detailed syntax of the definition statement in this appendix. A detailed description of the four statements can be found in the IBM DATABASE 2 Reference.

The CREATE, DROP, ALTER, and COMMENT ON statement

The Data Definition statements of SQL provide facilities for creating and dropping tables and indexes, for adding new fields to existing tables, and for adding comments in the catalog tables.

The following list will indicate the various DB2 objects that can be created, altered, dropped, and commented on:

```
CREATE statement       CREATE

                           STOGROUP    <----- Create storage group
                           DATABASE    <----- Create data base
                           TABLESPACE  <----- Create table space
                           TABLE       <----- Create base table
                           INDEX       <----- Create index on table
                           VIEW        <----- Create view on table
                           SYNONYM     <----- Create synonym for table

ALTER statement        ALTER

                           STOGROUP    <----- Alter storage group
                           TABLESPACE  <----- Alter table space
                           TABLE       <----- Alter base table
                           INDEX       <----- Alter index on table

DROP statement         DROP

                           STOGROUP    <----- Drop storage group
                           DATABASE    <----- Drop data base
                           TABLESPACE  <----- Drop table space
                           TABLE       <----- Drop base table
                           INDEX       <----- Drop index on table
                           VIEW        <----- Drop view on table
                           SYNONYM     <----- Drop synonym for table

COMMENT ON statement   COMMENT ON

                           TABLE       <----- Add comment in SYSTABLES table
                           COLUMN      <----- Add comment in SYSCOLUMNS table
```

Figure 32. Data Definition Language Syntax

This appendix lists the DB2 table definitions of the base tables. The first part shows the definition of the Organization Application tables. The second part shows the definition of the Project Application tables.

Table	Field Information		
Name	Name	Format	Description
DSN8.TDEPT	DEPTNO	CHAR(3)	Department Id (unique)
	DEPTNAME	VARCHAR(36)	Department Name
Department	MGRNO	CHAR(6)	Department Manager Id
Table	ADMRDEPT	CHAR(3)	Department Id to Report to
DSN8.TEMPL	EMPNO	CHAR(6)	Employee Serial Number (unique)
	FIRSTNME	VARCHAR(12)	Employee First Name
Employee	MIDINIT	CHAR(1)	Employee Middle Initial
Table	LASTNAME	VARCHAR(15)	Employee Last Name
	WORKDEPT	CHAR(3)	Employee Department Id
	PHONENO	CHAR(4)	Employee Phone Number
	HIREDATE	DECIMAL(6)	Employee Hire Date
	JOBCODE	DECIMAL(3)	Employee Job Code
	EDUCLVL	SMALLINT	Employee Education Level
	SEX	CHAR(1)	Employee Sex Code
	BRTHDATE	DECIMAL(6)	Employee Birth Date
	SALARY	DECIMAL(8,2)	Employee Salary

Figure 33. Organization Application Tables Definition

Table	Field Information		
Name	Name	Format	Description

DSN8.TPROJ
 	
Project
Table
	PROJNO
	PROJNAME
	DEPTNO
	RESPEMP
	PRSTAFF
	PRSTDATE
	PRENDATE
	MAJPROJ
		CHAR(6)
		VARCHAR(24)
		CHAR(3)
		CHAR(6)
		DECIMAL(5,2)
		DECIMAL(6)
		DECIMAL(6)
		CHAR(6)
			Project Identification (unique)
			Project Name
			Department Id Assoc. With Proj.
			Id of Responsible Employee
			Estimated Staffing Requirement
			Estimated Project Start Date
			Estimated Project End Date
			Id of Major Project (if any)

DSN8.TACTYPE
Activity
Type Table
	ACTNO
	ACTKWD
	ACTDESC
		SMALLINT
		CHAR(6)
		VARCHAR(20)
			Activity Identification (unique)
			Activity Keyword (unique)
			Activity Description

DSN8.TPROJAC
Project
Activity
Estimates
Table
	PROJNO
	ACTNO
	ACSTAFF
	ACSTDATE
	ACENDATE
		CHAR(6)
		SMALLINT
		DECIMAL(5,2)
		DECIMAL(6)
		DECIMAL(6)
			Project Identification
			Activity Identification
			Estimated Staffing Requirement
			Estimated Activity Start Date
			Estimated Activity End Date

DSN8.TEMPRAC
 	
Employee
Project
Activity
Table
	EMPNO
	PROJNO
	ACTNO
	EMPTIME
	EMSTDATE
	EMENDATE
		CHAR(6)
		CHAR(6)
		SMALLINT
		DECIMAL(5,2)
		DECIMAL(6)
		DECIMAL(6)
			Employee Identification
			Project Identification
			Activity Identification
			Fraction of Emp. Time Allocated
			Sub-activity Start Date
			Sub-activity End Date

Figure 34. Project Application Tables Definition

APPENDIX E. ORGANIZATION APPLICATION BASE TABLES

This appendix lists the contents of the two organization application base tables.

DEPTNO	DEPTNAME	MGRNO	ADMRDEPT
A00	SPIFFY COMPUTER SERVICE DIV.	000010	
B01	PLANNING	000020	A00
C01	INFORMATION CENTER	000030	A00
D01	DEVELOPMENT CENTER		A00
E01	SUPPORT SERVICES	000050	A00
D11	MANUFACTURING SYSTEMS	000060	D01
D21	ADMINISTRATION SYSTEMS	000070	D01
E11	OPERATIONS	000090	E01
E21	SOFTWARE SUPPORT	000100	E01

Figure 35. Department Base Table (DSN8.TDEPT)

EMPNO	FIRSTNME	MIDINIT	LASTNAME	WORKDEPT	PHONENO
000010	CHRISTINE	I	HAAS	A00	3978
000020	MICHAEL	L	THOMPSON	B01	3476
000030	SALLY	A	KWAN	C01	4738
000050	JOHN	B	GEYER	E01	6789
000060	IRVING	F	STERN	D11	6423
000070	EVA	D	PULASKI	D21	7831
000090	EILEEN	W	HENDERSON	E11	5498
000100	THEODORE	Q	SPENSER	E21	0972
000110	VINCENZO	G	LUCCHESI	A00	3490
000120	SEAN		O'CONNELL	A00	2167
000130	DOLORES	M	QUINTANA	C01	4578
000140	HEATHER	A	NICHOLLS	C01	1793
000150	BRUCE		ADAMSON	D11	4510
000160	ELISABETH	R	PIANKA	D11	3782
000170	MASATOSHI	J	YOSHIMURA	D11	2890
000180	MARILYN	S	SCOUTTEN	D11	1682
000190	JAMES	H	WALKER	D11	2986
000200	DAVID		BROWN	D11	4501
000210	WILLIAM	T	JONES	D11	0942
000220	JENNIFER	K	LUTZ	D11	0672
000230	JAMES	J	JEFFERSON	D21	2094
000240	SALVATORE	M	MARINO	D21	3780
000250	DANIEL	S	SMITH	D21	0961
000260	SYBIL	V	JOHNSON	D21	8953
000270	MARIA	L	PEREZ	D21	9001
000280	ETHEL	R	SCHNEIDER	E11	8997
000290	JOHN	R	PARKER	E11	4502
000300	PHILIP	X	SMITH	E11	2095
000310	MAUDE	F	SETRIGHT	E11	3332
000320	RAMLAL	V	MEHTA	E21	9990
000330	WING		LEE	E21	2103
000340	JASON	R	GOUNOT	E21	5698

...continue

Figure 36. Employee Base Table (DSN8.TEMPL) - Part 1

HIREDATE	JOBCODE	EDUCLVL	SEX	BRTHDATE	SALARY
650101.	066.	18	F	330814.	52750.00
731010.	061.	18	M	480202.	41250.00
750405.	060.	20	F	410511.	38250.00
490817.	058.	16	M	250915.	40175.00
730914.	055.	16	M	450707.	32250.00
800930.	056.	16	F	530526.	36170.00
700815.	055.	16	F	410515.	29750.00
800619.	054.	14	M	561218.	26150.00
580516.	058.	19	M	291105.	46500.00
631205.	058.	14	M	421018.	29250.00
710728.	055.	16	F	250915.	23800.00
761215.	056.	18	F	460119.	28420.00
720212.	055.	16	M	470517.	25280.00
771011.	054.	17	F	550412.	22250.00
780915.	054.	16	M	510105.	24680.00
730707.	053.	17	F	490221.	21340.00
740726.	053.	16	M	520625.	20450.00
660303.	055.	16	M	410529.	27740.00
790411.	052.	17	M	530223.	18270.00
680829.	055.	18	F	480319.	29840.00
661121.	053.	14	M	350530.	22180.00
791205.	055.	17	M	540331.	28760.00
691030.	052.	15	M	391112.	19180.00
750911.	052.	16	F	361005.	17250.00
800930.	055.	15	F	530526.	27380.00
670324.	054.	17	F	360328.	26250.00
800530.	042.	12	M	460709.	15340.00
720619.	048.	14	M	361027.	17750.00
640912.	043.	12	F	310421.	15900.00
650707.	052.	16	M	320811.	19950.00
760223.	055.	14	M	410718.	25370.00
470505.	054.	16	M	260517.	23840.00

Figure 37. Employee Base Table (DSN8.TEMPL) - Part 2

APPENDIX F. PROJECT APPLICATION BASE TABLES

This appendix lists the contents of the four project application base tables.

ACTNO	ACTKWD	ACTDESC
10	MANAGE	MANAGE/ADVISE
20	ECOST	ESTIMATE COST
30	DEFINE	DEFINE SPECS
40	LEADPR	LEAD PROGRAM/DESIGN
50	SPECS	WRITE SPECS
60	LOGIC	DESCRIBE LOGIC
70	CODE	CODE PROGRAMS
80	TEST	TEST PROGRAMS
90	ADMQS	ADM QUERY SYSTEM
100	TEACH	TEACH CLASSES
110	COURSE	DEVELOP COURSES
120	STAFF	PERS AND STAFFING
130	OPERAT	OPER COMPUTER SYS
140	MAINT	MAINT SOFTWARE SYS
150	ADMSYS	ADM OPERATING SYS
160	ADMDB	ADM DATA BASES
170	ADMDC	ADM DATA COMM
180	DOC	DOCUMENT

Figure 38. Activity Type Base Table (DSN8.TACTYPE)

PROJNO	PROJNAME	DEPTNO	RESPEMP	PRSTAFF	PRSTDATE	PRENDATE	MAJPROJ
MA2100	WELD LINE AUTOMATION	D01	000010	12.00	820101.	830201.	
MA2110	W L PROGRAMMING	D11	000060	9.00	820101.	830201.	MA2100
MA2111	W L PROGRAM DESIGN	D11	000220	2.00	820101.	821201.	MA2110
MA2112	W L ROBOT DESIGN	D11	000150	3.00	820101.	821201.	MA2110
MA2113	W L PROD CONT PROGS	D11	000160	3.00	820215.	821201.	MA2110
PL2100	WELD LINE PLANNING	B01	000020	1.00	820101.	820915.	MA2100
IF1000	QUERY SERVICES	C01	000030	2.00	820101.	830201.	
IF2000	USER EDUCATION	C01	000030	1.00	820101.	830201.	
AD3100	ADMIN SERVICES	D01	000010	6.50	820101.	830201.	
AD3110	GENERAL AD SYSTEMS	D21	000070	6.00	820101.	830201.	AD3100
AD3111	PAYROLL PROGRAMMING	D21	000230	2.00	820101.	830201.	AD3110
AD3112	PERSONNEL PROGRAMMG	D21	000250	1.00	820101.	830201.	AD3110
AD3113	ACCOUNT.PROGRAMMING	D21	000270	2.00	820101.	830201.	AD3110
OP1000	OPERATION SUPPORT	E01	000050	6.00	820101.	830201.	
OP1010	OPERATION	E11	000090	5.00	820101.	830201.	OP1000
OP2000	GEN SYSTEMS SERVICES	E01	000050	5.00	820101.	830201.	
OP2010	SYSTEMS SUPPORT	E21	000100	4.00	820101.	830201.	OP2000
OP2011	SCP SYSTEMS SUPPORT	E21	000320	1.00	820101.	830201.	OP2010
OP2012	APPLICATIONS SUPPORT	E21	000330	1.00	820101.	830201.	OP2010
OP2013	DB/DC SUPPORT	E21	000340	1.00	820101.	830201.	OP2010

Figure 39. Project Base Table (DSN8.TPROJ)

PROJNO	ACTNO	ACSTAFF	ACSTDATE	ACENDATE
MA2100	10	.50	820101.	821101.
MA2100	20	1.00	820101.	820301.
MA2110	10	1.00	820101.	830201.
MA2111	40	1.00	820101.	830201.
MA2111	50	1.00	820101.	820601.
MA2111	60	1.00	820601.	830201.
MA2112	60	2.00	820101.	820701.
MA2112	180	1.00	820701.	830201.
MA2112	70	1.50	820215.	830201.
MA2113	80	1.50	820901.	830201.
MA2113	60	1.00	820215.	820901.
MA2113	70	2.00	820401.	821215.
MA2113	180	.50	821001.	830101.
PL2100	30	1.00	820201.	820901.
IF1000	90	1.00	820101.	830101.
IF1000	100	.50	821001.	830101.
IF1000	10	.50	820101.	830101.
IF2000	10	.50	820101.	830101.
IF2000	100	.75	820101.	820701.
IF2000	110	.50	820301.	820701.
IF2000	110	.50	821001.	830101.
AD3100	10	.50	820101.	820701.
AD3110	10	1.00	820101.	830101.
AD3111	60	.80	820101.	820415.
AD3111	70	1.50	820215.	821015.
AD3111	180	1.00	821015.	830115.
AD3111	80	1.25	820415.	830115.
AD3112	60	.75	820101.	820315.
AD3112	60	.75	821201.	830101.
AD3112	70	.75	820101.	821015.
AD3112	80	.35	820815.	821201.
AD3112	180	.50	820815.	830101.
AD3113	60	.75	820301.	821015.
AD3113	70	1.25	820601.	821215.
AD3113	80	1.75	820101.	820415.
AD3113	180	.75	820301.	820701.
OP1000	10	.25	820101.	830201.
OP2000	50	.75	820101.	830201.
OP1010	10	1.00	820101.	830201.
OP1010	130	4.00	820101.	830201.
OP2010	10	1.00	820101.	830201.
OP2011	140	.75	820101.	830201.
OP2011	150	.25	820101.	830201.
OP2012	160	1.00	820101.	830201.
OP2013	170	1.00	820101.	830201.

Figure 40. Project-Activity Base Table (DSN8.TPROJAC)

EMPNO	PROJNO	ACTNO	EMPTIME	EMSTDATE	EMENDATE
000010	AD3100	10	.50	820101.	820701.
000070	AD3110	10	1.00	820101.	830201.
000230	AD3111	60	1.00	820101.	820315.
000240	AD3111	70	1.00	820215.	820915.
000230	AD3111	60	.50	820315.	820415.
000230	AD3111	70	.50	820315.	821015.
000230	AD3111	80	.50	820415.	821015.
000240	AD3111	80	1.00	820915.	830101.
000230	AD3111	180	1.00	821015.	830101.
000250	AD3112	60	1.00	820101.	820201.
000250	AD3112	60	.50	820201.	820315.
000250	AD3112	70	.50	820201.	820315.
000250	AD3112	70	1.00	820315.	820815.
000250	AD3112	70	.25	820815.	821015.
000250	AD3112	80	.25	820815.	821015.
000250	AD3112	180	.50	820815.	830101.
000250	AD3112	80	.50	821015.	821201.
000250	AD3112	60	.50	821201.	830101.
000250	AD3112	60	1.00	830101.	830201.
000260	AD3113	80	1.00	820101.	820301.
000270	AD3113	80	1.00	820101.	820301.
000270	AD3113	60	.50	820301.	820401.
000260	AD3113	80	.50	820301.	820415.
000270	AD3113	80	.50	820301.	820401.
000260	AD3113	180	.50	820301.	820415.
000270	AD3113	60	1.00	820401.	820901.
000260	AD3113	180	1.00	820415.	820601.
000260	AD3113	180	.50	820601.	820701.
000260	AD3113	70	.50	820615.	820701.
000260	AD3113	70	1.00	820701.	830201.
000270	AD3113	60	.25	820901.	821015.
000270	AD3113	70	.75	820901.	821015.
000270	AD3113	70	1.00	821015.	830201.
000130	IF1000	90	1.00	820101.	821001.
000030	IF1000	10	.50	820601.	830101.
000140	IF1000	90	.50	821001.	830101.
000130	IF1000	100	.50	821001.	830101.
000030	IF2000	10	.50	820101.	830101.
000140	IF2000	100	1.00	820101.	820301.
000140	IF2000	100	.50	820301.	820701.
000140	IF2000	110	.50	820301.	820701.
000140	IF2000	110	.50	821001.	830101.
...	...	...	...	...	...

continue

Figure 41. Employee-Project-Activity Base Table (DSN8.TEMPRAC) - Part 1

...	...	...	...	...	...
000010	MA2100	10	.50	820101.	821101.
000110	MA2100	20	1.00	820101.	820301.
000010	MA2110	10	1.00	820101.	830201.
000220	MA2111	40	1.00	820101.	830201.
000200	MA2111	50	1.00	820101.	820615.
000200	MA2111	60	1.00	820615.	830201.
000150	MA2112	60	1.00	820101.	820715.
000170	MA2112	60	1.00	820101.	830601.
000190	MA2112	70	1.00	820201.	821001.
000170	MA2112	70	1.00	820601.	830201.
000150	MA2112	180	1.00	820715.	830201.
000190	MA2112	80	1.00	821001.	831001.
000170	MA2113	80	1.00	820101.	830201.
000180	MA2113	70	1.00	820401.	821215.
000160	MA2113	60	1.00	820715.	830201.
000210	MA2113	80	.50	821001.	830201.
000210	MA2113	180	.50	821001.	830201.
000050	OP1000	10	.25	820101.	830201.
000090	OP1010	10	1.00	820101.	830201.
000280	OP1010	130	1.00	820101.	830201.
000290	OP1010	130	1.00	820101.	830201.
000300	OP1010	130	1.00	820101.	830201.
000310	OP1010	130	1.00	820101.	830201.
000050	OP2010	10	.75	820101.	830201.
000100	OP2010	10	1.00	820101.	830201.
000320	OP2011	140	.75	820101.	830201.
000320	OP2011	150	.25	820101.	830201.
000330	OP2012	160	1.00	820101.	830201.
000340	OP2013	170	1.00	820101.	830201.

Figure 42. Employee-Project-Activity Base Table (DSN8.TEMPRAC) - Part 2

GG24-1583-00 READER'S COMMENT FORM

IBM DATABASE 2 SQL Usage Guide

Reader's Comment Form

Cut or Fold Along Line

Fold

IBM INTERNATIONAL SYSTEMS CENTER
Department 471
Building F27
555 Bailey Avenue
P. O. Box 50020
San Jose, California 95150
U.S.A.

Fold

IBM

GG24-1583-00 **READER'S COMMENT FORM**

IBM DATABASE 2 SQL Usage Guide

This form may be used to communicate your views about this publication. They will be sent to the author's department for whatever review and action, if any, is deemed appropriate. Comments may be written in your own language; use of English is not required.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever.

Note: Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.

Possible topics for comment are:

Clarity Accuracy Completeness Organization Coding Retrieval Legibility

What is your occupation?

____

Number of latest TNL applied:

____

Thank you for your cooperation.

If you wish a reply, please give your name and mailing address:

____

____

____

____

Reader's Comment Form

Cut or Fold Along Line

Fold

IBM INTERNATIONAL SYSTEMS CENTER
Department 471
Building F27
555 Bailey Avenue
P. O. Box 50020
San Jose, California 95150
U.S.A.

Fold

IBM

GG24-1583-00 **READER'S COMMENT FORM**

IBM DATABASE 2 SQL Usage Guide

This form may be used to communicate your views about this publication. They will be sent to the author's department for whatever review and action, if any, is deemed appropriate. Comments may be written in your own language; use of English is not required.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever.

Note: Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.

Possible topics for comment are:

Clarity Accuracy Completeness Organization Coding Retrieval Legibility

What is your occupation?

____

Number of latest TNL applied:

____

Thank you for your cooperation.

If you wish a reply, please give your name and mailing address:

____

____

____

____

Reader's Comment Form

Cut or Fold Along Line

Fold

IBM INTERNATIONAL SYSTEMS CENTER
Department 471
Building F27
555 Bailey Avenue
P. O. Box 50020
San Jose, California   95150
U.S.A.

Fold

IBM

GG24-1583

IBM DATABASE 2 SQL Usage Guide GG24-1583 Printed in the U.S.A.

IBM®