# Db2 SQL Cookbook

# Content

Graeme Birchall (until Version 9.7, 2011)
Version 1.0, Rodney Krick, November 2019
Contributors: Andres Gomez Casanova

# Introduction

Project in GitHub: https://github.com/rodneykrick/db2-sql-cookbook The PDF version can be found here: http://db2-sql-cookbook.org/pdf/Db2_SQL_Cookbook.pdf.

This version of the book is intended to be a community effort to help people learn SQL. Join, improve, enjoy, have fun! == Document history

(The history of the original book can bee seen at Graeme Birchall Book Editions Upload Dates)

| Version | Date | Content |
|---|---|---|
| 1.0 | 9.10.2019 | Started new project in GitHub. Text transformed to asciidoc.<br>Syntax diagrams were removed.<br>Cross references updated. |
| 1.1 | 24.11.2019 | Added license terms |
| 1.2 | 06.12.2019 | Update functions to Db2 version 11.1 |
| 1.3 | 18.01.2020 | Added Time Travel chapter. |
| 1.4 | 30.08.2022 | Personal notes removed from introduction. |

# Chapter 1. Quick find

## 1.1. Index of Concepts

### 1.1.1. Join rows

To combine matching rows in multiple tables, use a join (see Joins).

EMP_NM

| ID | NAME |
|---|---|
| 10 | Sanders |
| 20 | Pernal |
| 50 | Hanes |

EMP_JB

| ID | JOB |
|---|---|
| 10 | Sales |
| 20 | Clerk |

*Join example*

```
SELECT nm.id
     , nm.name
     , jb.job
FROM emp_nm nm
   , emp_jb jb
WHERE nm.id = jb.id
ORDER BY 1;
```

*ANSWER*

| ID | NAME | JOB |
|---|---|---|
| 10 | Sanders | Sales |
| 20 | Pernal | Clerk |

### 1.1.2. Outer Join

To get all of the rows from one table, plus the matching rows from another table (if there are any), use an outer join (see Join Types).

EMP_NM

| ID | NAME |
|---|---|
| 10 | Sanders |
| 20 | Pernal |
| 50 | Hanes |

EMP_JB

| ID | JOB |
|---|---|
| 10 | Sales |
| 20 | Clerk |

*Left-outer-join example*

```
SELECT nm.id ,nm.name ,jb.job
FROM emp_nm nm
LEFT OUTER JOIN emp_jb jb
ON nm.id = jb.id
ORDER BY nm.id;
```

*ANSWER*

| ID | NAME | JOB |
|---|---|---|
| 10 | Sanders | Sales |
| 20 | Pernal | Clerk |
| 50 | Hanes | - |

To get rows from either side of the join, regardless of whether they match (the join) or not, use a full outer join (see Full Outer Joins).

### 1.1.3. Null Values & Replace

Use the COALESCE function (see COALESCE) to replace a null value (e.g. generated in an outer join) with a non-null value.

### 1.1.4. Select Where No Match

To get the set of the matching rows from one table where something is true or false in another table (e.g. no corresponding row), use a sub-query (see Sub-Query).

EMP_NM

| ID | NAME |
|---|---|
| 10 | Sanders |
| 20 | Pernal |

| ID | NAME |
|---|---|
| 50 | Hanes |

EMP_JB

| ID | JOB |
|---|---|
| 10 | Sales |
| 20 | Clerk |

*Sub-query example*

```
SELECT * FROM emp_nm nm
WHERE NOT EXISTS
  (SELECT * FROM emp_jb jb
   WHERE nm.id = jb.id)
ORDER BY id;
```

*ANSWER*

| ID | NAME |
|---|---|
| 50 | Hanes |

### 1.1.5. Append Rows

To add (append) one set of rows to another set of rows, use a union (see Union, Intersect, and Except).

EMP_NM

| ID | NAME |
|---|---|
| 10 | Sanders |
| 20 | Pernal |
| 50 | Hanes |

EMP_JB

| ID | JOB |
|---|---|
| 10 | Sales |
| 20 | Clerk |

*Union example*

```
SELECT *
FROM emp_nm
WHERE emp_nm name < 'S'
  UNION
SELECT *
FROM emp_jb
ORDER BY 1, 2;
```

*ANSWER*

| ID | 2 |
|---|---|
| 10 | Sales |
| 20 | Clerk |
| 20 | Pernal |
| 50 | Hanes |

### 1.1.6. Assign Output Numbers

To assign line numbers to SQL output, use the ROW_NUMBER function (see ROW_NUMBER).

EMP_JB

| ID | JOB |
|---|---|
| 10 | Sales |
| 20 | Clerk |

*Assign row-numbers example*

```
SELECT id
     , job
     , ROW_NUMBER() OVER(ORDER BY job) AS R
FROM emp_jb
ORDER BY job;
```

*ANSWER*

| ID | JOB | R |
|---|---|---|
| 20 | Clerk | 1 |
| 10 | Sales | 2 |

### 1.1.7. Assign Unique Key Numbers

To make each row inserted into a table automatically get a unique key value, use an identity

column, or a sequence, when creating the table (see Identity Columns and Sequences).

### 1.1.8. If-Then-Else Logic

To include if-then-else logical constructs in SQL stmts, use the CASE phrase (see CASE Expression).

EMP_JB

| ID | JOB |
|---|---|
| 10 | Sales |
| 20 | Clerk |

*Case stmt example*

```
SELECT id
     , job
     , CASE
         WHEN job = 'Sales' THEN 'Fire'
         ELSE 'Demote'
       END AS STATUS
 FROM emp_jb;
```

*ANSWER*

| ID | JOB | STATUS |
|---|---|---|
| 10 | Sales | Fire |
| 20 | Clerk | Demote |

### 1.1.9. Get Dependents

To get all of the dependents of some object, regardless of the degree of separation from the parent to the child, use recursion (see Recursive SQL).

FAMILY

| PARNT | CHILD |
|---|---|
| GrDad | Dad |
| Dad | Dghtr |
| Dghtr | GrSon |
| Dghtr | GrDtr |

*Recursion example*

```
WITH temp (persn, lvl)
AS (SELECT parnt, 1
    FROM family
    WHERE parnt = 'Dad'
      UNION ALL
    SELECT child, Lvl + 1
    FROM temp, family
    WHERE persn = parnt)
SELECT * FROM temp;
```

*ANSWER*

| PERSN | LVL |
|---|---|
| Dad | 1 |
| Dghtr | 2 |
| GrSon | 3 |
| GrDtr | 3 |

### 1.1.10. Convert String to Rows

To convert a (potentially large) set of values in a string (character field) into separate rows (e.g. one row per word), use recursion (see Recursive SQL).

**INPUT DATA** "Some silly text" Use Recursive SQL

*ANSWER*

*Table 1. Convert string to rows*

| TEXT | LINE# |
|---|---|
| Some | 1 |
| silly | 2 |
| text | 3 |

Be warned - in many cases, the code is not pretty.

### 1.1.11. Convert Rows to String

To convert a (potentially large) set of values that are in multiple rows into a single combined field, use recursion (see Recursive SQL).

INPUT DATA

| TEXT | LINE# |
|---|---|
| Some | 1 |
| silly | 2 |
| text | 3 |

Use Recursive SQL

ANSWER: "Some silly text"

### 1.1.12. Fetch First "n" Rows

To fetch the first "n" matching rows, use the FETCH FIRST notation (see FETCH FIRST Clause).

EMP_NM

| ID | NAME |
|---|---|
| 10 | Sanders |
| 20 | Pernal |
| 50 | Hanes |

*Fetch first "n" rows example*

```
SELECT * FROM
emp_nm
ORDER BY id DESC
FETCH FIRST 2 ROWS ONLY;
```

*ANSWER*

| ID | NAME |
|---|---|
| 50 | Hanes |
| 20 | Pernal |

Another way to do the same thing is to assign row numbers to the output, and then fetch those rows where the row-number is less than "n" (see Selecting "n" Rows).

### 1.1.13. Fetch Subsequent "n" Rows

To the fetch the "n" through "n + m" rows, first use the ROW_NUMBER function to assign output numbers, then put the result in a nested-table-expression, and then fetch the rows with desired numbers.

### 1.1.14. Fetch Uncommitted Data

To retrieve data that may have been changed by another user, but which they have yet to commit,

use the WITH UR (Uncommitted Read) notation.

EMP_NM

| ID | NAME |
|---|---|
| 10 | Sanders |
| 20 | Pernal |
| 50 | Hanes |

*Fetch WITH UR example*

```
SELECT *
FROM emp_nm
WHERE name like 'S%'
WITH UR;
```

*ANSWER*

| ID | NAME |
|---|---|
| 10 | Sanders |

Using this option can result in one fetching data that is subsequently rolled back, and so was never valid. Use with extreme care.

### 1.1.15. Summarize Column Contents

Use a column function (see Column Functions or Aggregate Functions) to summarize the contents of a column.

EMP_NM

| ID | NAME |
|---|---|
| 10 | Sanders |
| 20 | Pernal |
| 50 | Hanes |

*Column Functions example*

```
SELECT AVG(id)AS avg
      ,MAX(name) AS maxn
      ,COUNT(*) AS #rows
FROM emp_nm;
```

*ANSWER*

| AVG | MAXN | #ROWS |
|---|---|---|
| 26 | Sanders | 3 |

### 1.1.16. Subtotals and Grand Totals

To obtain subtotals and grand-totals, use the ROLLUP or CUBE statements (see ROLLUP Statement).

*Subtotal and Grand-total example*

```
SELECT job
      ,dept
      ,SUM(salary) AS sum_sal
      ,COUNT(*) AS #emps
FROM staff
WHERE dept < 30
AND salary < 90000
AND job < 'S'
GROUP BY ROLLUP(job, dept)
ORDER BY job, dept;
```

ANSWER

| JOB | DEPT | SUM_SAL | #EMPS |
|---|---|---|---|
| Clerk | 15 | 84766.70 | 2 |
| Clerk | 20 | 77757.35 | 2 |
| Clerk | - | 162524.05 | 4 |
| Mgr | 10 | 243453.45 | 3 |
| Mgr | 15 | 80659.80 | 1 |
| Mgr | - | 324113.25 | 4 |
| - | - | 486637.30 | 8 |

### 1.1.17. Enforcing Data Integrity

When a table is created, various Db2 features can be used to ensure that the data entered in the table is always correct:

- Uniqueness (of values) can be enforced by creating unique indexes.
- Check constraints can be defined to limit the values that a column can have.
- Default values (for a column) can be defined - to be used when no value is provided.
- Identity columns (see Identity Columns and Sequences), can be defined to automatically generate unique numeric values (e.g. invoice numbers) for all of the rows in a table. Sequences can do the same thing over multiple tables.
- Referential integrity rules can be created to enforce key relationships between tables.

- Triggers can be defined to enforce more complex integrity rules, and also to do things (e.g. populate an audit trail) whenever data is changed.

See the Db2 manuals for documentation or Protecting Your Data for more information about the above.

### 1.1.18. Hide Complex SQL

One can create a view (see View) to hide complex SQL that is run repetitively. Be warned however that doing so can make it significantly harder to tune the SQL - because some of the logic will be in the user code, and some in the view definition.

## 1.2. Summary Table

Some queries that use a GROUP BY can be made to run much faster by defining a summary table (see Materialized Query Tables) that Db2 automatically maintains. Subsequently, when the user writes the original GROUP BY against the source-data table, the optimizer substitutes with a much simpler (and faster) query against the summary table.

# Chapter 2. Introduction to SQL

This chapter contains a basic introduction to Db2 SQL. It also has numerous examples illustrating how to use this language to answer particular business problems. However, it is not meant to be a definitive guide to the language. Please refer to the relevant IBM manuals for a more detailed description.

## 2.1. Syntax Diagram

The original book has lots of syntax diagrams in it. We decided not to put them here anymore. Syntax diagrams can be found in the SQL Reference of Db2 (or your prefered RDBMS). We only give you lots of examples how to use SQL. When you master this stuff, you should learn more and improve your knowledge using the "real" manuals.

## 2.2. SQL Comments

A comment in a SQL statement starts with two dashes and goes to the end of the line:

*SQL Comment example*

```
SELECT name
FROM staff
ORDER BY id;
-- this is a comment.
-- this is another comment.
```

Some Db2 command processors (e.g. db2batch on the PC, or SPUFI on the mainframe) can process intelligent comments. These begin the line with a `--#SET` phrase, and then identify the value to be set. In the following example, the statement delimiter is changed using an intelligent comment:

*Set Delimiter example*

```
--#SET DELIMITER !
SELECT name
FROM staff
WHERE id = 10!
--#SET DELIMITER ;
SELECT name
FROM staff
WHERE id = 20;
```

When using the Db2 Command Processor (batch) script, the default statement terminator can be set using the "-tdx" option, where "x" is the value have chosen.

See the section titled Special Character Usage for notes on how to refer to the statement delimiter in the SQL text.

## 2.3. Statement Delimiter

Db2 SQL does not come with a designated statement delimiter (terminator), though a semicolon is often used. A semi-colon cannot be used when writing a compound SQL statement (see Compound SQL) because that character is used to terminate the various subcomponents of the statement.

## 2.4. SQL Components

### 2.4.1. Db2 Objects

Db2 is a relational database that supports a variety of object types. In this section we shall overview those items which one can obtain data from using SQL.

#### Table

A table is an organized set of columns and rows. The number, type, and relative position, of the various columns in the table is recorded in the Db2 catalogue. The number of rows in the table will fluctuate as data is inserted and deleted. The CREATE TABLE statement is used to define a table. The following example will define the EMPLOYEE table, which is found in the Db2 sample database.

*Db2 sample table – EMPLOYEE*

```
CREATE TABLE employee
( empno     CHARACTER(6) NOT NULL
, firstnme  VARCHAR(12)  NOT NULL
, midinit   CHARACTER(1) NOT NULL
, lastname  VARCHAR(15)  NOT NULL
, workdept  CHARACTER(3)
, phoneno   CHARACTER(4)
, hiredate  DATE
, job       CHARACTER(8)
, edlevel   SMALLINT     NOT NULL
, sex       CHARACTER(1)
, birthdate DATE
, salary    DECIMAL(9,02)
, bonus     DECIMAL(9,02)
, comm      DECIMAL(9,02)
);
```

#### View

A view is another way to look at the data in one or more tables (or other views). For example, a user of the following view will only see those rows (and certain columns) in the EMPLOYEE table where the salary of a particular employee is greater than or equal to the average salary for their particular department.

*Db2 sample view – EMPLOYEE_VIEW*

```
CREATE VIEW employee_view AS
SELECT a.empno
     , a.firstnme
     , a.salary
     , a.workdept
FROM employee a
WHERE a.salary >=
(SELECT AVG(b.salary)
 FROM employee b
 WHERE a.workdept = b.workdept
);
```

A view need not always refer to an actual table. It may instead contain a list of values:

*Define a view using a VALUES clause*

```
CREATE VIEW silly (c1, c2, c3)
AS VALUES
 (11, 'AAA', SMALLINT(22))
,(12, 'BBB', SMALLINT(33))
,(13, 'CCC', NULL);
```

Selecting from the above view works the same as selecting from a table:

*SELECT from a view that has its own data*

```
SELECT c1, c2, c3
FROM silly
ORDER BY c1 ASC;
```

*ANSWER*

| C1 | C2 | C3 |
|---|---|---|
| 11 | AAA | 22 |
| 12 | BBB | 33 |
| 13 | CCC | - |

We can go one step further and define a view that begins with a single value that is then manipulated using SQL to make many other values. For example, the following view, when selected from, will return 10,000 rows. Note however that these rows are not stored anywhere in the database - they are instead created on the fly when the view is queried.

*Define a view that creates data on the fly*

```
CREATE VIEW test_data AS
WITH temp1 (num1) AS
 (VALUES (1)
  UNION ALL
 SELECT num1 + 1
 FROM temp1
 WHERE num1 < 10000)
SELECT *
FROM temp1;
```

### Alias

An alias is an alternate name for a table or a view. Unlike a view, an alias can not contain any processing logic. No authorization is required to use an alias other than that needed to access to the underlying table or view.

*Define three aliases, the latter on the earlier*

```
CREATE ALIAS employee_al1 FOR employee;
COMMIT;

CREATE ALIAS employee_al2 FOR employee_al1;
COMMIT;

CREATE ALIAS employee_al3 FOR employee_al2;
COMMIT;
```

Neither a view, nor an alias, can be linked in a recursive manner (e.g. V1 points to V2, which points back to V1). Also, both views and aliases still exist after a source object (e.g. a table) has been dropped. In such cases, a view, but not an alias, is marked invalid.

### Nickname

A nickname is the name that one provides to Db2 for either a remote table, or a non-relational object that one wants to query as if it were a table.

*Define a nickname*

```
CREATE NICKNAME emp FOR unixserver.production.employee;
```

### Tablesample

Use of the optional TABLESAMPLE reference enables one to randomly select (sample) some fraction of the rows in the underlying base table:

*TABLESAMPLE example*

```
SELECT *
FROM staff
TABLESAMPLE BERNOULLI(10);
```

See Randomly Sample Data for information on using the TABLESAMPLE feature.

### 2.4.2. Db2 Data Types

Db2 comes with the following standard data types:

- SMALLINT, INT, and BIGINT (i.e. integer numbers).
- FLOAT, REAL, and DOUBLE (i.e. floating point numbers).
- DECIMAL and NUMERIC (i.e. decimal numbers).
- DECFLOAT (i.e. decimal floating-point numbers).
- CHAR, VARCHAR, and LONG VARCHAR (i.e. character values).
- GRAPHIC, VARGRAPHIC, and LONG VARGRAPHIC (i.e. graphical values).
- BLOB, CLOB, and DBCLOB (i.e. binary and character long object values).
- DATE, TIME, and TIMESTAMP (i.e. date/time values).
- DATALINK (i.e. link to external object).
- XML (i.e. contains well formed XML data).

Below is a simple table definition that uses some of the above data types:

*Sample table definition*

```
CREATE TABLE sales_record
(sales#         INTEGER       NOT NULL
                GENERATED ALWAYS AS IDENTITY (START WITH 1, INCREMENT BY 1)
, sale_ts       TIMESTAMP     NOT NULL
, num_items     SMALLINT      NOT NULL
, payment_type  CHAR(2)       NOT NULL
, sale_value    DECIMAL(12,2) NOT NULL
, sales_tax     DECIMAL(12,2)
, employee#     INTEGER       NOT NULL
, CONSTRAINT sales1 CHECK (payment_type IN ('CS','CR'))
, CONSTRAINT sales2 CHECK (sale_value > 0)
, CONSTRAINT sales3 CHECK (num_items > 0)
, CONSTRAINT sales4 FOREIGN KEY (employee#)
  REFERENCES staff (id) ON DELETE RESTRICT
, PRIMARY KEY (sales#)
);
```

In the above table, we have listed the relevant columns, and added various checks to ensure that the data is always correct. In particular, we have included the following:

- The sales# is automatically generated (see Identity Columns and Sequences for details). It is also the primary key of the table, and so must always be unique.
- The payment-type must be one of two possible values.
- Both the sales-value and the num-items must be greater than zero.
- The employee# must already exist in the staff table. Furthermore, once a row has been inserted into this table, any attempt to delete the related row from the staff table will fail.

#### Default Lengths

The following table has two columns:

*Table with default column lengths*

```
CREATE TABLE default_values
(c1 CHAR    NOT NULL
,d1 DECIMAL NOT NULL);
```

The length has not been provided for either of the above columns. In this case, Db2 defaults to CHAR(1) for the first column and DECIMAL(5,0) for the second column.

#### Data Type Usage

In general, use the standard Db2 data types as follows:

- Always store monetary data in a decimal field.
- Store non-fractional numbers in one of the integer field types.
- Use floating-point when absolute precision is not necessary.

A Db2 data type is not just a place to hold data. It also defines what rules are applied when the data in manipulated. For example, storing monetary data in a Db2 floating-point field is a no-no, in part because the data-type is not precise, but also because a floating-point number is not manipulated (e.g. during division) according to internationally accepted accounting rules.

#### DECFLOAT Arithmetic

DECFLOAT numbers have quite different processing characteristics from the other number types. For a start, they support more values:

- Zero.
- Negative and positive numbers (e.g. -1234.56).
- Negative and positive infinity.
- Negative and positive NaN (i.e. Not a Number).
- Negative and positive sNaN (i.e. signaling Not a Number).

### NaN Usage

The value NaN represents the result of an arithmetic operation that does not return a number (e.g. the square root of a negative number), but is also not infinity. For example, the expression 0/0 returns NaN, while 1/0 returns infinity.

The value NaN propagates through any arithmetic expression. Thus the final result is always either positive or negative NaN, as the following query illustrates:

*NaN arithmetic usage*

```
SELECT    DECFLOAT(+1.23)       +  NaN  AS "  NaN"
        , DECFLOAT(-1.23)       +  NaN  AS "  NaN"
        , DECFLOAT(-1.23)       + -NaN  AS " -NaN"
        , DECFLOAT(+infinity)   +  NaN  AS "  NaN"
        , DECFLOAT(+sNaN)       +  NaN  AS "  NaN"
        , DECFLOAT(-sNaN)       +  NaN  AS " -NaN"
        , DECFLOAT(+NaN)        +  NaN  AS "  NaN"
        , DECFLOAT(-NaN)        +  NaN  AS " -NaN"
FROM sysibm.sysdummy1;
```

Any reference to a signaling NaN value in a statement (as above) will result in a warning message being generated.

### Infinity Usage

The value infinity works similar to NaN. Its reference in an arithmetic expression almost always returns either positive or negative infinity (assuming NaN is not also present). The one exception is division by infinity, which returns a really small, but still finite, number:

*Infinity arithmetic usage*

```
SELECT  DECFLOAT(1) / +infinity AS      " 0E-6176"
    ,   DECFLOAT(1) * +infinity AS      " Infinity"
    ,   DECFLOAT(1) + +infinity AS      " Infinity"
    ,   DECFLOAT(1) - +infinity AS      "-Infinity"
    ,   DECFLOAT(1) / -infinity AS      " -0E-6176"
    ,   DECFLOAT(1) * -infinity AS      "-Infinity"
    ,   DECFLOAT(1) + -infinity AS      "-Infinity"
    ,   DECFLOAT(1) - -infinity AS      " Infinity"
FROM sysibm.sysdummy1;
```

The next query shows some situations where either infinity or NaN is returned:

*DECFLOAT arithmetic results*

```
SELECT  DECFLOAT(+1.23) / 0           AS " Infinity"
    ,   DECFLOAT(-1.23) / 0           AS "-Infinity"
    ,   DECFLOAT(+1.23) + infinity    AS " Infinity"
    ,   DECFLOAT(0)     / 0           AS "NaN"
    ,   DECFLOAT(infinity) + -infinity AS "NaN"
    ,   LOG(DECFLOAT(0))              AS "-Infinity"
    ,   LOG(DECFLOAT(-123))           AS "NaN"
    ,   SQRT(DECFLOAT(-123))          AS "NaN"
FROM sysibm.sysdummy1;
```

**DECFLOAT Value Order**

The DECFLOAT values have the following order, from low to high:

*DECFLOAT value order*

```
-NaN -sNan -infinity -1.2 -1.20 0 1.20 1.2 infinity sNaN NaN
```

Please note that the numbers 1.2 and 1.200 are "equal", but they will be stored as different values, and will have a different value order. The TOTALORDER function can be used to illustrate this. It returns one of three values:

- Zero if the two values have the same order.
- +1 if the first value has a higher order (even if it is equal).
- -1 if the first value has a lower order (even if it is equal).

*Equal values that may have different orders*

```
WITH temp1 (d1, d2) AS
(VALUES (DECFLOAT(+1.0), DECFLOAT(+1.00))
       ,(DECFLOAT(-1.0), DECFLOAT(-1.00))
       ,(DECFLOAT(+0.0), DECFLOAT(+0.00))
       ,(DECFLOAT(-0.0), DECFLOAT(-0.00))
       ,(DECFLOAT(+0), DECFLOAT(-0)) )
SELECT TOTALORDER(d1,d2) AS TOTALORDER
FROM temp1;
```

*ANSWER*

| TOTALORDER |
|---|
| 1 |
| -1 |
| 1 |
| 1 |

| TOTALORDER |
| --- |
| 0 |

The NORMALIZE_DECFLOAT scalar function can be used to strip trailing zeros from a DECFLOAT value:

*Remove trailing zeros*

```
WITH temp1 (d1) AS
(VALUES (DECFLOAT(+0 ,16))
       ,(DECFLOAT(+0.0 ,16))
       ,(DECFLOAT(+0.00 ,16))
       ,(DECFLOAT(+0.000 ,16))
)
SELECT d1
     , HEX(d1)                     AS hex_d1
     , NORMALIZE_DECFLOAT(d1)      AS d2
     , HEX(NORMALIZE_DECFLOAT(d1)) AS hex_d2
FROM temp1;
```

*ANSWER*

| D1 | HEX_D1 | D2 | HEX_D2 |
| --- | --- | --- | --- |
| 0 | 0000000000003822 | 0 | 0000000000003822 |
| 0.0 | 0000000000003422 | 0 | 0000000000003822 |
| 0.00 | 0000000000003022 | 0 | 0000000000003822 |
| 0.000 | 0000000000002C22 | 0 | 0000000000003822 |

### DECFLOAT Scalar Functions

The following scalar functions support the DECFLOAT data type:

- **COMPARE_DECFLOAT**: Compares order of two DECFLOAT values.
- **DECFLOAT**: Converts input value to DECFLOAT.
- **NORMALIZE_DECFLOAT**: Removes trailing blanks from DECFLOAT value.
- **QUANTIZE**: Converts number to DECFLOAT, using mask to define precision.
- **TOTALORDER**: Compares order of two DECFLOAT values.

### Date/Time Arithmetic

Manipulating date/time values can sometimes give unexpected results. What follows is a brief introduction to the subject. The basic rules are:

- Multiplication and division is not allowed.
- Subtraction is allowed using date/time values, date/time durations, or labeled durations.

- Addition is allowed using date/time durations, or labeled durations.

The valid labeled durations are listed below:

*Table 2. Labeled Durations and Date/Time Types*

| **LABELED DURATIONS** | | | **WORKS WITH DATE/TIME** | | |
|---|---|---|---|---|---|
| SINGULAR | PLURAL | ITEM FIXED SIZE | DATE | TIME | TIMESTAMP |
| YEAR | YEARS | N | Y | - | Y |
| MONTH | MONTHS | N | Y | - | Y |
| DAY | DAYS | Y | Y | - | Y |
| HOUR | HOURS | Y | - | Y | Y |
| MINUTE | MINUTES | Y | - | Y | Y |
| SECOND | SECONDS | Y | - | Y | Y |
| MICROSECOND | MICROSECONDS | Y | - | Y | Y |

**Usage Notes**

- It doesn’t matter if one uses singular or plural. One can add "4 day" to a date.
- Some months and years are longer than others. So when one adds "2 months" to a date the result is determined, in part, by the date that you began with. More on this below.
- One cannot add "minutes" to a date, or "days" to a time, etc.
- One cannot combine labeled durations in parenthesis: "date - (1 day + 2 months)" will fail. One should instead say: "date - 1 day - 2 months".
- Adding too many hours, minutes or seconds to a time will cause it to wrap around. The overflow will be lost.
- Adding 24 hours to the time '00.00.00' will get '24.00.00'. Adding 24 hours to any other time will return the original value.
- When a decimal value is used (e.g. 4.5 days) the fractional part is discarded. So to add (to a timestamp value) 4.5 days, add 4 days and 12 hours.

Now for some examples:

*Example, Labeled Duration usage*

```
SELECT     sales_date
    ,      sales_date - 10 DAY     AS d1
    ,      sales_date + -1 MONTH   AS d2
    ,      sales_date + 99 YEARS   AS d3
    ,      sales_date + 55 DAYS
                       - 22 MONTHS  AS d4
    ,      sales_date + (4+6) DAYS AS d5
FROM sales
WHERE sales_person = 'GOUNOT'
AND  sales_date = '1995-12-31';
```

*ANSWER*

| sales_date | d1 | d2 | d3 | d4 | d5 |
|---|---|---|---|---|---|
| 1995-12-31 | 1995-12-21 | 1995-11-30 | 2094-12-31 | 1994-04-24 | 1996-01-10 |

Adding or subtracting months or years can give somewhat odd results when the month of the beginning date is longer than the month of the ending date. For example, adding 1 month to '2004-01-31' gives '2004-02-29', which is not the same as adding 31 days, and is not the same result that one will get in 2005. Likewise, adding 1 month, and then a second 1 month to '2004-01-31' gives '2004-03-29', which is not the same as adding 2 months. Below are some examples of this issue:

*Adding Months - Varying Results*

```
SELECT sales_date
     , sales_date + 2 MONTH           AS d1
     , sales_date + 3 MONTHS          AS d2
     , sales_date + 2 MONTH + 1 MONTH AS d3
     , sales_date + (2+1) MONTHS      AS d4
FROM sales
WHERE sales_person = 'GOUNOT'
AND sales_date = '1995-12-31';
```

*ANSWER*

| sales_date | d1 | d2 | d3 | d4 |
|---|---|---|---|---|
| 1995-12-31 | 1996-02-29 | 1996-03-31 | 1996-03-29 | 1996-03-31 |

### Date/Time Duration Usage

When one date/time value is subtracted from another date/time value the result is a date, time,or timestamp duration. This decimal value expresses the difference thus:

*Table 3. Date/Time Durations*

| DURATION-TYPE | FORMAT | NUMBER-REPRESENTS | USE-WITH-D-TYPE |
|---|---|---|---|
| DATE | DECIMAL(8,0) | yyyymmdd | TIMESTAMP, DATE |
| TIME | DECIMAL(6,0) | hhmmss | TIMESTAMP, TIME |
| TIMESTAMP | DECIMAL(20,6) | yyyymmddhhmmss.zzzzzz | TIMESTAMP |

Below is an example of date duration generation:

*Date Duration Generation*

```
SELECT empno
     , hiredate
     , birthdate
     , hiredate - birthdate
FROM employee
WHERE workdept = 'D11'
AND lastname < 'L'
ORDER BY empno;
```

*ANSWER*

| EMPNO | HIREDATE | BIRTHDATE | - |
|---|---|---|---|
| 000150 | 1972-02-12 | 1947-05-17 | 240826 |
| 000200 | 1966-03-03 | 1941-05-29 | 240905 |
| 000210 | 1979-04-11 | 1953-02-23 | 260116 |

A date/time duration can be added to or subtracted from a date/time value, but it does not make for very pretty code:

*Subtracting a Date Duration*

```
SELECT hiredate
     , hiredate - 12345678
     , hiredate - 1234 years
                - 56 months
                - 78 days
FROM employee
WHERE empno = '000150';
```

*ANSWER*

| HIREDATE | - | - |
|---|---|---|
| 1972-02-12 | 0733-03-26 | 0733-03-26 |

### Date/Time Subtraction

One date/time can be subtracted (only) from another valid date/time value. The result is a date/time duration value. Date Duration Generation above has an example.

### Db2 Special Registers

A special register is a Db2 variable that contains information about the state of the system. The complete list follows:

*Table 4. Db2 Special Registers*

| Special Register | Updatable | Data type |
|---|---|---|
| CURRENT CLIENT_ACCTNG | no | VARCHAR(255) |
| CURRENT CLIENT_APPLNAME | no | VARCHAR(255) |
| CURRENT CLIENT_USERID | no | VARCHAR(255) |
| CURRENT CLIENT_WRKSTNNAME | no | VARCHAR(255) |
| CURRENT DATE | no | DATE |
| CURRENT DBPARTITIONNUM | no | INTEGER |
| CURRENT DECFLOAT ROUNDING MODE | no | VARCHAR(128) |
| CURRENT DEFAULT TRANSFORM GROUP | yes | VARCHAR(18) |
| CURRENT DEGREE | yes | CHAR(5) |
| CURRENT EXPLAIN MODE | yes | VARCHAR(254) |
| CURRENT EXPLAIN SNAPSHOT | yes | CHAR(8) |
| CURRENT FEDERATED ASYNCHRONY | yes | INTEGER |
| CURRENT IMPLICIT XMLPARSE OPTION | yes | VARCHAR(19) |
| CURRENT ISOLATION | yes | CHAR(2) |
| CURRENT LOCK TIMEOUT | yes | INTEGER |
| CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION | yes | VARCHAR(254) |
| CURRENT MDC ROLLOUT MODE | yes | VARCHAR(9) |
| CURRENT OPTIMIZATION PROFILE | yes | VARCHAR(261) |
| CURRENT PACKAGE PATH | yes | VARCHAR(4096) |
| CURRENT PATH | yes | VARCHAR(2048) |
| CURRENT QUERY OPTIMIZATION | yes | INTEGER |
| CURRENT REFRESH AGE | yes | DECIMAL(20,6) |
| CURRENT SCHEMA | yes | VARCHAR(128) |
| CURRENT SERVER | no | VARCHAR(128) |
| CURRENT TIME | no | TIME |

| Special Register | Updatable | Data type |
|---|---|---|
| CURRENT TIMESTAMP | no | TIMESTAMP |
| CURRENT TIMEZONE | no | DECIMAL(6,0) |
| CURRENT USER | no | VARCHAR(128) |
| SESSION_USER | yes | VARCHAR(128) |
| SYSTEM_USER | no | VARCHAR(128) |
| USER | yes | VARCHAR(128) |

### Usage Notes

- Some special registers can be referenced using an underscore instead of a blank in the name - as in: CURRENT_DATE.
- Some special registers can be updated using the SET command (see list above).
- All special registers can be queried using the SET command. They can also be referenced in ordinary SQL statements.
- Those special registers that automatically change over time (e.g. current timestamp) are always the same for the duration of a given SQL statement. So if one inserts a thousand rows in a single insert, all will get the same current timestamp.
- One can reference the current timestamp in an insert or update, to record in the target table when the row was changed. To see the value assigned, query the DML statement. See Select DML Changes for details.

Refer to the Db2 SQL Reference Volume 1 for a detailed description of each register.

### Sample SQL

*Using Special Registers*

```
SET CURRENT ISOLATION = RR;
SET CURRENT SCHEMA = 'ABC';
SELECT CURRENT TIME      AS cur_TIME
     , CURRENT ISOLATION AS cur_ISO
     , CURRENT SCHEMA    AS cur_ID
FROM sysibm.sysdummy1;
```

*ANSWER*

| CUR_TIME | CUR_ISO | CUR_ID |
|---|---|---|
| 12:15:16 | RR | ABC |

### Distinct Types

A distinct data type is a field type that is derived from one of the base Db2 field types. It is used

when one wants to prevent users from combining two separate columns that should never be manipulated together (e.g. adding US dollars to Japanese Yen).

The following source type do not support distinct types: XML, Array.

The creation of a distinct type, under the covers, results in the creation of two implied functions that can be used to convert data to and from the source type and the distinct type. Support for the basic comparison operators ( `=, <>, <, < =, >, and > =` ) is also provided. Below is a typical create and drop statement:

*Create and drop distinct type*

```
CREATE DISTINCT TYPE JAP_YEN AS DECIMAL(15,2) WITH COMPARISONS;
DROP DISTINCT TYPE JAP_YEN;
```

A distinct type cannot be dropped if it is currently being used in a table.

**Usage Example**

Imagine that we had the following customer table:

*Sample table, without distinct types*

```
CREATE TABLE customer
( id             INTEGER      NOT NULL
, fname          VARCHAR(10)  NOT NULL WITH DEFAULT ''
, lname          VARCHAR(15)  NOT NULL WITH DEFAULT ''
, date_of_birth  DATE
, citizenship    CHAR(3) NOT NULL WITH DEFAULT ''
, usa_sales      DECIMAL(9,2)
, eur_sales      DECIMAL(9,2)
, sales_office#  SMALLINT
, last_updated   TIMESTAMP
, PRIMARY KEY(id));
```

One problem with the above table is that the user can add the American and European sales values, which if they are expressed in dollars and euros respectively, is silly:

*Silly query, but works*

```
SELECT id
     , usa_sales + eur_sales AS tot_sales
FROM customer;
```

To prevent the above, we can create two distinct types:

*Create Distinct Type examples*

```
CREATE DISTINCT TYPE USA_DOLLARS AS DECIMAL(9,2) WITH COMPARISONS;
CREATE DISTINCT TYPE EUROS       AS DECIMAL(9,2) WITH COMPARISONS;
```

Now we can define the customer table thus:

*Sample table, with distinct types*

```
CREATE TABLE customer
( id            INTEGER     NOT NULL
, fname         VARCHAR(10) NOT NULL WITH DEFAULT ''
, lname         VARCHAR(15) NOT NULL WITH DEFAULT ''
, date_of_birth DATE
, citizenship   CHAR(3)
, usa_sales     USA_DOLLARS
, eur_sales     EUROS
, sales_office# SMALLINT
, last_updated  TIMESTAMP
, PRIMARY KEY(id));
```

Now, when we attempt to run the following, it will fail:

*Silly query, now fails*

```
SELECT id
     , usa_sales + eur_sales AS tot_sales
FROM customer;
```

The creation of a distinct type, under the covers, results in the creation two implied functions that can be used to convert data to and from the source type and the distinct type. In the next example, the two monetary values are converted to their common decimal source type, and then added together:

*Silly query, works again*

```
SELECT id
    , DECIMAL(usa_sales) + DECIMAL(eur_sales) AS tot_sales
FROM customer;
```

#### Fullselect, Subselect, & Common Table Expression

It is not the purpose of this book to give you detailed description of SQL terminology, but there are a few words that you should know. For example, the following diagram illustrates the various components of a query:

```
WITH get_matching_rows AS
     (SELECT id
           , name
           , salary
      FROM staff
      WHERE id < 50
      UNION ALL
      SELECT id
           , name
           , salary
      FROM staff
      WHERE id = 100
     )
     SELECT *
     FROM get_matching_rows
     ORDER BY id
     FETCH FIRST 10 ROWS ONLY
     FOR FETCH ONLY
     WITH UR;
```

*Query components*

- The structure from WITH until the last parenthesis is a COMMON TABLE EXPRESSION.
- Each select block is called a SUBSELECT.
- The block inside the WITH with two SUBSELECTS is a FULLSELECT.

### 2.4.3. Query Components

- **SUBSELECT**: A query that selects zero or more rows from one or more tables.
- **FULLSELECT**: One or more subselects or VALUES clauses, connected using a UNION, INTERSECT, or EXCEPT, all enclosed in parenthesis.
- **COMMON TABLE EXPRESSION**: A named fullselect that can be referenced one more times in another subselect. See Common Table Expression for a more complete definition.

## 2.5. SELECT Statement

A SELECT statement is used to query the database. It has the following components, not all of which need be used in any particular query:

- **SELECT** clause. One of these is required, and it must return at least one item, be it a column, a literal, the result of a function, or something else. One must also access at least one table, be that a true table, a temporary table, a view, an alias or a table function.
- **WITH** clause. This clause is optional. Use this phrase to include independent SELECT statements that are subsequently accessed in a final SELECT (see Common Table Expression).
- **ORDER BY** clause. Optionally, order the final output (see Order By, Group By, and Having).
- **FETCH FIRST** clause. Optionally, stop the query after "n" rows (see FETCH FIRST Clause). If an

optimize-for value is also provided, both values are used independently by the optimizer.

- **READ-ONLY** clause. Optionally, state that the query is read-only. Some queries are inherently read-only, in which case this option has no effect.
- **FOR UPDATE** clause. Optionally, state that the query will be used to update certain columns that are returned during fetch processing.
- **OPTIMIZE FOR n ROWS** clause. Optionally, tell the optimizer to tune the query assuming that not all of the matching rows will be retrieved. If a first-fetch value is also provided, both values are used independently by the optimizer.

Refer to the IBM manuals for a complete description of all of the above. Some of the more interesting options are described below.

#### SELECT Clause

Every query must have at least one SELECT statement, and it must return at least one item, and access at least one object.

### 2.5.1. SELECT Items

- **Column**: A column in one of the table being selected from.
- **Literal**: A literal value (e.g. "ABC"). Use the AS expression to name the literal.
- **Special Register**: A special register (e.g. CURRENT TIME).
- **Expression**: An expression result (e.g. MAX(COL1*10)).
- **Full Select**: An embedded SELECT statement that returns a single row.

### 2.5.2. FROM Objects

- **Table**: Either a permanent or temporary Db2 table.
- **View**: A standard Db2 view.
- **Alias**: A Db2 alias that points to a table, view, or another alias.
- **Full Select**: An embedded SELECT statement that returns a set of rows.
- **Table function**: A kind of function that returns a table.

#### Sample SQL

*Sample SELECT statement*

```
SELECT deptno
     , admrdept
     , 'ABC' AS abc
FROM department
WHERE deptname LIKE '%ING%'
ORDER BY 1;
```

*ANSWER*

| DEPTNO | ADMRDEPT | ABC |
|---|---|---|
| B01 | A00 | ABC |
| D11 | D01 | ABC |

To select all of the columns in a table (or tables) one can use the "*" notation:

*Use "*" to select all columns in table*

```
SELECT *
FROM department
WHERE deptname LIKE '%ING%'
ORDER BY 1;
```

*ANSWER (part of)*

| DEPTNO | etc... |
|---|---|
| B01 | PLANNING |
| D11 | MANUFACTU |

To select both individual columns, and all of the columns (using the "**" notation), in a single SELECT statement, one can still use the "**", but it must fully-qualified using either the object name, or a correlation name:

*Select an individual column, and all columns*

```
SELECT deptno
     , department.*
FROM department
WHERE deptname LIKE '%ING%'
ORDER BY 1;
```

*ANSWER (part of)*

| DEPTNO | DEPTNO | etc... |
|---|---|---|
| B01 | B01 | PLANNING |
| D11 | D11 | MANUFACTU |

Use the following notation to select all the fields in a table twice:

*Select all columns twice*

```
SELECT department.*
     , department.*
FROM department
WHERE eptname LIKE '%NING%'
ORDER BY 1;
```

*ANSWER (part of)*

| DEPTNO | etc... | ... | DEPTNO | etc... | ... |
|---|---|---|---|---|---|
| B01 | PLANNING | ... | B01 | PLANNING | ... |
| D11 | MANUFACTU | ... | D11 | MANUFACTU | ... |

### 2.5.3. FETCH FIRST Clause

The fetch first clause limits the cursor to retrieving "n" rows. If the clause is specified and no number is provided, the query will stop after the first fetch. If this clause is used, and there is no ORDER BY, then the query will simply return a random set of matching rows, where the randomness is a function of the access path used and/or the physical location of the rows in the table:

*FETCH FIRST without ORDER BY, gets random rows*

```
SELECT years
     , name
     , id
FROM staff
FETCH FIRST 3 ROWS ONLY;
```

*ANSWER*

| YEARS | NAME | ID |
|---|---|---|
| 7 | Sanders | 10 |
| 8 | Pernal | 20 |
| 5 | Marenghi | 30 |

Using the FETCH FIRST clause to get the first "n" rows can sometimes return an answer that is not what the user really intended. See below for details.

If an ORDER BY is provided, then the FETCH FIRST clause can be used to stop the query after a certain number of what are, perhaps, the most desirable rows have been returned. However, the phrase should only be used in this manner when the related ORDER BY uniquely identifies each row returned. To illustrate what can go wrong, imagine that we wanted to query the STAFF table in order to get the names of those three employees that have worked for the firm the longest - in order to give them a little reward (or possibly to fire them). The following query could be run:

*FETCH FIRST with ORDER BY, gets wrong answer*

```
SELECT years
     , name
     , id
FROM staff
WHERE years IS NOT NULL
ORDER BY years DESC
FETCH FIRST 3 ROWS ONLY;
```

*ANSWER*

| YEARS | NAME | ID |
|---|---|---|
| 13 | Graham | 310 |
| 12 | Jones | 260 |
| 10 | Hanes | 50 |

The above query answers the question correctly, but the question was wrong, and so the answer is wrong. The problem is that there are two employees that have worked for the firm for ten years, but only one of them shows, and the one that does show was picked at random by the query processor. This is almost certainly not what the business user intended. The next query is similar to the previous, but now the ORDER ID uniquely identifies each row returned (presumably as per the end-user's instructions):

*FETCH FIRST with ORDER BY, gets right answer*

```
SELECT years
     , name
     , id
FROM staff
WHERE years IS NOT NULL
ORDER BY years DESC
       , id DESC
FETCH FIRST 3 ROWS ONLY;
```

*ANSWER*

| YEARS | NAME | ID |
|---|---|---|
| 13 | Graham | 310 |
| 12 | Jones | 260 |
| 10 | Quill | 290 |

Getting the first "n" rows from a query is actually quite a complicated problem. Refer to Selecting "n" or more Rows for a more complete discussion.

### 2.5.4. Correlation Name

The correlation name is defined in the FROM clause and relates to the preceding object name. In some cases, it is used to provide a short form of the related object name. In other situations, it is required in order to uniquely identify logical tables when a single physical table is referred to twice in the same query. Some sample SQL follows:

*Correlation Name usage example*

```
SELECT a.empno
     , a.lastname
, (SELECT MAX(empno)AS empno
   FROM employee) AS b
FROM employee a
WHERE a.empno = b.empno;
```

*ANSWER*

| EMPNO | LASTNAME |
|---|---|
| 000340 | GOUNOT |

*Correlation name usage example*

```
SELECT a.empno
     , a.lastname
     , b.deptno AS dept
FROM employee   a
   , department b
WHERE a.workdept = b.deptno
AND a.job <> 'SALESREP'
AND b.deptname = 'OPERATIONS'
AND a.sex IN ('M','F')
AND b.location IS NULL
ORDER BY 1;
```

*ANSWER*

| EMPNO | LASTNAME | DEPT |
|---|---|---|
| 000090 | HENDERSON | E11 |
| 000280 | SCHNEIDER | E11 |
| 000290 | PARKER | E11 |
| 000300 | SMITH | E11 |
| 000310 | SETRIGHT | E11 |

### 2.5.5. Renaming Fields

The AS phrase can be used in a SELECT list to give a field a different name. If the new name is an invalid field name (e.g. contains embedded blanks), then place the name in quotes:

*Renaming fields using AS*

```
SELECT empno    AS e_num
     , midinit  AS "m int"
     , phoneno  AS "..."
FROM employee
WHERE empno < '000030'
ORDER BY 1;
```

*ANSWER*

| E_NUM | M INT | ... |
|---|---|---|
| 000010 | I | 3978 |
| 000020 | L | 3476 |

The new field name must not be qualified (e.g. A.C1), but need not be unique. Subsequent usage of the new name is limited as follows:

- It can be used in an order by clause.
- It cannot be used in other part of the select (where-clause, group-by, or having).
- It cannot be used in an update clause.
- It is known outside of the fullselect of nested table expressions, common table expressions, and in a view definition.

*View field names defined using AS*

```
CREATE view emp2
AS SELECT empno AS e_num
        , midinit AS "m int"
        , phoneno AS "..."
FROM employee;

SELECT * FROM emp2 WHERE "..." = '3978';
```

*ANSWER*

| E_NUM | M INT | ... |
|---|---|---|
| 000010 | I | 3978 |

### 2.5.6. Working with Nulls

In SQL something can be true, false, or NULL. This three-way logic has to always be considered when accessing data. To illustrate, if we first select all the rows in the STAFF table where the SALARY is < $10,000, then all the rows where the SALARY is >= $10,000, we have not necessarily found all the rows in the table because we have yet to select those rows where the SALARY is null. The presence of null values in a table can also impact the various column functions. For example, the AVG function ignores null values when calculating the average of a set of rows. This means that a user-calculated average may give a different result from a Db2 calculated equivalent:

*AVG of data containing null values*

```
SELECT AVG(comm)            AS a1
     , SUM(comm) / COUNT(*) AS a2
FROM staff
WHERE id < 100;
```

*ANSWER*

| A1 | A2 |
|---|---|
| 796.025 | 530.68 |

Null values can also pop in columns that are defined as NOT NULL. This happens when a field is processed using a column function and there are no rows that match the search criteria:

*Getting a NULL value from a field defined NOT NULL*

```
SELECT COUNT(*)      AS num
     , MAX(lastname) AS max
FROM employee
WHERE firstnme = 'FRED';
```

ANSWER

| NUM | MAX |
|---|---|
| 0 | - |

#### Why Null Exist

NULL values can represent two kinds of data. In first case, the value is unknown (e.g. we do not know the name of the person's spouse). Alternatively, the value is not relevant to the situation (e.g. the person does not have a spouse). Many people prefer not to have to bother with nulls, so they use instead a special value when necessary (e.g. an unknown employee name is blank). This trick works OK with character data, but it can lead to problems when used on numeric values (e.g. an unknown salary is set to zero).

#### Locating Null Values

One can not use an equal predicate to locate those values that are null because a null value does not actually equal anything, not even null, it is simply null. The IS NULL or IS NOT NULL phrases are used instead. The following example gets the average commission of only those rows that are not null. Note that the second result differs from the first due to rounding loss.

*AVG of those rows that are not null*

```
SELECT AVG(comm)            AS a1
     , SUM(comm) / COUNT(*) AS a2
FROM staff
WHERE id < 100
AND comm IS NOT NULL;
```

*ANSWER*

| A1 | A2 |
|---|---|
| 796.025 | 796.02 |

### 2.5.7. Quotes and Double-quotes

To write a string, put it in quotes. If the string contains quotes, each quote is represented by a pair of quotes:

*Quote usage*

```
SELECT 'JOHN'        AS J1
     , 'JOHN''S'     AS J2
     , '''JOHN''S''' AS J3
     , '"JOHN''S"'   AS J4
FROM staff
WHERE id = 10;
```

*ANSWER*

| J1 | J2 | J3 | J4 |
|---|---|---|---|
| JOHN | JOHN'S | 'JOHN'S' | "JOHN'S" |

Double quotes can be used to give a name to an output field that would otherwise not be valid. To put a double quote in the name, use a pair of quotes:

*Double-quote usage*

```
SELECT id    AS "USER ID"
     , dept  AS "D#"
     , years AS "#Y"
     , 'ABC' AS "'TXT'"
     , '"'   AS """quote"" fld"
FROM staff s
WHERE id < 40
ORDER BY "USER ID";
```

*ANSWER*

| USER ID | D# | #Y | 'TXT' | "quote" fld |
|---|---|---|---|---|
| 10 | 20 | 7 | ABC | " |
| 20 | 20 | 8 | ABC | " |
| 30 | 38 | 5 | ABC | " |

# 2.6. SQL Predicates

A predicate is used in either the WHERE or HAVING clauses of a SQL statement. It specifies a condition that true, false, or unknown about a row or a group.

## 2.6.1. Predicate Precedence

As a rule, a query will return the same result regardless of the sequence in which the various predicates are specified. However, note the following:

- Predicates separated by an OR may need parenthesis - see AND/OR Precedence.
- Checks specified in a CASE statement are done in the order written - see CASE Expression.

### Basic Predicate

A basic predicate compares two values. If either value is null, the result is unknown. Otherwise the result is either true or false.

*Basic Predicate examples*

```
SELECT id, job, dept
FROM staff
WHERE job = 'Mgr'
AND NOT job <> 'Mgr'
AND NOT job = 'Sales'
AND id <> 100
AND id >= 0
AND id <= 150
AND NOT dept = 50
ORDER BY id;
```

*ANSWER*

| ID | JOB | DEPT |
|---|---|---|
| 10 | Mgr | 20 |
| 30 | Mgr | 38 |
| 50 | Mgr | 15 |
| 140 | Mgr | 51 |

A variation of this predicate type can be used to compare sets of columns/values. Everything on both sides must equal in order for the expressions to match:

*Basic Predicate example, multi-value check*

```
SELECT id, dept, job
FROM staff
WHERE (id,dept) = (30,28)
OR (id,years) = (90, 7)
OR (dept,job) = (38,'Mgr')
ORDER BY 1;
```

*ANSWER*

| ID | DEPT | JOB |
|---|---|---|
| 30 | 38 | Mgr |

Below is the same query written the old fashioned way:

*Same query as prior, using individual predicates*

```
SELECT id, dept, job
FROM staff
WHERE (id   = 30 AND dept = 28)
OR    (id   = 90 AND years = 7)
OR    (dept = 38 AND job = 'Mgr')
ORDER BY 1;
```

ANSWER

| ID | DEPT | JOB |
|---|---|---|
| 30 | 38 | Mgr |

### 2.6.2. Quantified Predicate

A quantified predicate compares one or more values with a collection of values.

*Quantified Predicate example, two single-value sub-queries*

```
SELECT id, job
FROM staff
WHERE job = ANY (SELECT job FROM staff)
AND id <= ALL (SELECT id FROM staff)
ORDER BY id;
```

*ANSWER*

| ID | JOB |
|---|---|
| 10 | Mgr |

*Quantified Predicate example, multi-value sub-query*

```
SELECT id, dept, job
FROM staff
WHERE (id,dept) = ANY
(SELECT dept, id
 FROM staff
)
ORDER BY 1;
```

*ANSWER*

| ID | DEPT | JOB |
|---|---|---|
| 20 | 20 | Sales |

See the sub-query chapter on Sub-Query for more data on this predicate type.

### 2.6.3. BETWEEN Predicate

The BETWEEN predicate compares a value within a range of values.

The between check always assumes that the first value in the expression is the low value and the second value is the high value. For example, BETWEEN 10 AND 12 may find data, but BETWEEN 12 AND 10 never will.

*BETWEEN Predicate examples*

```
SELECT id, job
FROM staff
WHERE   id     BETWEEN 10 AND 30
AND     id NOT BETWEEN 30 AND 10
AND NOT id NOT BETWEEN 10 AND 30
ORDER BY id;
```

*ANSWER*

| ID | JOB |
|---|---|
| 10 | Mgr |
| 20 | Sales |
| 30 | Mgr |

### 2.6.4. EXISTS Predicate

An EXISTS predicate tests for the existence of matching rows.

*EXISTS Predicate example*

```
SELECT id, job
FROM staff a
WHERE EXISTS
(SELECT *
 FROM staff b
 WHERE b.id = a.id
 AND b.id < 50
)
ORDER BY id;
```

*ANSWER*

| ID | JOB |
|---|---|
| 10 | Mgr |
| 20 | Sales |
| 30 | Mgr |

| ID | JOB |
|---|---|
| 40 | Sales |

See the sub-query chapter on Sub-Query for more data on this predicate type.

### 2.6.5. IN Predicate

The IN predicate compares one or more values with a list of values.

The list of values being compared in the IN statement can either be a set of in-line expressions (e.g. ID in (10,20,30)), or a set rows returned from a sub-query. Either way, Db2 simply goes through the list until it finds a match.

*IN Predicate examples, single values*

```
SELECT id, job
FROM staff a
WHERE id IN (10,20,30)
AND id IN
    (SELECT id
     FROM staff
    )
AND id NOT IN 99
ORDER BY id;
```

*ANSWER*

| ID | JOB |
|---|---|
| 10 | Mgr |
| 20 | Sales |
| 30 | Mgr |

The IN statement can also be used to compare multiple fields against a set of rows returned from a sub-query. A match exists when all fields equal. This type of statement is especially useful when doing a search against a table with a multi-columns key.

Be careful when using the NOT IN expression against a sub-query result. If any one row in the sub-query returns null, the result will be no match. See Sub-Query for more details.

*IN Predicate example, multi-value*

```
SELECT empno, lastname
FROM employee
WHERE (empno, 'AD3113') IN
    (SELECT empno, projno
     FROM emp_act
     WHERE emptime > 0.5
    )
ORDER BY 1;
```

*ANSWER*

| EMPNO | LASTNAME |
|---|---|
| 000260 | JOHNSON |
| 000270 | PEREZ |

See the sub-query chapter on Sub-Query for more data on this statement type.

### 2.6.6. LIKE Predicate

The LIKE predicate does partial checks on character strings.

The percent and underscore characters have special meanings. The first means skip a string of any length (including zero) and the second means skip one byte. For example:

- LIKE 'AB_D%' Finds 'ABCD' and 'ABCDE', but not 'ABD', nor 'ABCCD'.
- LIKE '_X' Finds 'XX' and 'DX', but not 'X', nor 'ABX', nor 'AXB'.
- LIKE '%X' Finds 'AX', 'X', and 'AAX', but not 'XA'.

*LIKE Predicate examples*

```
SELECT id
     , name
FROM staff
WHERE name LIKE 'S%n'
   OR name LIKE '_a_a%'
   OR name LIKE '%r_%a'
ORDER BY id;
```

*ANSWER*

| ID | NAME |
|---|---|
| 130 | Yamaguchi |
| 200 | Scoutten |

### The ESCAPE Phrase

The escape character in a LIKE statement enables one to check for percent signs and/or underscores in the search string. When used, it precedes the '%' or '_' in the search string indicating that it is the actual value and not the special character which is to be checked for. When processing the LIKE pattern, Db2 works thus: Any pair of escape characters is treated as the literal value (e.g. "+" means the string ""). Any single occurrence of an escape character followed by either a "%" or a "_" means the literal "%" or "_" (e.g. "+%" means the string "%"). Any other "%" or "_" is used as in a normal LIKE pattern.

*Table 5. LIKE and ESCAPE examples*

| LIKE STATEMENT TEXT | WHAT VALUES MATCH |
|---|---|
| LIKE 'AB%' | Finds AB, any string |
| LIKE 'AB%' ESCAPE '+' | Finds AB, any string |
| LIKE 'AB+%' ESCAPE '+' | Finds AB% |
| LIKE 'AB+' ESCAPE '' | Finds AB+ |
| LIKE 'AB+%%' ESCAPE '+' | Finds AB%, any string |
| LIKE 'AB+%' ESCAPE '' | Finds AB+, any string |
| LIKE 'AB++%' ESCAPE '' | Finds AB+% |
| LIKE 'AB++%%' ESCAPE '' | Finds AB+%, any string |
| LIKE 'AB+%%%' ESCAPE '' | Finds AB%%, any string |
| LIKE 'AB' ESCAPE '+' | Finds AB++ |
| LIKE 'AB%' ESCAPE '' | Finds AB++% |
| LIKE 'AB%' ESCAPE '+' | Finds AB++, any string |
| LIKE 'AB+%+%' ESCAPE '' | Finds AB%+, any string |

Now for sample SQL:

*LIKE and ESCAPE examples*

```
SELECT id
FROM staff
WHERE id = 10
AND 'ABC' LIKE 'AB%'
AND 'A%C' LIKE 'A/%C' ESCAPE '/'
AND 'A_C' LIKE 'A\_C' ESCAPE '\'
AND 'A_$' LIKE 'A$_$$' ESCAPE '$';
```

*ANSWER*

| ID |
|---|
| 10 |

### 2.6.7. LIKE_COLUMN Function

The LIKE predicate cannot be used to compare one column against another. One may need to do this when joining structured to unstructured data. For example, imagine that one had a list of SQL statements (in a table) and a list of view names in a second table. One might want to scan the SQL text (using a LIKE predicate) to find those statements that referenced the views. The LOCATE function can be used to do a simple equality check. The LIKE predicate allows a more sophisticated search. The following code creates a scalar function and dependent procedure that can compare one column against another (by converting both column values into input variables). The function is just a stub. It passes the two input values down to the procedure where they are compared using a LIKE predicate. If there is a match, the function returns one, else zero.

These examples use an "!" as the stmt delimiter.

*Create LIKE_COLUMN procedure*

```
--#SET DELIMITER !
CREATE PROCEDURE LIKE_COLUMN
 ( IN instr1 VARCHAR(4000)
 , IN instr2 VARCHAR(4000)
 , OUT outval SMALLINT)
LANGUAGE SQL
CONTAINS SQL
DETERMINISTIC
NO EXTERNAL ACTION
BEGIN
    SET outval =
        CASE
            WHEN instr1 LIKE instr2
                THEN 1
            ELSE 0
        END;
    RETURN;
END!
```

*Create LIKE_COLUMN function*

```
CREATE FUNCTION LIKE_COLUMN
( instr1 VARCHAR(4000)
, instr2 VARCHAR(4000))
RETURNS SMALLINT
LANGUAGE SQL
CONTAINS SQL
DETERMINISTIC
NO EXTERNAL ACTION
BEGIN ATOMIC
    DECLARE outval SMALLINT;
    CALL LIKE_COLUMN(instr1, instr2, outval);
    RETURN outval;
END!
```

Below is an example of the above function being used to compare to the contents of one column against another:

*Use LIKE_COLUMN function*

```
WITH temp1 (jtest) AS
(VALUES ('_gr%')
      , ('S_le%')
)
SELECT
  s.id
, s.name
, s.job
, t.jtest
FROM staff s
   , temp1 t
WHERE LIKE_COLUMN(s.job , t.jtest) = 1
AND s.id < 70
ORDER BY s.id;
```

*ANSWER*

| ID | NAME | JOB | JTEST |
|---|---|---|---|
| 10 | Sanders | Mgr | _gr% |
| 20 | Pernal | Sales | S_le% |
| 30 | Marenghi | Mgr | _gr% |
| 40 | O'Brien | Sales | S_le% |
| 50 | Hanes | Mgr | _gr% |
| 60 | Quigley | Sales | S_le% |

### 2.6.8. NULL Predicate

The NULL predicate checks for null values. The result of this predicate cannot be unknown. If the value of the expression is null, the result is true. If the value of the expression is not null, the result is false.

*NULL predicate examples*

```
SELECT id, comm
FROM staff
WHERE id < 100
AND id IS NOT NULL
AND comm IS NULL
AND NOT comm IS NOT NULL
ORDER BY id;
```

ANSWER

| ID | COMM |
|---|---|
| 10 | - |
| 30 | - |
| 50 | - |

Use the COALESCE function to convert null values into something else.

### 2.6.9. Special Character Usage

To refer to a special character in a predicate, or anywhere else in a SQL statement, use the "X" notation to substitute with the ASCII hex value. For example, the following query will list all names in the STAFF table that have an "a" followed by a semicolon:

*Refer to semi-colon in SQL text*

```
SELECT id
     , name
FROM staff
WHERE name LIKE '%a' || X'3B' || '%'
ORDER BY id;
```

### 2.6.10. Precedence Rules

Expressions within parentheses are done first, then prefix operators (e.g. -1), then multiplication and division, then addition and subtraction. When two operations of equal precedence are together (e.g. 1 * 5 / 4) they are done from left to right.

*Precedence rules example*

```
Example:

555 +     -22  /  (12 - 3) * 66
    ^     ^    ^      ^    ^
    5th  2nd  3rd    1st  4th

ANSWER: 423
```

Be aware that the result that you get depends very much on whether you are doing integer or decimal arithmetic. Below is the above done using integer numbers:

*Precedence rules, integer example*

```
SELECT              (12 - 3)      AS int1
       ,       -22 / (12 - 3)      AS int2
       ,       -22 / (12 - 3) * 66 AS int3
       , 555 + -22 / (12 - 3) * 66 AS int4
FROM sysibm.sysdummy1;
```

*ANSWER*

| INT1 | INT2 | INT3 | INT4 |
|---|---|---|---|
| 9 | -2 | -132 | 423 |

Db2 truncates, not rounds, when doing integer arithmetic.

Here is the same done using decimal numbers:

*Precedence rules, decimal example*

```
SELECT              (12.0 - 3)      AS dec1
       ,       -22 / (12.0 - 3)      AS dec2
       ,       -22 / (12.0 - 3) * 66 AS dec3
       , 555 + -22 / (12.0 - 3) * 66 AS dec4
FROM sysibm.sysdummy1;
```

*ANSWER*

| DEC1 | DEC2 | DEC3 | DEC4 |
|---|---|---|---|
| 9.0 | -2.4 | -161.3 | 393.6 |

## 2.6.11. AND/OR Precedence

AND operations are done before OR operations. This means that one side of an OR is fully processed before the other side is begun. To illustrate:

**TABLE1**

| col1 | col2 |
|---|---|
| A | AA |
| B | BB |
| C | CC |

```
SELECT *
FROM table1
WHERE col1 =  'C'
AND   col1 >= 'A'
OR    col2 >= 'AA'
ORDER BY col1;
```

*ANSWER*

| COL1 | COL2 |
|---|---|
| A | AA |
| B | BB |
| C | CC |

```
SELECT *
FROM table1
WHERE (col1 = 'C'
AND    col1 >= 'A')
OR     col2 >= 'AA'
ORDER BY col1;
```

*ANSWER*

| COL1 | COL2 |
|---|---|
| A | AA |
| B | BB |
| C | CC |

*Use of OR and parenthesis*

```
SELECT *
FROM table1
WHERE col1 = 'C'
AND  (col1 >= 'A'
OR    col2 >= 'AA')
ORDER BY col1;
```

*ANSWER*

| COL1 | COL2 |
|---|---|
| C | CC |

The omission of necessary parenthesis surrounding OR operators is a very common mistake. The result is usually the wrong answer. One symptom of this problem is that many more rows are returned (or updated) than anticipated.

### 2.6.12. Processing Sequence

The various parts of a SQL statement are always executed in a specific sequence in order to avoid semantic ambiguity:

- FROM clause.
- JOIN ON clause.
- WHERE clause.
- GROUP BY and aggregate.
- HAVING clause.
- SELECT list.
- ORDER BY clause.
- FETCH FIRST.

Observe that ON predicates (e.g. in an outer join) are always processed before any WHERE predicates (in the same join) are applied. Ignoring this processing sequence can cause what looks like an outer join to run as an inner join (see ON and WHERE Usage). Likewise, a function that is referenced in the SELECT section of a query (e.g. row-number) is applied after the set of matching rows has been identified, but before the data has been ordered.

## 2.7. CAST Expression

The CAST expression is used to convert one data type to another. It is similar to the various field-type functions (e.g. CHAR, SMALLINT) except that it can also handle null values and host-variable parameter markers.

### 2.7.1. Input vs. Output Rules

- **EXPRESSION**: If the input is neither null, nor a parameter marker, the input data-type is converted to the output data-type. Truncation and/or padding with blanks occur as required. An error is generated if the conversion is illegal.
- **NULL**: If the input is null, the output is a null value of the specified type.
- **PARAMETER MAKER**: This option is only used in programs and need not concern us here. See the Db2 SQL Reference for details.

**Examples**

Use the CAST expression to convert the SALARY field from decimal to integer:

*Use CAST expression to convert Decimal to Integer*

```
SELECT id
     , salary
     , CAST(salary AS INTEGER) AS sal2
FROM staff
WHERE id < 30
ORDER BY id;
```

*ANSWER*

| ID | SALARY | SAL2 |
|---|---|---|
| 10 | 98357.50 | 98357 |
| 20 | 78171.25 | 78171 |

Use the CAST expression to truncate the JOB field. A warning message will be generated for the second line of output because non-blank truncation is being done.

*Use CAST expression to truncate Char field*

```
SELECT id
     , job
     , CAST(job AS CHAR(3)) AS job2
FROM staff
WHERE id < 30
ORDER BY id;
```

*ANSWER*

| ID | JOB | JOB2 |
|---|---|---|
| 10 | Mgr | Mgr |
| 20 | Sales | Sal |

Use the CAST expression to make a derived field called JUNK of type SMALLINT where all of the values are null.

*Use CAST expression to define SMALLINT field with null values*

```
SELECT id
     , CAST(NULL AS SMALLINT) AS junk
FROM staff
WHERE id < 30
ORDER BY id;
```

*ANSWER*

| ID | JUNK |
|---|---|
| 10 | - |
| 20 | - |

The CAST expression can also be used in a join, where the field types being matched differ:

*CAST expression in join*

```
SELECT stf.id
     , emp.empno
FROM staff stf
LEFT OUTER JOIN employee emp
ON stf.id = CAST(emp.empno AS INTEGER)
AND emp.job = 'MANAGER'
WHERE stf.id < 60
ORDER BY stf.id;
```

*ANSWER*

| ID | EMPNO |
|---|---|
| 10 | - |
| 20 | 000020 |
| 30 | 000030 |
| 40 | - |
| 50 | 000050 |

Of course, the same join can be written using the raw function:

*Function usage in join*

```
SELECT stf.id
     , emp.empno
FROM staff stf
LEFT OUTER JOIN employee emp
ON stf.id = INTEGER(emp.empno)
AND emp.job = 'MANAGER'
WHERE stf.id < 60
ORDER BY stf.id;
```

*ANSWER*

| ID | EMPNO |
|---|---|
| 10 | - |

| ID | EMPNO |
|---|---|
| 20 | 000020 |
| 30 | 000030 |
| 40 | - |
| 50 | 000050 |

# 2.8. VALUES Statement

The VALUES clause is used to define a set of rows and columns with explicit values. The clause is commonly used in temporary tables, but can also be used in view definitions. Once defined in a table or view, the output of the VALUES clause can be grouped by, joined to, and otherwise used as if it is an ordinary table - except that it can not be updated.

Each column defined is separated from the next using a comma. Multiple rows (which may also contain multiple columns) are separated from each other using parenthesis and a comma. When multiple rows are specified, all must share a common data type. Some examples follow:

*VALUES usage examples*

```
VALUES 6                        <== 1 row, 1 column
VALUES(6)                       <== 1 row, 1 column
VALUES 6, 7, 8                  <== 1 row, 3 columns
VALUES (6), (7), (8)            <== 3 rows, 1 column
VALUES (6,66), (7,77), (8,NULL) <== 3 rows, 2 column
```

## 2.8.1. Sample SQL

The VALUES clause can be used by itself as a very primitive substitute for the SELECT statement. One key difference is that output columns cannot be named. But they can be ordered, and fetched, and even named externally, as the next example illustrates:

#### PLAIN VALUES

*Logically equivalent VALUES statements*

```
VALUES
  (1,2)
, (2,3)
, (3,4)
ORDER BY 2 DESC;
```

#### VALUES + WITH

```
WITH temp (c1,c2) AS
( VALUES (1,2)
       , (2,3)
       , (3,4)
)
SELECT  *
FROM temp
ORDER BY 2 DESC;
```

**VALUES + SELECT**

```
SELECT *
FROM (VALUES (1,2)
           , (2,3)
           , (3,4)
) temp (c1,c2)
ORDER BY 2 DESC;
```

*ANSWER*

| 1 | 2 |
|---|---|
| 3 | 4 |
| 2 | 3 |
| 1 | 2 |

The VALUES clause can encapsulate several independent queries:

*VALUES running selects*

```
VALUES
(
  (SELECT COUNT(*) FROM employee)
, (SELECT AVG(salary) FROM staff)
, (SELECT MAX(deptno) FROM department)
)
FOR FETCH ONLY
WITH UR;
```

*ANSWER*

| 1 | 2 | 3 |
|---|---|---|
| 42 | 67932.78 | J22 |

The next statement defines a temporary table containing two columns and three rows. The first column defaults to type integer and the second to type varchar.

*Use VALUES to define a temporary table (1 of 4)*

```
WITH temp1 (col1, col2) AS
(VALUES
  (0, 'AA')
, (1, 'BB')
, (2, NULL)
)
SELECT *
FROM temp1;
```

*ANSWER*

| COL1 | COL2 |
|---|---|
| 0 | AA |
| 1 | BB |
| 2 | - |

If we wish to explicitly control the output field types we can define them using the appropriate function. This trick does not work if even a single value in the target column is null.

*Use VALUES to define a temporary table (2 of 4)*

```
WITH temp1 (col1, col2) AS
(VALUES
  (DECIMAL(0 ,3, 1), 'AA')
, (DECIMAL(1 ,3, 1), 'BB')
, (DECIMAL(2 ,3, 1), NULL)
)
SELECT *
FROM temp1;
```

*ANSWER*

| COL1 | COL2 |
|---|---|
| 0.0 | AA |
| 1.0 | BB |
| 2.0 | - |

If any one of the values in the column that we wish to explicitly define has a null value, we have to use the CAST expression to set the output field type:

*Use VALUES to define a temporary table (3 of 4)*

```
WITH temp1 (col1,col2) AS
(VALUES
  (0, CAST('AA' AS CHAR(1)))
, (1, CAST('BB' AS CHAR(1)))
, (2, CAST(NULL AS CHAR(1)))
)
SELECT *
FROM temp1;
```

*ANSWER*

| COL1 | COL2 |
|---|---|
| 0 | A |
| 1 | B |
| 2 | - |

Alternatively, we can set the output type for all of the not-null rows in the column. Db2 will then use these rows as a guide for defining the whole column:

*Use VALUES to define a temporary table (4 of 4)*

```
WITH temp1 (col1,col2) AS
(VALUES
  (0, CHAR('AA', 1))
, (1, CHAR('BB', 1))
, (2, NULL)
)
SELECT *
FROM temp1;
```

*ANSWER*

| COL1 | COL2 |
|---|---|
| 0 | A |
| 1 | B |
| 2 | - |

### 2.8.2. More Sample SQL

Temporary tables, or (permanent) views, defined using the VALUES expression can be used much like a Db2 table. They can be joined, unioned, and selected from. They can not, however, be updated, or have indexes defined on them. Temporary tables can not be used in a sub-query.

*Derive one temporary table from another*

```
WITH temp1 (col1, col2,col3) AS
(VALUES
  (0, 'AA', 0.00)
, (1, 'BB', 1.11)
, (2, 'CC', 2.22)
)
, temp2 (col1b, colx) AS
(SELECT col1
      , col1 + col3
FROM temp1
)
SELECT *
FROM temp2;
```

*ANSWER*

| COL1B | COLX |
|---|---|
| 0 | 0.00 |
| 1 | 2.11 |
| 2 | 4.22 |

*Define a view using a VALUES clause*

```
CREATE VIEW silly (c1, c2, c3)
AS VALUES
  (11, 'AAA', SMALLINT(22))
, (12, 'BBB', SMALLINT(33))
, (13, 'CCC', NULL);
COMMIT;
```

*Use VALUES defined data to seed a recursive SQL statement*

```
WITH temp1 (col1) AS
(VALUES 0
 UNION ALL
 SELECT col1 + 1
 FROM temp1
 WHERE col1 + 1 < 100
)
SELECT *
FROM temp1;
```

*ANSWER*

| COL1 |
|---|
| 0 |
| 1 |
| 2 |
| 3 |
| etc |

All of the above examples have matched a VALUES statement up with a prior WITH expression, so as to name the generated columns. One doesn't have to use the latter, but if you don't, you get a table with unnamed columns, which is pretty useless:

*Generate table with unnamed columns*

```
SELECT *
FROM
(VALUES
       (123, 'ABC')
     , (234, 'DEF')
) AS ttt
ORDER BY 1 DESC;
```

*ANSWER*

| - | - |
|---|---|
| 234 | DEF |
| 123 | ABC |

**Combine Columns**

The VALUES statement can be used inside a TABLE function to combine separate columns into one. In the following example, three columns in the STAFF table are combined into a single column – with one row per item:

*Combine columns example*

```
SELECT id
, salary AS sal
, comm   AS com
, combo
, typ
FROM staff
, TABLE( VALUES(salary , 'SAL')
              , (comm   , 'COM')
 ) AS tab(combo, typ)
WHERE id < 40
ORDER BY id
       , typ;
```

*ANSWER*

| ID | SAL | COM | COMBO | TYP |
|---|---|---|---|---|
| 10 | 98357.50 | - | | COM |
| 10 | 98357.50 | - | 98357.50 | SAL |
| 20 | 78171.25 | 612.45 | 612.45 | COM |
| 20 | 78171.25 | 612.45 | 78171.25 | SAL |
| 30 | 77506.75 | - | | COM |
| 30 | 77506.75 | - | 77506.75 | SAL |

The above query works as follows:

- The set of matching rows are obtained from the STAFF table.
- For each matching row, the TABLE function creates two rows, the first with the salary value, and the second with the commission.
- Each new row as gets a second literal column – indicating the data source.
- Finally, the "AS" expression assigns a correlation name to the table output, and also defines two column names.

The TABLE function is resolved row-by-row, with the result being joined to the current row in the STAFF table. This explains why we do not get a Cartesian product, even though no join criteria are provided.

The keyword LATERAL can be used instead of TABLE in the above query.

## 2.9. CASE Expression

CASE expressions enable one to do if-then-else type processing inside of SQL statements.

The sequence of the CASE conditions can affect the answer. The first WHEN check that matches is the one used.

### 2.9.1. CASE Syntax Styles

There are two general flavors of the CASE expression. In the first kind, each WHEN statement does its own independent check. In the second kind, all of the WHEN conditions do similar "equal" checks against a common reference expression.

*Use CASE (1st type) to expand a value*

```
SELECT Lastname
, sex AS sx
, CASE sex
   WHEN 'F' THEN 'FEMALE'
   WHEN 'M' THEN 'MALE'
   ELSE NULL
  END AS sexx
FROM employee
WHERE lastname LIKE 'J%'
ORDER BY 1;
```

*ANSWER*

| LASTNAME | SX | SEXX |
|---|---|---|
| JEFFERSON | M | MALE |
| JOHN | F | FEMALE |
| JOHNSON | F | FEMALE |
| JONES | M | MALE |

*Use CASE (2nd type) to expand a value*

```
SELECT lastname
     , sex AS sx
     , CASE WHEN sex = 'F' THEN 'FEMALE'
            WHEN sex = 'M' THEN 'MALE'
            ELSE NULL
       END AS sexx
FROM employee
WHERE lastname LIKE 'J%'
ORDER BY 1;
```

*ANSWER*

| LASTNAME | SX | SEXX |
|---|---|---|
| JEFFERSON | M | MALE |

| LASTNAME | SX | SEXX |
|---|---|---|
| JOHN | F | FEMALE |
| JOHNSON | F | FEMALE |
| JONES | M | MALE |

**Notes & Restrictions**

- If more than one WHEN condition is true, the first one processed that matches is used.
- If no WHEN matches, the value in the ELSE clause applies. If no WHEN matches and there is no ELSE clause, the result is NULL.
- There must be at least one non-null result in a CASE statement. Failing that, one of the NULL results must be inside of a CAST expression.
- All result values must be of the same type.
- Functions that have an external action (e.g. RAND) can not be used in the expression part of a CASE statement.

### 2.9.2. Sample SQL

*Use CASE to display the higher of two values*

```
SELECT lastname
     , midinit AS mi
     , sex AS sx
     , CASE WHEN midinit > SEX THEN midinit
            ELSE sex
       END AS mx
FROM employee
WHERE lastname LIKE 'J%'
ORDER BY 1;
```

*ANSWER*

| LASTNAME | MI | SX | MX |
|---|---|---|---|
| JEFFERSON | J | M | M |
| JOHN | K | K | K |
| JOHNSON | P | F | P |
| JONES | T | M | T |

*Use CASE to get multiple counts in one pass*

```
SELECT COUNT(*) AS tot
     , SUM(CASE sex WHEN 'F' THEN 1 ELSE 0 END) AS #f
     , SUM(CASE sex WHEN 'M' THEN 1 ELSE 0 END) AS #m
FROM employee
WHERE lastname LIKE 'J%';
```

*ANSWER*

| TOT | #F | #M |
|---|---|---|
| 4 | 2 | 2 |

*Use CASE inside a function*

```
SELECT lastname
     , LENGTH(RTRIM(lastname)) AS len
     , SUBSTR(lastname , 1 ,
                CASE WHEN LENGTH(RTRIM(lastname)) > 6 THEN 6
                ELSE LENGTH(RTRIM(lastname))
                END
              ) AS lastnm
FROM employee
WHERE lastname LIKE 'J%'
ORDER BY 1;
```

*ANSWER*

| LASTNAME | LEN | LASTNM |
|---|---|---|
| JEFFERSON | 9 | JEFFER |
| JOHN | 4 | JOHN |
| JOHNSON | 7 | JOHNSO |
| JONES | 5 | JONES |

The CASE expression can also be used in an UPDATE statement to do any one of several alternative updates to a particular field in a single pass of the data:

*UPDATE statement with nested CASE expressions*

```
UPDATE staff
SET comm =
CASE dept
    WHEN 15 THEN comm * 1.1
    WHEN 20 THEN comm * 1.2
    WHEN 38 THEN
        CASE
            WHEN years < 5 THEN comm * 1.3
            WHEN years >= 5 THEN comm * 1.4
            ELSE NULL
        END
    ELSE comm
END
WHERE comm IS NOT NULL
AND dept < 50;
```

In the next example a CASE expression is used to avoid a divide-by-zero error:

*Use CASE to avoid divide by zero*

```
WITH temp1 (c1, c2) AS
(VALUES
(88, 9),(44, 3),(22, 0),(0, 1))
SELECT c1
     , c2
     , CASE c2
         WHEN 0 THEN NULL
         ELSE c1/c2
       END AS c3
FROM temp1;
```

*ANSWER*

| C1 | C2 | C3 |
|---|---|---|
| 88 | 9 | 9 |
| 44 | 3 | 14 |
| 22 | 0 | 0 |
| 1 | 0 | - |

At least one of the results in a CASE expression must be a value (i.e. not null). This is so that Db2 will know what output type to make the result.

### 2.9.3. Problematic CASE Statements

The case WHEN checks are always processed in the order that they are found. The first one that

matches is the one used. This means that the answer returned by the query can be affected by the sequence on the WHEN checks. To illustrate this, the next statement uses the SEX field (which is always either "F" or "M") to create a new field called SXX. In this particular example, the SQL works as intended.

*Use CASE to derive a value (correct)*

```
SELECT lastname
     , sex
     , CASE
        WHEN sex >= 'M' THEN 'MAL'
        WHEN sex >= 'F' THEN 'FEM'
       END AS sxx
FROM employee
WHERE lastname LIKE 'J%'
ORDER BY 1;
```

*ANSWER*

| LASTNAME | SX | SXX |
|---|---|---|
| JEFFERSON | M | MAL |
| JOHN | F | FEM |
| JOHNSON | F | FEM |
| JONES | M | MAL |

In the example below all of the values in SXX field are "FEM". This is not the same as what happened above, yet the only difference is in the order of the CASE checks.

*Use CASE to derive a value (incorrect)*

```
SELECT lastname
     , sex
     , CASE
        WHEN sex >= 'F' THEN 'FEM'
        WHEN sex >= 'M' THEN 'MAL'
       END AS sxx
FROM employee
WHERE lastname LIKE 'J%'
ORDER BY 1;
```

*ANSWER*

| LASTNAME | SX | SXX |
|---|---|---|
| JEFFERSON | M | FEM |
| JOHN | F | FEM |
| JOHNSON | F | FEM |

| LASTNAME | SX | SXX |
|---|---|---|
| JONES | M | FEM |

In the prior statement the two WHEN checks overlap each other in terms of the values that they include. Because the first check includes all values that also match the second, the latter never gets invoked. Note that this problem can not occur when all of the WHEN expressions are equality checks.

### 2.9.4. CASE in Predicate

The result of a CASE expression can be referenced in a predicate:

*Use CASE in a predicate*

```
SELECT id
     , dept
     , salary
     , comm
FROM staff
WHERE CASE
        WHEN comm < 70                    THEN 'A'
        WHEN name LIKE 'W%'               THEN 'B'
        WHEN salary < 11000               THEN 'C'
        WHEN salary < 18500 AND dept <> 33  THEN 'D'
        WHEN salary < 19000               THEN 'E'
END IN ('A','C','E')
ORDER BY id;
```

*ANSWER*

| ID | DEPT | SALARY | COMM |
|---|---|---|---|
| 130 | 42 | 10505.90 | 75.60 |
| 270 | 66 | 18555.50 | |
| 330 | 66 | 10988.00 | 55.50 |

The above query is arguably more complex than it seems at first glance, because unlike in an ordinary query, the CASE checks are applied in the sequence they are defined. So a row will only match "B" if it has not already matched "A". In order to rewrite the above query using standard AND/OR predicates, we have to reproduce the CASE processing sequence. To this end, the three predicates in the next example that look for matching rows also apply any predicates that preceded them in the CASE statement:

*Same stmt as prior, without CASE predicate*

```
SELECT id
     , name
     , salary
     , comm
FROM staff
WHERE (comm < 70)
   OR (salary < 11000 AND NOT name LIKE 'W%')
   OR (salary < 19000 AND NOT (name LIKE 'W%'
                          OR (salary < 18500 AND dept <> 33)
    )
   )
ORDER BY id;
```

*ANSWER*

| ID | DEPT | SALARY | COMM |
|---|---|---|---|
| 130 | 42 | 10505.90 | 75.60 |
| 270 | 66 | 18555.50 | |
| 330 | 66 | 10988.00 | 55.50 |

# 2.10. Miscellaneous SQL Statements

This section will briefly discuss several miscellaneous SQL statements. See the Db2 manuals for more details.

## 2.10.1. Cursor

A cursor is used in an application program to retrieve and process individual rows from a result set. To use a cursor, one has to do the following:

- **DECLARE** the cursor. The declare statement has the SQL text that the cursor will run. If the cursor is declared "with hold", it will remain open after a commit, otherwise it will be closed at commit time.

The declare cursor statement is not actually executed when the program is run. It simply defines the query that will be run.

- **OPEN** the cursor. This is when the contents of on any host variables referenced by the cursor (in the predicate part of the query) are transferred to Db2.
- **FETCH** rows from the cursor. One does as many fetches as is needed. If no row is found, the SQLCODE from the fetch will be 100.
- **CLOSE** the cursor.

**Syntax Notes**

- The cursor-name must be unique with the application program.
- The WITH HOLD phrase indicates that the cursor will remain open if the unit of work ends with a commit. The cursor will be closed if a rollback occurs.
- The WITH RETURN phrase is used when the cursor will generate the result set returned by a stored procedure. If the cursor is open when the stored procedure ends the result set will be return either to the calling procedure, or directly to the client application.
- The FOR phrase can either refer to a select statement, the text for which will follow, or to the name of a statement has been previously prepared.

**Usage notes**

- Cursors that require a sort (e.g. to order the output) will obtain the set of matching rows at open time, and then store them in an internal temporary table. Subsequent fetches will be from the temporary table.
- Cursors that do not require a sort are resolved as each row is fetched from the data table.
- All references to the current date, time, and timestamp will return the same value (i.e. as of when the cursor was opened) for all fetches in a given cursor invocation.
- One does not have to close a cursor, but one cannot reopen it until it is closed. All open cursors are automatically closed when the thread terminates, or when a rollback occurs, or when a commit is done - except if the cursor is defined "with hold".
- One can both update and delete "where current of cursor". In both cases, the row most recently fetched is updated or deleted. An update can only be used when the cursor being referenced is declared "for update of".

**Examples**

*Sample cursor*

```
DECLARE fred CURSOR FOR
WITH RETURN TO CALLER
SELECT id
     , name
     , salary
     , comm
FROM staff
WHERE id < :id-var
AND   salary > 1000
ORDER BY id ASC
FETCH FIRST 10 ROWS ONLY
OPTIMIZE FOR 10 ROWS
FOR FETCH ONLY
WITH UR
```

*Use cursor in program*

```
DECLARE fred CURSOR WITH HOLD FOR
SELECT name
     , salary
FROM staff
WHERE id > :id-var
FOR UPDDATE OF salary, comm

OPEN fred

DO UNTIL SQLCODE = 100
    FETCH fred INTO :name-var
                  , :salary-var
    IF salary < 1000 THEN
    DO
        UPDATE staff
        SET salary = :new-salary-var
        WHERE CURRENT OF fred
    END-IF
END-DO

CLOSE fred
```

### 2.10.2. Select Into

A SELECT-INTO statement is used in an application program to retrieve a single row. If more than one row matches, an error is returned. The statement text is the same as any ordinary query, except that there is an INTO section (listing the output variables) between the SELECT list and the FROM section.

#### Example

*Singleton select*

```
SELECT name
     , salary
INTO :name-var
   , :salary-var
FROM staff
WHERE id = :id-var
```

### 2.10.3. Prepare

The PREPARE statement is used in an application program to dynamically prepare a SQL statement for subsequent execution.

#### Syntax Notes

- The statement name names the statement. If the name is already in use, it is overridden.
- The OUTPUT descriptor will contain information about the output parameter markers.
- The DESCRIBE statement may be used instead of this clause.
- The INPUT descriptor will contain information about the input parameter markers.
- The FROM phrase points to the host-variable which contains the SQL statement text.

Prepared statement can be used by the following:

*Table 6. What statements can use prepared statement*

| STATEMENT CAN BE USED BY | STATEMENT TYPE |
|---|---|
| DESCRIBE | Any statement |
| DECLARE CURSOR | Must be SELECT |
| EXECUTE | Must not be SELECT |

### 2.10.4. Describe

The DESCRIBE statement is typically used in an application program to get information about a prepared statement. It can also be used in the Db2 command processor (but not in Db2BATCH) to get a description of a table, or the output columns in a SQL statement:

Below are some examples of using the statement:

*DESCRIBE the output columns in a select statement*

```
DESCRIBE OUTPUT SELECT * FROM staff
SQLDA Information
sqldaid : SQLDA sqldabc: 896  sqln: 20 sqld: 7
Column Information



sqltype           sqllen sqlname.data              sqlname.length
----------------- -----  ------------------------- -------------
500 SMALLINT      2      ID                        2
449 VARCHAR       9      NAME                      4
501 SMALLINT      2      DEPT                      4
453 CHARACTER     5      JOB                       3
501 SMALLINT      2      YEARS                     5
485 DECIMAL       7, 2   SALARY                    6
485 DECIMAL       7, 2   COMM                      4
```

*DESCRIBE the columns in a table*

```
DESCRIBE TABLE staff

Column name             Type schema Type name    Length Scale Nulls
----------------------  -------     -----------  ------ ----  ----
ID                      SYSIBM      SMALLINT     2      0     No
NAME                    SYSIBM      VARCHAR      9      0     Yes
DEPT                    SYSIBM      SMALLINT     2      0     Yes
JOB                     SYSIBM      CHARACTER    5      0     Yes
YEARS                   SYSIBM      SMALLINT     2      0     Yes
SALARY                  SYSIBM      DECIMAL      7      2     Yes
COMM                    SYSIBM      DECIMAL      7      2     Yes
```

### 2.10.5. Execute

The EXECUTE statement is used in an application program to execute a prepared statement. The statement can not be a select.

### 2.10.6. Execute Immediate

The EXECUTE IMMEDIATE statement is used in an application program to prepare and execute a statement. Only certain kinds of statement (e.g. insert, update, delete, commit) can be run this way. The statement can not be a select.

### 2.10.7. Set Variable

The SET statement is used in an application program to set one or more program variables to values that are returned by Db2.

#### Examples

*SET single host-variable*

```
SET :host-var = CURRENT TIMESTAMP
```

*SET multiple host-variables*

```
SET :host-v1 = CURRENT TIME
  , :host-v2 = CURRENT DEGREE
  , :host-v3 = NULL
```

The SET statement can also be used to get the result of a select, as long as the select only returns a single row:

*SET using row-fullselect*

```
SET
( :hv1
, :hv2
, :hv3) =
(SELECT id
      , name
      , salary
FROM staff
WHERE   id = :id-var)
```

### 2.10.8. Set Db2 Control Structures

In addition to setting a host-variable, one can also set various Db2 control structures:

*Other SET statements*

```
SET CONNECTION
SET CURRENT DEFAULT TRANSFORM GROUP
SET CURRENT DEGREE
SET CURRENT EXPLAIN MODE
SET CURRENT EXPLAIN SNAPSHOT
SET CURRENT ISOLATION
SET CURRENT LOCK TIMEOUT
SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION
SET CURRENT PACKAGE PATH
SET CURRENT PACKAGESET
SET CURRENT QUERY OPTIMIZATION
SET CURRENT REFRESH AGE
SET ENCRYPTION PASSWORD
SET EVENT MONITOR STATE
SET INTEGRITY SET PASSTHRU
SET PATH
SET SCHEMA
SET SERVER OPTION
SET SESSION AUTHORIZATION
```

## 2.11. Unit-of-Work Processing

No changes that you make are deemed to be permanent until they are committed. This section briefly lists the commands one can use to commit or rollback changes.

### 2.11.1. Commit

The COMMIT statement is used to commit whatever changes have been made. Locks that were taken as a result of those changes are freed. If no commit is specified, an implicit one is done when the thread terminates.

### 2.11.2. Savepoint

The SAVEPOINT statement is used in an application program to set a savepoint within a unit of work. Subsequently, the program can be rolled back to the savepoint, as opposed to rolling back to the start of the unit of work.

**Notes**

- If the savepoint name is the same as a savepoint that already exists within the same level, it overrides the prior savepoint - unless the latter was defined a being unique, in which case an error is returned.
- The RETAIN CURSORS phrase tells Db2 to, if possible, keep open any active cursors.
- The RETAIN LOCKS phrase tells Db2 to retain any locks that were obtained subsequent to the savepoint. In other words, the changes are rolled back, but the locks that came with those changes remain.

**Savepoint Levels**

Savepoints exist within a particular savepoint level, which can be nested within another level. A new level is created whenever one of the following occurs:

- A new unit of work starts.
- A procedure defined with NEW SAVEPOINT LEVEL is called.
- An atomic compound SQL statement starts.

A savepoint level ends when the process that caused its creation finishes. When a savepoint level ends, all of the savepoints created within it are released. The following rules apply to savepoint usage:

- Savepoints can only be referenced from within the savepoint level in which they were created. Active savepoints in prior levels are not accessible.
- The uniqueness of savepoint names is only enforced within a given savepoint level. The same name can exist in multiple active savepoint levels.

**Example**

Savepoints are especially useful when one has multiple SQL statements that one wants to run or rollback as a whole, without affecting other statements in the same transaction. For example, imagine that one is transferring customer funds from one account to another. Two updates will be required - and if one should fail, both should fail:

*Example of savepoint usage*

```
INSERT INTO transaction_audit_table;
SAVEPOINT before_updates ON ROLLBACK RETAIN CURSORS;
UPDATE savings_account
    SET balance = balance - 100
    WHERE cust# = 1234;
IF SQLCODE <> 0 THEN
    ROLLBACK TO SAVEPOINT before_updates;
ELSE
    UPDATE checking_account
        SET balance = balance + 100
        WHERE cust# = 1234;
    IF SQLCODE <> 0 THEN
        ROLLBACK TO SAVEPOINT before_updates;
    END
END
COMMIT;
```

In the above example, if either of the update statements fail, the transaction is rolled back to the predefined savepoint. And regardless of what happens, there will still be a row inserted into the transaction-audit table.

### 2.11.3. Savepoints vs. Commits

Savepoints differ from commits in the following respects:

- One cannot rollback changes that have been committed.
- Only a commit guarantees that the changes are stored in the database. If the program subsequently fails, the data will still be there.
- Once a commit is done, other users can see the changed data. After a savepoint, the data is still not visible to other users.

### 2.11.4. Release Savepoint

The RELEASE SAVEPOINT statement will remove the named savepoint. Any savepoints nested within the named savepoint are also released. Once run, the application can no longer rollback to any of the released savepoints.

### 2.11.5. Rollback

The ROLLBACK statement is used to rollback any database changes since the beginning of the unit of work, or since the named savepoint - if one is specified.

# Chapter 3. Data Manipulation Language

The chapter has a very basic introduction to the DML (Data Manipulation Language) statements. See the Db2 manuals for more details.

## 3.1. Insert

The INSERT statement is used to insert rows into a table, view, or fullselect. To illustrate how it is used, this section will use a copy of the EMP_ACT sample table:

*EMP_ACT_COPY sample table - DDL*

```
CREATE TABLE emp_act_copy
( empno    CHARACTER (00006) NOT NULL
, projno   CHARACTER (00006) NOT NULL
, actno    SMALLINT          NOT NULL
, emptime  DECIMAL (05,02)
, emstdate DATE
, emendate DATE);
```

### 3.1.1. Target Objects

One can insert into a table, view, nickname, or SQL expression. For views and SQL expressions, the following rules apply:

- The list of columns selected cannot include a column function (e.g. MIN).
- There must be no GROUP BY or HAVING acting on the select list.
- The list of columns selected must include all those needed to insert a new row.
- The list of columns selected cannot include one defined from a constant, expression, or a scalar function.
- Sub-queries, and other predicates, are fine, but are ignored (see Insert into a fullselect).
- The query cannot be a join, nor (plain) union.
- A "union all" is permitted - as long as the underlying tables on either side of the union have check constraints such that a row being inserted is valid for one, and only one, of the tables in the union.

All bets are off if the insert is going to a table that has an INSTEAD OF trigger defined.

**Usage Notes**

- One has to provide a list of the columns (to be inserted) if the set of values provided does not equal the complete set of columns in the target table, or are not in the same order as the columns are defined in the target table.
- The columns in the INCLUDE list are not inserted. They are intended to be referenced in a SELECT statement that encompasses the INSERT (see Select DML Changes).

- The input data can either be explicitly defined using the VALUES statement, or retrieved from some other table using a fullselect.

### 3.1.2. Direct Insert

To insert a single row, where all of the columns are populated, one lists the input the values in the same order as the columns are defined in the table:

*Single row insert*

```
INSERT INTO emp_act_copy VALUES
('100000' ,'ABC' ,10 ,1.4 ,'2003-10-22', '2003-11-24');
```

To insert multiple rows in one statement, separate the row values using a comma:

*Multi row insert*

```
INSERT INTO emp_act_copy VALUES
    ('200000' ,'ABC' ,10 ,1.4 ,'2003-10-22', '2003-11-24')
,   ('200000' ,'DEF' ,10 ,1.4 ,'2003-10-22', '2003-11-24')
,   ('200000' ,'IJK' ,10 ,1.4 ,'2003-10-22', '2003-11-24');
```

If multiple rows are inserted in one statement, and one of them violates a unique index check, all of the rows are rejected.

The NULL and DEFAULT keywords can be used to assign these values to columns. One can also refer to special registers, like the current date and current time:

*Using null and default values*

```
INSERT INTO emp_act_copy VALUES
('400000' ,'ABC' ,10 ,NULL ,DEFAULT, CURRENT DATE);
```

To leave some columns out of the insert statement, one has to explicitly list the columns that are included. When this is done, one can refer to the columns used in any order:

*Explicitly listing columns being populated during insert*

```
INSERT INTO emp_act_copy (projno, emendate, actno, empno)
VALUES
('ABC' ,DATE(CURRENT TIMESTAMP) ,123 ,'500000');
```

### 3.1.3. Insert into Full-Select

The next statement inserts a row into a fullselect that just happens to have a predicate which, if used in a subsequent query, would not find the row inserted. The predicate has no impact on the insert itself:

*Insert into a fullselect*

```
INSERT INTO
(SELECT *
 FROM emp_act_copy
 WHERE empno < '1'
)
VALUES
('510000' ,'ABC' ,10 ,1.4 ,'2003-10-22', '2003-11-24');
```

One can insert rows into a view (with predicates in the definition) that are outside the bounds of the predicates. To prevent this, define the view WITH CHECK OPTION.

### 3.1.4. Insert from Select

One can insert a set of rows that is the result of a query using the following notation:

*Insert result of select statement*

```
INSERT INTO emp_act_copy
SELECT LTRIM(CHAR(id + 600000))
     , SUBSTR(UCASE(name),1,6)
     , salary / 229
     , 123
     , CURRENT DATE
     , '2003-11-11'
FROM staff
WHERE id < 50;
```

In the above example, the fractional part of the SALARY value is eliminated when the data is inserted into the ACTNO field, which only supports integer values.

If only some columns are inserted using the query, they need to be explicitly listed:

*Insert result of select - specified columns only*

```
INSERT INTO emp_act_copy (empno, actno, projno)
SELECT LTRIM(CHAR(id + 700000))
     , MINUTE(CURRENT TIME)
     , 'DEF'
FROM staff
WHERE id < 40;
```

One reason why tables should always have unique indexes is to stop stupid SQL statements like the following, which will double the number of rows in the table:

*Stupid - insert - doubles rows*

```
INSERT INTO emp_act_copy
SELECT *
FROM emp_act_copy;
```

The select statement using the insert can be as complex as one likes. In the next example, it contains the union of two queries:

*Inserting result of union*

```
INSERT INTO emp_act_copy (empno, actno, projno)
SELECT LTRIM(CHAR(id + 800000))
     , 77
     , 'XYZ'
FROM staff
WHERE id < 40
UNION
SELECT LTRIM(CHAR(id + 900000))
     , SALARY / 100
     , 'DEF'
FROM staff
WHERE id < 50;
```

The select can also refer to a common table expression. In the following example, six values are first generated, each in a separate row. These rows are then selected during the insert:

*Insert from common table expression*

```
INSERT INTO emp_act_copy (empno, actno, projno, emptime)
WITH temp1 (col1) AS
( VALUES (1),(2),(3),(4),(5),(6))
SELECT LTRIM(CHAR(col1 + 910000))
     , col1
     , CHAR(col1)
     , col1 / 2
FROM temp1;
```

The next example inserts multiple rows - all with an EMPNO beginning "92". Three rows are found in the STAFF table, and all three are inserted, even though the sub-query should get upset once the first row has been inserted. This doesn't happen because all of the matching rows in the STAFF table are retrieved and placed in a work-file before the first insert is done:

*Insert with irrelevant sub-query*

```
INSERT INTO emp_act_copy (empno, actno, projno)
SELECT LTRIM(CHAR(id + 920000))
     , id
     , 'ABC'
FROM staff
WHERE id < 40
AND NOT EXISTS
( SELECT *
 FROM emp_act_copy
 WHERE empno LIKE '92%'
);
```

### 3.1.5. Insert into Multiple Tables

Below are two tables that hold data for US and international customers respectively:

*Customer tables - for insert usage*

```
CREATE TABLE us_customer
( cust#     INTEGER     NOT NULL
, cname     CHAR(10)    NOT NULL
, country   CHAR(03)    NOT NULL
, CHECK (country = 'USA')
, PRIMARY KEY (cust#));

CREATE TABLE intl_customer
( cust#     INTEGER  NOT NULL
, cname     CHAR(10) NOT NULL
, country   CHAR(03) NOT NULL
, CHECK (country <> 'USA')
, PRIMARY KEY (cust#));
```

One can use a single insert statement to insert into both of the above tables because they have mutually exclusive check constraints. This means that a new row will go to one table or the other, but not both, and not neither. To do so one must refer to the two tables using a "union all" phrase - either in a view, or a query, as is shown below:

*Insert into multiple tables*

```
INSERT INTO
(SELECT *
 FROM us_customer
 UNION ALL
 SELECT *
 FROM intl_customer
)
VALUES
 (111,'Fred','USA')
,(222,'Dave','USA')
,(333,'Juan','MEX');
```

The above statement will insert two rows into the table for US customers, and one row into the table for international customers.

## 3.2. Update

The UPDATE statement is used to change one or more columns/rows in a table, view, or fullselect. Each column that is to be updated has to specified. Here is an example:

*Single row update*

```
UPDATE emp_act_copy
SET emptime  = NULL
  , emendate = DEFAULT
  , emstdate = CURRENT DATE + 2 DAYS
  , actno    = ACTNO / 2
  , projno   = 'ABC'
WHERE empno = '100000';
```

**Usage Notes**

- One can update rows in a table, view, or fullselect. If the object is not a table, then it must be updateable (i.e. refer to a single table, not have any column functions, etc).
- The correlation name is optional, and is only needed if there is an expression or predicate that references another table.
- The columns in the INCLUDE list are not updated. They are intended to be referenced in a SELECT statement that encompasses the UPDATE (see Select DML Changes).
- The SET statement lists the columns to be updated, and the new values they will get.
- Predicates are optional. If none are provided, all rows in the table are updated.
- Usually, all matching rows are updated. To update some fraction of the matching rows, use a fullselect (see Use Full-Select).

**Update Examples**

To update all rows in a table, leave off all predicates:

*Mass update*

```
UPDATE emp_act_copy
SET actno = actno / 2;
```

In the next example, both target columns get the same values. This happens because the result for both columns is calculated before the first column is updated:

*Two columns get same value*

```
UPDATE emp_act_copy ac1
SET actno   = actno * 2
  , emptime = actno * 2
WHERE empno LIKE '910%';
```

One can also have an update refer to the output of a select statement - as long as the result of the select is a single row:

*Update using select*

```
UPDATE emp_act_copy
SET actno =
    ( SELECT MAX(salary) / 10
      FROM staff)
WHERE empno = '200000';
```

The following notation lets one update multiple columns using a single select:

*Multi-row update using select*

```
UPDATE emp_act_copy
SET (actno, emstdate, projno) =
    ( SELECT MAX(salary) / 10
    , CURRENT DATE + 2 DAYS
    , MIN(CHAR(id))
    FROM staff
    WHERE id <> 33
    )
WHERE empno LIKE '600%';
```

Multiple rows can be updated using multiple different values, as long as there is a one-to-one relationship between the result of the select, and each row to be updated.

*Multi-row update using correlated select*

```
UPDATE emp_act_copy ac1
SET (actno, emptime) =
    (SELECT ac2.actno + 1
          , ac1.emptime / 2
      FROM emp_act_copy ac2
      WHERE ac2.empno LIKE '60%'
      AND SUBSTR(ac2.empno,3) = SUBSTR(ac1.empno,3)
     )
WHERE EMPNO LIKE '700%';
```

### 3.2.1. Use Full-Select

An update statement can be run against a table, a view, or a fullselect. In the next example, the table is referred to directly:

*Direct update of table*

```
UPDATE emp_act_copy
SET   emptime = 10
WHERE empno   = '000010'
AND   projno  = 'MA2100';
```

Below is a logically equivalent update that pushes the predicates up into a fullselect:

*Update of fullselect*

```
UPDATE
(SELECT *
 FROM  emp_act_copy
 WHERE empno  = '000010'
 AND   projno = 'MA2100'
) AS ea
SET emptime = 20;
```

### 3.2.2. Update First "n" Rows

An update normally changes all matching rows. To update only the first "n" matching rows do the following:

- In a fullselect, retrieve the first "n" rows that you want to update.
- Update all rows returned by the fullselect.

In the next example, the first five staff with the highest salary get a nice fat commission:

*Update first "n" rows*

```
UPDATE
(SELECT *
 FROM staff
 ORDER BY salary DESC
 FETCH FIRST 5 ROWS ONLY
) AS xxx
SET comm = 10000;
```

The above statement may update five random rows – if there is more than one row with the ordering value. To prevent this, ensure that the list of columns provided in the ORDER BY identify each matching row uniquely.

### 3.2.3. Use OLAP Function

Imagine that we want to set the employee-time for a particular row in the EMP_ACT table to the MAX time for that employee. Below is one way to do it:

*Set employee-time in row to MAX - for given employee*

```
UPDATE emp_act_copy ea1
SET emptime =
    (SELECT MAX(emptime)
     FROM emp_act_copy ea2
     WHERE ea1.empno = ea2.empno
    )
WHERE empno  = '000010'
AND   projno = 'MA2100';
```

The same result can be achieved by calling an OLAP function in a fullselect, and then updating the result. In next example, the MAX employee-time per employee is calculated (for each row), and placed in a new column. This column is then used to do the final update:

*Use OLAP function to get max-time, then apply (correct)*

```
UPDATE
    (SELECT ea1.*
          , MAX(emptime) OVER(PARTITION BY empno) AS maxtime
     FROM emp_act_copy ea1
    ) AS ea2
SET   emptime = maxtime
WHERE empno   = '000010'
AND   projno  = 'MA2100';
```

The above statement has the advantage of only accessing the EMP_ACT table once. If there were many rows per employee, and no suitable index (i.e. on EMPNO and EMPTIME), it would be much faster than the prior update. The next update is similar to the prior - but it does the wrong update!

In this case, the scope of the OLAP function is constrained by the predicate on PROJNO, so it no longer gets the MAX time for the employee:

*Use OLAP function to get max-time, then apply (wrong)*

```
UPDATE emp_act_copy
SET    emptime = MAX(emptime) OVER(PARTITION BY empno)
WHERE  empno   = '000010'
AND    projno  = 'MA2100';
```

### 3.2.4. Correlated and Uncorrelated Update

In the next example, regardless of the number of rows updated, the ACTNO will always come out as one. This is because the sub-query that calculates the row-number is correlated, which means that it is resolved again for each row to be updated in the "AC1" table. At most, one "AC2" row will match, so the row-number must always equal one:

*Update with correlated query*

```
UPDATE emp_act_copy ac1
SET (actno, emptime)
    = (SELECT ROW_NUMBER() OVER()
            , ac1.emptime / 2
       FROM emp_act_copy ac2
       WHERE ac2.empno LIKE '60%'
       AND SUBSTR(ac2.empno,3) = SUBSTR(ac1.empno,3)
      )
WHERE EMPNO LIKE '800%';
```

In the next example, the ACTNO will be updated to be values 1, 2, 3, etc, in order that the rows are updated. In this example, the sub-query that calculates the row-number is uncorrelated, so all of the matching rows are first resolved, and then referred to in the next, correlated, step:

*Update with uncorrelated query*

```
UPDATE emp_act_copy ac1
SET (actno, emptime) =
    (SELECT c1
          , c2
     FROM (SELECT ROW_NUMBER() OVER() AS c1
                , actno / 100         AS c2
                , empno
           FROM emp_act_copy
           WHERE empno LIKE '60%'
          ) AS ac2
     WHERE SUBSTR(ac2.empno,3) = SUBSTR(ac1.empno,3)
    )
WHERE empno LIKE '900%';
```

## 3.3. Delete

The DELETE statement is used to remove rows from a table, view, or fullselect. The set of rows deleted depends on the scope of the predicates used. The following example would delete a single row from the EMP_ACT sample table:

*Single-row delete*

```
DELETE
FROM emp_act_copy
WHERE empno  = '000010'
AND   projno = 'MA2100'
AND   actno  = 10;
```

**Usage Notes**

- One can delete rows from a table, view, or fullselect. If the object is not a table, then it must be deletable (i.e. refer to a single table, not have any column functions, etc).
- The correlation name is optional, and is only needed if there is a predicate that references another table.
- The columns in the INCLUDE list are not updated. They are intended to be referenced in a SELECT statement that encompasses the DELETE (see Select DML Changes).
- Predicates are optional. If none are provided, all rows are deleted.
- Usually, all matching rows are deleted. To delete some fraction of the matching rows, use a fullselect (see Use Full-Select).

### 3.3.1. Basic Delete

This statement would delete all rows in the EMP_ACT table:

*Mass delete*

```
DELETE
FROM emp_act_copy;
```

This statement would delete all the matching rows in the EMP_ACT:

*Selective delete*

```
DELETE
FROM emp_act_copy
WHERE empno  LIKE '00%'
AND   projno >=   'MA';
```

### 3.3.2. Correlated Delete

The next example deletes all the rows in the STAFF table - except those that have the highest ID in their respective department:

*Correlated delete (1 of 2)*

```
DELETE
FROM staff s1
WHERE id NOT IN
    (SELECT MAX(id)
     FROM staff s2
     WHERE s1.dept = s2.dept
    );
```

Here is another way to write the same:

*Correlated delete (2 of 2)*

```
DELETE
FROM staff s1
WHERE EXISTS
    (SELECT *
     FROM staff s2
     WHERE s2.dept = s1.dept
     AND   s2.id > s1.id
    );
```

The next query is logically equivalent to the prior two, but it works quite differently. It uses a fullselect and an OLAP function to get, for each row, the ID, and also the highest ID value in the current department. All rows where these two values do not match are then deleted:

*Delete using fullselect and OLAP function*

```
DELETE
FROM
    (SELECT id
          , MAX(id) OVER(PARTITION BY dept) AS max_id
     FROM staff
    ) AS ss
WHERE id <> max_id;
```

### 3.3.3. Delete First "n" Rows

A delete removes all encompassing rows. Sometimes this is not desirable - usually because an unknown, and possibly undesirably large, number rows is deleted. One can write a delete that stops after "n" rows using the following logic:

- In a fullselect, retrieve the first "n" rows that you want to delete.

- Delete all rows returned by the fullselect.

In the following example, those five employees with the highest salary are deleted:

*Delete first "n" rows*

```
DELETE
FROM
    (SELECT *
     FROM staff
     ORDER BY salary DESC
     FETCH FIRST 5 ROWS ONLY
) AS xxx;
```

The above statement may delete five random rows – if there is more than one row with the same salary. To prevent this, ensure that the list of columns provided in the ORDER BY identify each matching row uniquely.

# 3.4. Select DML Changes

A special kind of SELECT statement (see Select DML Changes) can encompass an INSERT, UPDATE, or DELETE statement to get the before or after image of whatever rows were changed (e.g. select the list of rows deleted). This kind of SELECT can be very useful when the DML statement is internally generating a value that one needs to know (e.g. an INSERT automatically creates a new invoice number using a sequence column), or get the set of rows that were removed by a delete. All of this can be done by coding a special kind of select.

## 3.4.1. Table Types

- **OLD**: Returns the state of the data prior to the statement being run. This is allowed for an update and a delete.
- **NEW**: Returns the state of the data prior to the application of any AFTER triggers or referential constraints. Data in the table will not equal what is returned if it is subsequently changed by AFTER triggers or R.I. This is allowed for an insert and an update.
- **FINAL**: Returns the final state of the data. If there AFTER triggers that alter the target table after running of the statement, an error is returned. Ditto for a view that is defined with an INSTEAD OF trigger. This is allowed for an insert and an update.

**Usage Notes**

- Only one of the above tables can be listed in the FROM statement.
- The table listed in the FROM statement cannot be given a correlation name.
- No other table can be listed (i.e. joined to) in the FROM statement. One can reference another table in the SELECT list (see example Join result to another table), or by using a sub-query in the predicate section of the statement.
- The SELECT statement cannot be embedded in a nested-table expression.

- The SELECT statement cannot be embedded in an insert statement.
- To retrieve (generated) columns that are not in the target table, list them in an INCLUDE phrase in the DML statement. This technique can be used to, for example, assign row numbers to the set of rows entered during an insert.
- Predicates (on the select) are optional. They have no impact on the underlying DML.
- The INPUT SEQUENCE phrase can be used in the ORDER BY to retrieve the rows in the same sequence as they were inserted. It is not valid in an update or delete.
- The usual scalar functions, OLAP functions, and column functions, plus the GROUP BY phrase, can be applied to the output - as desired.

### 3.4.2. Insert Examples

The example below selects from the final result of the insert:

*Select rows inserted*

```
SELECT empno
, projno AS prj
, actno  AS act
FROM FINAL TABLE
(INSERT INTO emp_act_copy
 VALUES
  ('200000', 'ABC', 10, 1, '2003-10-22', '2003-11-24')
, ('200000', 'DEF', 10, 1, '2003-10-22', '2003-11-24')
)
ORDER BY 1,2,3;
```

*ANSWER*

| EMPNO | PRJ | ACT |
|---|---|---|
| 200000 | ABC | 10 |
| 200000 | DEF | 10 |

One way to retrieve the new rows in the order that they were inserted is to include a column in the insert statement that is a sequence number:

*Include column to get insert sequence*

```
SELECT empno
, projno AS prj
, actno AS act
, row#
AS r#
FROM FINAL TABLE
    (INSERT INTO emp_act_copy (empno, projno, actno)
    INCLUDE (row# SMALLINT)
    VALUES
     ('300000','ZZZ',999,1)
    ,('300000','VVV',111,2)
   )
ORDER BY row#;
```

*ANSWER*

| EMPNO | PRJ | ACT | R# |
|---|---|---|---|
| 300000 | ZZZ | 999 | 1 |
| 300000 | VVV | 111 | 2 |

The next example uses the INPUT SEQUENCE phrase to select the new rows in the order that they were inserted. Row numbers are assigned to the output:

*Select rows in insert order*

```
SELECT empno
     , projno AS prj
     , actno AS act
     , ROW_NUMBER() OVER() AS r#
FROM FINAL TABLE
    (INSERT INTO emp_act_copy (empno, projno, actno)
     VALUES
       ('400000','ZZZ',999)
     , ('400000','VVV',111)
    )
ORDER BY INPUT SEQUENCE;
```

*ANSWER*

| EMPNO | PRJ | ACT | R# |
|---|---|---|---|
| 400000 | ZZZ | 999 | 1 |
| 400000 | VVV | 111 | 2 |

The INPUT SEQUENCE phrase only works in an insert statement. It can be listed in the ORDER BY part of the statement, but not in the SELECT part. The only way to display the row number of each row inserted is to explicitly assign row numbers.

In the next example, the only way to know for sure what the insert has done is to select from the result. This is because the select statement (in the insert) has the following unknowns:

- We do not, or may not, know what ID values were selected, and thus inserted.
- The project-number is derived from the current-time special register.
- The action-number is generated using the RAND function.

Now for the insert:

*Select from an insert that has unknown values*

```
SELECT empno
,projno AS prj
,actno AS act
,ROW_NUMBER() OVER() AS r#
FROM NEW TABLE
    (INSERT INTO emp_act_copy (empno, actno, projno)
        SELECT LTRIM(CHAR(id + 600000))
             , SECOND(CURRENT TIME)
             , CHAR(SMALLINT(RAND(1) * 1000))
        FROM staff
        WHERE id < 40
    )
ORDER BY INPUT SEQUENCE;
```

*ANSWER*

| EMPNO | PRJ | ACT | R# |
|---|---|---|---|
| 600010 | 1 | 59 | 1 |
| 600020 | 563 | 59 | 2 |
| 600030 | 193 | 59 | 3 |

### 3.4.3. Update Examples

The statement below updates the matching rows by a fixed amount. The select statement gets the old EMPTIME values:

*Select values - from before update*

```
SELECT empno
     , projno  AS prj
     , emptime AS etime
FROM OLD TABLE
    (UPDATE emp_act_copy
     SET emptime = emptime * 2
     WHERE empno = '200000')
ORDER BY projno;
```

*ANSWER*

| EMPNO | PRJ | ETIME |
|---|---|---|
| 200000 | ABC | 1.00 |
| 200000 | DEF | 1.00 |

The next statement updates the matching EMPTIME values by random amount. To find out exactly what the update did, we need to get both the old and new values. The new values are obtained by selecting from the NEW table, while the old values are obtained by including a column in the update which is set to them, and then subsequently selected:

*Select values - before and after update*

```
SELECT projno  AS prj
     , old_t   AS old_t
     , emptime AS new_t
FROM NEW TABLE
    (UPDATE emp_act_copy
     INCLUDE (old_t DECIMAL(5,2))
     SET    emptime = emptime * RAND(1) * 10
          , old_t   = emptime
     WHERE empno    = '200000'
    )
ORDER BY 1;
```

*ANSWER*

| PRJ | OLD_T | NEW_T |
|---|---|---|
| ABC | 2.00 | 0.02 |
| DEF | 2.00 | 11.27 |

### 3.4.4. Delete Examples

The following example lists the rows that were deleted:

*List deleted rows*

```
SELECT projno AS prj
     , actno  AS act
FROM OLD TABLE
    (DELETE
    FROM emp_act_copy
    WHERE empno = '300000'
   )
ORDER BY 1,2;
```

*ANSWER*

| PRJ | ACT |
|---|---|
| VVV | 111 |
| ZZZ | 999 |

The next query deletes a set of rows, and assigns row-numbers (to the included field) as the rows are deleted. The subsequent query selects every second row:

*Assign row numbers to deleted rows*

```
SELECT empno
     , projno
     , actno AS act
     , row#  AS r#
FROM OLD TABLE
    (DELETE
     FROM emp_act_copy
     INCLUDE (row# SMALLINT)
     SET    row# = ROW_NUMBER() OVER()
     WHERE empno = '000260'
    )
WHERE row# = row# / 2 * 2
ORDER BY 1, 2, 3;
```

*ANSWER*

| EMPNO | PROJNO | ACT | R# |
|---|---|---|---|
| 000260 | AD3113 | 70 | 2 |
| 000260 | AD3113 | 80 | 4 |
| 000260 | AD3113 | 180 | 6 |

Predicates (in the select result phrase) have no impact on the range of rows changed by the underlying DML, which is determined by its own predicates.

One cannot join the table generated by a DML statement to another table, nor include it in a nested table expression, but one can join in the SELECT phrase. The following delete illustrates this concept by joining to the EMPLOYEE table:

*Join result to another table*

```
SELECT empno
    , (SELECT lastname
       FROM
        (SELECT empno AS e#
              , lastname
         FROM employee
        ) AS xxx
       WHERE empno = e#)
    , projno AS projno
    , actno AS act
FROM OLD TABLE
    (DELETE
     FROM emp_act_copy
     WHERE empno < '0001')
ORDER BY 1, 2, 3
FETCH FIRST 5 ROWS ONLY;
```

*ANSWER*

| EMPNO | LASTNAME | PROJNO | ACT |
|---|---|---|---|
| 000010 | HAAS | AD3100 | 10 |
| 000010 | HAAS | MA2100 | 10 |
| 000010 | HAAS | MA2110 | 10 |
| 000020 | THOMPSON | PL2100 | 30 |
| 000030 | KWAN | IF1000 | 10 |

Observe above that the EMPNO field in the EMPLOYEE table was be renamed (before doing the join) using a nested table expression. This was necessary because one cannot join on two fields that have the same name, without using correlation names. A correlation name cannot be used on the OLD TABLE, so we had to rename the field to get around this problem.

## 3.5. Merge

The MERGE statement is a combination insert and update, or delete, statement on steroids. It can be used to take the data from a source table, and combine it with the data in a target table.

The qualifying rows in the source and target tables are first matched by unique key value, and then evaluated:

- If the source row is already in the target, the latter can be either updated or deleted.
- If the source row is not in the target, it can be inserted.

- If desired, a SQL error can also be generated.

**Usage Rules**

The following rules apply to the merge statement:

- Correlation names are optional, but are required if the field names are not unique.
- If the target of the merge is a fullselect or a view, it must allow updates, inserts, and deletes - as if it were an ordinary table.
- At least one ON condition must be provided.
- The ON conditions must uniquely identify the matching rows in the target table.
- Each individual WHEN check can only invoke a single modification statement.
- When a MATCHED search condition is true, the matching target row can be updated, deleted, or an error can be flagged.
- When a NOT MATCHED search condition is true, the source row can be inserted into the target table, or an error can be flagged.
- When more than one MATCHED or NOT MATCHED search condition is true, the first one that matches (for each type) is applied. This prevents any target row from being updated or deleted more than once. Ditto for any source row being inserted.
- The ELSE IGNORE phrase specifies that no action be taken if no WHEN check evaluates to true.
- If an error is encountered, all changes are rolled back.
- Row-level triggers are activated for each row merged, depending on the type of modification that is made. So if the merge initiates an insert, all insert triggers are invoked. If the same input initiates an update, all update triggers are invoked.
- Statement-level triggers are activated, even if no rows are processed. So if a merge does either an insert, or an update, both types of statement triggers are invoked, even if all of the input is inserted.

### 3.5.1. Sample Tables

To illustrate the merge statement, the following test tables were created and populated:

*Sample tables for merge*

```
CREATE TABLE old_staff AS
(SELECT id
      , job
      , salary
 FROM staff
)
WITH NO DATA;

CREATE TABLE new_staff AS
(SELECT id
      , salary
 FROM staff
)
WITH NO DATA;

INSERT INTO old_staff
    SELECT id
         , job
         , salary
    FROM staff
    WHERE id BETWEEN 20 and 40;
```

**OLD_STAFF**

| ID | JOB | SALARY |
|---|---|---|
| 20 | Sales | 78171.25 |
| 30 | Mgr | 77506.75 |
| 40 | Sales | 78006.00 |

```
INSERT INTO new_staff
SELECT id, salary / 10
FROM staff
WHERE id BETWEEN 30 and 50;
```

**NEW_STAFF**

| ID | SALARY |
|---|---|
| 30 | 7750.67 |
| 40 | 7800.60 |
| 50 | 8065.98 |

### 3.5.2. Update or Insert Merge

The next statement merges the new staff table into the old, using the following rules:

- The two tables are matched on common ID columns.
- If a row matches, the salary is updated with the new value.
- If there is no matching row, a new row is inserted.

Now for the code:

*Merge - do update or insert*

```
MERGE INTO old_staff oo
USING new_staff nn
ON oo.id = nn.id
WHEN MATCHED THEN
    UPDATE
    SET oo.salary = nn.salary
WHEN NOT MATCHED THEN
    INSERT
    VALUES (nn.id,'?',nn.salary);
```

**OLD_STAFF**

| ID | JOB | SALARY |
|---|---|---|
| 20 | Sales | 78171.25 |
| 30 | Mgr | 77506.75 |
| 40 | Sales | 78006.00 |

**NEW_STAFF**

| ID | SALARY |
|---|---|
| 30 | 7750.67 |
| 40 | 7800.60 |
| 50 | 8065.98 |

**AFTER-MERGE**

| ID | JOB | SALARY |
|---|---|---|
| 20 | Sales | 78171.25 |
| 30 | Mgr | 7750.67 |
| 40 | Sales | 7800.60 |
| 50 | ? | 8065.98 |

### 3.5.3. Delete-only Merge

The next statement deletes all matching rows:

*Merge - delete if match*

```
MERGE INTO old_staff oo
USING new_staff nn
ON oo.id = nn.id
WHEN MATCHED THEN
    DELETE;
```

**AFTER-MERGE**

| ID | JOB | SALARY |
|---|---|---|
| 20 | Sales | 78171.25 |

### 3.5.4. Complex Merge

The next statement has the following options:

- The two tables are matched on common ID columns.
- If a row matches, and the old salary is < 18,000, it is updated.
- If a row matches, and the old salary is > 18,000, it is deleted.
- If no row matches, and the new ID is > 10, the new row is inserted.
- If no row matches, and (by implication) the new ID is ⇐ 10, an error is flagged.

Now for the code:

*Merge with multiple options*

```
MERGE INTO old_staff oo
USING      new_staff nn
ON oo.id = nn.id
WHEN MATCHED
AND oo.salary < 78000 THEN
    UPDATE
    SET oo.salary = nn.salary
WHEN MATCHED
AND oo.salary > 78000 THEN
    DELETE
WHEN NOT MATCHED
AND nn.id > 10 THEN
    INSERT
    VALUES (nn.id,'?',nn.salary)
WHEN NOT MATCHED THEN
    SIGNAL SQLSTATE '70001'
    SET MESSAGE_TEXT = 'New ID <= 10';
```

**OLD_STAFF**

| ID | JOB | SALARY |
|---|---|---|
| 20 | Sales | 78171.25 |
| 30 | Mgr | 77506.75 |
| 40 | Sales | 78006.00 |

**NEW_STAFF**

| ID | SALARY |
|---|---|
| 30 | 7750.67 |
| 40 | 7800.60 |
| 50 | 8065.98 |

**AFTER-MERGE**

| ID | JOB | SALARY |
|---|---|---|
| 20 | Sales | 78171.25 |
| 30 | Mgr | 7750.67 |
| 50 | ? | 8065.98 |

The merge statement is like the case statement (see CASE Expression) in that the sequence in which one writes the WHEN checks determines the processing logic. In the above example, if the last check was written before the prior, any non-match would generate an error.

### 3.5.5. Using a Fullselect

The following merge generates an input table (i.e. fullselect) that has a single row containing the MAX value of every field in the relevant table. This row is then inserted into the table:

*Merge MAX row into table*

```
MERGE INTO old_staff
USING
    (SELECT MAX(id) + 1 AS max_id
          , MAX(job)    AS max_job
          , MAX(salary) AS max_sal
     FROM old_staff
    ) AS mx
ON id = max_id
WHEN NOT MATCHED THEN
    INSERT
    VALUES (max_id, max_job, max_sal);
```

**AFTER-MERGE**

| ID | JOB | SALARY |
|---|---|---|
| 20 | Sales | 78171.25 |
| 30 | Mgr | 77506.75 |
| 40 | Sales | 78006.00 |
| 41 | Sales | 78171.25 |

Here is the same thing written as a plain on insert:

*Merge logic - done using insert*

```
INSERT INTO old_staff
SELECT MAX(id) + 1 AS max_id
     , MAX(job)    AS max_job
     , MAX(salary) AS max_sal
FROM old_staff;
```

Use a fullselect on the target and/or source table to limit the set of rows that are processed during the merge:

*Merge using two fullselects*

```
MERGE INTO
    (SELECT *
     FROM old_staff
     WHERE id < 40
    ) AS oo
USING
    (SELECT *
     FROM new_staff
     WHERE id < 50
    ) AS nn
ON oo.id = nn.id
WHEN MATCHED THEN
    DELETE
WHEN NOT MATCHED THEN
    INSERT
    VALUES (nn.id,'?',nn.salary);
```

**OLD_STAFF**

| ID | JOB | SALARY |
|---|---|---|
| 20 | Sales | 78171.25 |
| 30 | Mgr | 77506.75 |
| 40 | Sales | 78006.00 |

**NEW_STAFF**

| ID | SALARY |
|---|---|
| 30 | 7750.67 |
| 40 | 7800.60 |
| 50 | 8065.98 |

**AFTER-MERGE**

| ID | JOB | SALARY |
|---|---|---|
| 20 | Sales | 78171.25 |
| 40 | ? | 7800.60 |
| 40 | Sales | 78006.00 |

Observe that the above merge did the following:

- The target row with an ID of 30 was deleted - because it matched.
- The target row with an ID of 40 was not deleted, because it was excluded in the fullselect that was done before the merge.

- The source row with an ID of 40 was inserted, because it was not found in the target fullselect. This is why the base table now has two rows with an ID of 40.
- The source row with an ID of 50 was not inserted, because it was excluded in the fullselect that was done before the merge.

### 3.5.6. Listing Columns

The next example explicitly lists the target fields in the insert statement - so they correspond to those listed in the following values phrase:

*Listing columns and values in insert*

```
MERGE INTO old_staff oo
USING new_staff nn
ON oo.id = nn.id
WHEN MATCHED THEN
    UPDATE
    SET (salary, job) = (1234, '?')
WHEN NOT MATCHED THEN
    INSERT (id,salary,job)
    VALUES (id,5678.9,'?');
```

**AFTER-MERGE**

| ID | JOB | SALARY |
|---|---|---|
| 20 | Sales | 78171.25 |
| 30 | ? | 1234.00 |
| 40 | ? | 1234.00 |
| 50 | ? | 5678.90 |

# Chapter 4. Compound SQL

A compound statement groups multiple independent SQL statements into a single executable. In addition, simple processing logic can be included to create what is, in effect, a very basic program. Such statements can be embedded in triggers, SQL functions, SQL methods, and dynamic SQL statements.

## 4.1. Introduction

A compound SQL statement begins with an (optional) name, followed by the variable declarations, followed by the procedural logic.

Below is a compound statement that reads a set of rows from the STAFF table and, for each row fetched, updates the COMM field to equal the current fetch number.

*Sample Compound SQL statement*

```
BEGIN ATOMIC
    DECLARE cntr SMALLINT DEFAULT 1;
    FOR V1 AS
        SELECT id as idval
        FROM staff
        WHERE id < 80
        ORDER BY id
    DO
        UPDATE staff
            SET comm = cntr
            WHERE id = idval;
        SET cntr = cntr + 1;
    END FOR;
END
```

### 4.1.1. Statement Delimiter

Db2 SQL does not come with a designated statement delimiter (terminator), though a semicolon is typically used. However, a semi-colon cannot be used in a compound SQL statement because that character is used to differentiate the sub-components of the statement. In Db2BATCH, one can run the SET DELIMITER command (intelligent comment) to use something other than a semi-colon. The following script illustrates this usage:

*Set Delimiter example*

```
--#SET DELIMITER !
SELECT NAME FROM STAFF WHERE id = 10!
--#SET DELIMITER ;
SELECT NAME FROM STAFF WHERE id = 20;
```

In the Db2 command processor one can do the same thing using the terminator keyword:

*Set Terminator example*

```
--#SET TERMINATOR !
SELECT NAME FROM STAFF WHERE id = 10!
--#SET TERMINATOR ;
SELECT NAME FROM STAFF WHERE id = 20;
```

### 4.1.2. SQL Statement Usage

When used in dynamic SQL, the following control statements can be used:

- FOR statement
- GET DIAGNOSTICS statement
- IF statement
- ITERATE statement
- LEAVE statement
- SIGNAL statement
- WHILE statement

There are many more PSM (persistent stored modules) control statements than what is shown above. But only these ones can be used in Compound SQL statements.

The following SQL statements can be issued:

- fullselect
- UPDATE
- DELETE
- INSERT
- SET variable statement

### 4.1.3. DECLARE Variables

All variables have to be declared at the start of the compound statement. Each variable must be given a name and a type and, optionally, a default (start) value.

*DECLARE examples*

```
BEGIN ATOMIC
    DECLARE aaa, bbb, ccc SMALLINT DEFAULT 1;
    DECLARE ddd CHAR(10) DEFAULT NULL;
    DECLARE eee INTEGER;
    SET eee = aaa + 1;
    UPDATE staff
    SET comm   = aaa
      , salary = bbb
      , years  = eee
    WHERE id = 10;
END
```

**FOR Statement**

The FOR statement executes a group of statements for each row fetched from a query.

In the next example one row is fetched per year of service (for selected years) in the STAFF table. That row is then used to do two independent updates to the three matching rows:

*FOR statement example*

```
BEGIN ATOMIC
    FOR V1 AS
        SELECT years   AS yr_num
             , max(id) AS max_id
        FROM staff
        WHERE years < 4
        GROUP BY years
        ORDER BY years
    DO
        UPDATE staff
            SET salary = salary / 10
            WHERE id = max_id;
        UPDATE staff
            set comm = 0
            WHERE years = yr_num;
    END FOR;
END
```

**BEFORE**

| ID | SALARY | COMM |
|---|---|---|
| 180 | 37009.75 | 236.50 |
| 230 | 83369.80 | 189.65 |
| 330 | 49988.00 | 55.50 |

**AFTER**

| ID | SALARY | COMM |
|---|---|---|
| 180 | 37009.75 | 0.00 |
| 230 | 8336.98 | 0.00 |
| 330 | 4998.80 | 0.00 |

### 4.1.4. GET DIAGNOSTICS Statement

The GET DIAGNOSTICS statement returns information about the most recently run SQL statement. One can either get the number of rows processed (i.e. inserted, updated, or deleted), or the return status (for an external procedure call).

In the example below, some number of rows are updated in the STAFF table. Then the count of rows updated is obtained, and used to update a row in the STAFF table:

*GET DIAGNOSTICS statement example*

```
BEGIN ATOMIC
    DECLARE numrows INT DEFAULT 0;
    UPDATE staff
    SET salary = 12345
    WHERE id < 100;
    GET DIAGNOSTICS numrows = ROW_COUNT;
    UPDATE staff
        SET salary = numrows
        WHERE id = 10;
END
```

### 4.1.5. IF Statement

The IF statement is used to do standard if-then-else branching logic. It always begins with an IF THEN statement and ends with and END IF statement.

The next example uses if-then-else logic to update one of three rows in the STAFF table, depending on the current timestamp value:

*IF statement example*

```
BEGIN ATOMIC
    DECLARE cur INT;
    SET cur = MICROSECOND(CURRENT TIMESTAMP);
    IF cur > 600000 THEN
        UPDATE staff
            SET name = CHAR(cur)
            WHERE id = 10;
    ELSEIF cur > 300000 THEN
        UPDATE staff
            SET name = CHAR(cur)
            WHERE id = 20;
    ELSE
        UPDATE staff
            SET name = CHAR(cur)
            WHERE id = 30;
    END IF;
END
```

### 4.1.6. ITERATE Statement

The ITERATE statement causes the program to return to the beginning of the labeled loop.

In next example, the second update statement will never get performed because the ITERATE will always return the program to the start of the loop:

*ITERATE statement example*

```
BEGIN ATOMIC
    DECLARE cntr INT DEFAULT 0;
    whileloop:
    WHILE cntr < 60 DO
        SET cntr = cntr + 10;
        UPDATE staff
            SET salary = cntr
            WHERE id = cntr;
        ITERATE whileloop;
        UPDATE staff
            SET comm = cntr + 1
            WHERE id = cntr;
    END WHILE;
END
```

### 4.1.7. LEAVE Statement

The LEAVE statement exits the labeled loop.

In the next example, the WHILE loop would continue forever, if left to its own devices. But after

some random number of iterations, the LEAVE statement will exit the loop:

*LEAVE statement example*

```
BEGIN ATOMIC
    DECLARE cntr INT DEFAULT 1;
    whileloop:
    WHILE 1 <> 2 DO
        SET cntr = cntr + 1;
        IF RAND() > 0.99 THEN
            LEAVE whileloop;
        END IF;
    END WHILE;
    UPDATE staff
        SET salary = cntr
        WHERE id = 10;
END
```

### 4.1.8. SIGNAL Statement

The SIGNAL statement is used to issue an error or warning message.

The next example loops a random number of times, and then generates an error message using the SIGNAL command, saying how many loops were done:

*SIGNAL statement example*

```
BEGIN ATOMIC
    DECLARE cntr INT DEFAULT 1;
    DECLARE emsg CHAR(20);
    whileloop:
    WHILE RAND() < .99 DO
        SET cntr = cntr + 1;
    END WHILE;
    SET emsg = '#loops: ' || CHAR(cntr);
    SIGNAL SQLSTATE '75001' SET MESSAGE_TEXT = emsg;
END
```

### 4.1.9. WHILE Statement

The WHILE statement repeats one or more statements while some condition is true.

The next statement has two nested WHILE loops, and then updates the STAFF table:

*WHILE statement example*

```
BEGIN ATOMIC
    DECLARE c1, C2 INT DEFAULT 1;
    WHILE c1 < 10 DO
        WHILE c2 < 20 DO
            SET c2 = c2 + 1;
        END WHILE;
        SET c1 = c1 + 1;
    END WHILE;
    UPDATE staff
        SET salary = c1
          , comm = c2
        WHERE id = 10;
END
```

### 4.1.10. Other Usage

The following Db2 objects also support the language elements described above:

- Triggers
- Stored procedures
- User-defined functions
- Embedded compound SQL (in programs).

Some of the above support many more language elements. For example stored procedures that are written in SQL also allow the following: `ASSOCIATE, CASE, GOTO, LOOP, REPEAT, RESIGNAL` , and `RETURN` .

### 4.1.11. Test Query

To illustrate some of the above uses of compound SQL, we are going to get from the STAFF table a complete list of departments, and the number of rows in each department. Here is the basic query, with the related answer:

*List departments in STAFF table*

```
SELECT dept
     , count(*) as #rows
FROM staff
GROUP BY dept
ORDER BY dept;
```

**ANSWER**

| DEPT | #ROWS |
|---|---|
| 10 | 4 |

| DEPT | #ROWS |
|---|---|
| 15 | 4 |
| 20 | 4 |
| 38 | 5 |
| 42 | 4 |
| 51 | 5 |
| 66 | 5 |
| 84 | 4 |

If all you want to get is this list, the above query is the way to go. But we will get the same answer using various other methods, just to show how it can be done using compound SQL statements.

**Trigger**

One cannot get an answer using a trigger. All one can do is alter what happens during an insert, update, or delete. With this in mind, the following example does the following:

- Sets the statement delimiter to an "!". Because we are using compound SQL inside the trigger definition, we cannot use the usual semi-colon.
- Creates a new table (note: triggers are not allowed on temporary tables).
- Creates an INSERT trigger on the new table. This trigger gets the number of rows per department in the STAFF table - for each row (department) inserted.
- Inserts a list of departments into the new table.
- Selects from the new table.

Now for the code:

*Trigger with compound SQL*

```
--#SET DELIMITER !
CREATE TABLE dpt
( dept   SMALLINT NOT NULL
, #names SMALLINT
, PRIMARY KEY(dept))!
COMMIT!

CREATE TRIGGER dpt1 AFTER INSERT ON dpt
REFERENCING NEW AS NNN
FOR EACH ROW
MODE Db2SQL
BEGIN ATOMIC
    DECLARE namecnt SMALLINT DEFAULT 0;
    FOR getnames AS
        SELECT COUNT(*) AS #n
        FROM staff
        WHERE dept = nnn.dept
    DO
        SET namecnt = #n;
    END FOR;
    UPDATE dpt
    SET #names = namecnt
    WHERE dept = nnn.dept;
    END!
COMMIT!

INSERT INTO dpt (dept)
    SELECT DISTINCT dept
    FROM staff!
COMMIT!
SELECT *
FROM dpt
ORDER BY dept!
```

This example uses an "!" as the stmt delimiter.

**ANSWER**

| DEPT | #NAMES |
|---|---|
| 10 | 4 |
| 15 | 4 |
| 20 | 4 |
| 38 | 5 |
| 42 | 4 |
| 51 | 5 |

| DEPT | #NAMES |
|---|---|
| 66 | 5 |
| 84 | 4 |

The above code was designed to be run in Db2BATCH. The "set delimiter" notation will probably not work in other environments.

**Scalar Function**

One can do something very similar to the above that is almost as stupid using a user-defined scalar function, that calculates the number of rows in a given department. The basic logic will go as follows:

- Set the statement delimiter to an "!".
- Create the scalar function.
- Run a query that first gets a list of distinct departments, then calls the function.

Here is the code:

*Scalar Function with compound SQL*

```
--#SET DELIMITER !
CREATE FUNCTION dpt1 (deptin SMALLINT)
RETURNS SMALLINT
BEGIN ATOMIC
    DECLARE num_names SMALLINT;
    FOR getnames AS
        SELECT COUNT(*) AS #n
        FROM staff
        WHERE dept = deptin
    DO
        SET num_names = #n;
    END FOR;
    RETURN num_names;
END!

COMMIT!

SELECT XXX.*
     , dpt1(dept) as #names
FROM
    (SELECT dept
     FROM staff
     GROUP BY dept
    ) AS XXX
ORDER BY dept!
```

This example uses an "!" as the stmt delimiter.

**ANSWER**

| DEPT | #NAMES |
|---|---|
| 10 | 4 |
| 15 | 4 |
| 20 | 4 |
| 38 | 5 |
| 42 | 4 |
| 51 | 5 |
| 66 | 5 |
| 84 | 4 |

Because the query used in the above function will only ever return one row, we can greatly simplify the function definition thus:

*Scalar Function with compound SQL*

```
--#SET DELIMITER !
CREATE FUNCTION dpt1 (deptin SMALLINT)
RETURNS SMALLINT
BEGIN ATOMIC
    RETURN
        SELECT COUNT(*)
        FROM staff
        WHERE dept = deptin;
END!
COMMIT!
```

This example uses an "!" as the stmt delimiter.

```
SELECT XXX.*
     , dpt1(dept) as #names
FROM
    (SELECT dept
     FROM staff
     GROUP BY dept
) AS XXX
ORDER BY dept!
```

In the above example, the RETURN statement is directly finding the one matching row, and then returning it to the calling statement.

### Table Function

Below is almost exactly the same logic, this time using a table function:

Table Function with compound SQL

```
--#SET DELIMITER !
CREATE FUNCTION dpt2 ()
RETURNS TABLE ( dept    SMALLINT
              , #names  SMALLINT)
BEGIN ATOMIC
    RETURN
        SELECT dept
             , count(*)
        FROM staff
        GROUP BY dept
        ORDER BY dept;
END!

COMMIT!

--#SET DELIMITER ;
SELECT *
FROM TABLE(dpt2()) T1
ORDER BY dept;
```

This example uses an "!" as the stmt delimiter.

**ANSWER**

| DEPT | #NAMES |
|---|---|
| 10 | 4 |
| 15 | 4 |
| 20 | 4 |
| 38 | 5 |
| 42 | 4 |
| 51 | 5 |
| 66 | 5 |
| 84 | 4 |

*Sample Compound SQL statement*

```
BEGIN ATOMIC
  DECLARE cntr SMALLINT DEFAULT 1;
  FOR V1 AS
    SELECT id as idval
    FROM staff
    WHERE id < 80
    ORDER BY id
  DO
    UPDATE staff SET comm = cntr
    WHERE id = idval;
    SET cntr = cntr + 1;
  END FOR;
END
```

# Chapter 5. Column Functions or Aggregate Functions

By themselves, column functions work on the complete set of matching rows. One can use a GROUP BY expression to limit them to a subset of matching rows. One can also use them in an OLAP function to treat individual rows differently.

Be very careful when using either a column function, or the DISTINCT clause, in a join. If the join is incorrectly coded, and does some form of Cartesian Product, the column function may get rid of the all the extra (wrong) rows so that it becomes very hard to confirm that the answer is incorrect. Likewise, be appropriately suspicious whenever you see that someone (else) has used a DISTINCT statement in a join. Sometimes, users add the DISTINCT clause to get rid of duplicate rows that they didn't anticipate and don't understand.

## 5.1. Column Functions, Definitions

### 5.1.1. ARRAY_AGG

Aggregate the set of elements in an array. If an ORDER BY is provided, it determines the order in which the elements are entered into the array.

### 5.1.2. AVG

Get the average (mean) value of a set of non-null rows. The columns(s) must be numeric. ALL is the default. If DISTINCT is used duplicate values are ignored. If no rows match, the null value is returned.

*AVG function examples*

```
SELECT AVG(dept)          AS a1
     , AVG(ALL dept)      AS a2
     , AVG(DISTINCT dept) AS a3
     , AVG(dept/10)       AS a4
     , AVG(dept)/10       AS a5
FROM staff
HAVING AVG(dept) > 40;
```

*ANSWER*

| A1 | A2 | A3 | A4 | A5 |
|---|---|---|---|---|
| 41 | 41 | 40 | 3 | 4 |

Observe columns A4 and A5 above. Column A4 has the average of each value divided by 10. Column A5 has the average of all of the values divided by 10. In the former case, precision has been lost due to rounding of the original integer value and the result is arguably incorrect. This problem also occurs when using the SUM function.

### Averaging Null and Not-Null Values

Some database designers have an intense and irrational dislike of using nullable fields. What they do instead is define all columns as not-null and then set the individual fields to zero (for numbers) or blank (for characters) when the value is unknown. This solution is reasonable in some situations, but it can cause the AVG function to give what is arguably the wrong answer. One solution to this problem is some form of counseling or group therapy to overcome the phobia. Alternatively, one can use the CASE expression to put null values back into the answer-set being processed by the AVG function. The following SQL statement uses a modified version of the IBM sample STAFF table (all null COMM values were changed to zero) to illustrate the technique:

*Convert zero to null before doing AVG*

```
UPDATE staff
SET comm = 0
WHERE comm IS NULL;

SELECT AVG(salary) AS salary
     , AVG(comm)   AS comm1
     , AVG(CASE comm
               WHEN 0 THEN NULL
               ELSE comm
           END) AS comm2
FROM staff;
```

*ANSWER*

| SALARY | COMM1 | COMM2 |
|---|---|---|
| 67932.78 | 351.98 | 513.31 |

```
UPDATE staff
SET comm = NULL
WHERE comm = 0;
```

The COMM2 field above is the correct average. The COMM1 field is incorrect because it has factored in the zero rows with really represent null values. Note that, in this particular query, one cannot use a WHERE to exclude the "zero" COMM rows because it would affect the average salary value.

### Dealing with Null Output

The AVG, MIN, MAX, and SUM functions almost always return a null value when there are no

matching rows (see No Rows Match for exceptions). One can use the COALESCE function, or a CASE expression, to convert the null value into a suitable substitute. Both methodologies are illustrated below:

*Convert null output (from AVG) to zero*

```
SELECT COUNT(*)                AS c1
     , AVG(salary)             AS a1
     , COALESCE(AVG(salary),0) AS a2
     , CASE
         WHEN AVG(salary) IS NULL THEN 0
         ELSE AVG(salary)
       END                     AS a3
FROM staff
WHERE id < 10;
```

*ANSWER*

| C1 | A1 | A2 | A3 |
|---|---|---|---|
| 0 | - | 0 | 0 |

**AVG Date/Time Values**

The AVG function only accepts numeric input. However, one can, with a bit of trickery, also use the AVG function on a date field. First convert the date to the number of days since the start of the Current Era, then get the average, then convert the result back to a date. Please be aware that, in many cases, the average of a date does not really make good business sense. Having said that, the following SQL gets the average birth-date of all employees:

*AVG of date column*

```
SELECT AVG(DAYS(birthdate))
     , DATE(AVG(DAYS(birthdate)))
FROM employee;
```

*ANSWER*

| 1 | 2 |
|---|---|
| 721092 | 1975-04-14 |

Time data can be manipulated in a similar manner using the MIDNIGHT_SECONDS function. If one is really desperate (or silly), the average of a character field can also be obtained using the ASCII and CHR functions.

**Average of an Average**

In some cases, getting the average of an average gives an overflow error. Inasmuch as you shouldn't do this anyway, it is no big deal:

*Select average of average*

```
SELECT AVG(avg_sal) AS avg_avg
FROM (SELECT dept
           , AVG(salary) AS avg_sal
     FROM staff
     GROUP BY dept
    ) AS xxx;
```

ANSWER: Overflow error

### 5.1.3. CORRELATION

I don't know a thing about statistics, so I haven't a clue what this function does. But I do know that the SQL Reference is wrong - because it says the value returned will be between 0 and 1. I found that it is between -1 and +1 (see below). The output type is float.

*CORRELATION function examples*

```
WITH temp1(col1, col2, col3, col4) AS
(VALUES (0, 0, 0, RAND(1))
 UNION ALL
 SELECT col1 + 1
      , col2 - 1
      , RAND()
      , RAND()
 FROM temp1
 WHERE col1 < = 1000
 )
SELECT DEC(CORRELATION(col1, col1), 5, 3) AS cor11
     , DEC(CORRELATION(col1, col2), 5, 3) AS cor12
     , DEC(CORRELATION(col2, col3), 5, 3) AS cor23
     , DEC(CORRELATION(col3, col4), 5, 3) AS cor34
FROM temp1;
```

*ANSWER*

| COR11 | COR12 | COR23 | COR34 |
|---|---|---|---|
| 1.000 | -1.000 | -0.017 | -0.005 |

### 5.1.4. COUNT

Get the number of values in a set of rows. The result is an integer. The value returned depends upon the options used:

- COUNT(*) gets a count of matching rows.
- COUNT(expression) gets a count of rows with a non-null expression value.
- COUNT(ALL expression) is the same as the COUNT(expression) statement.

- COUNT(DISTINCT expression) gets a count of distinct non-null expression values.

*COUNT function examples*

```
SELECT COUNT(*)                    AS c1
     , COUNT(INT(comm/10))         AS c2
     , COUNT(ALL INT(comm/10))     AS c3
     , COUNT(DISTINCT INT(comm/10)) AS c4
     , COUNT(DISTINCT INT(comm))   AS c5
     , COUNT(DISTINCT INT(comm))/10 AS c6
FROM staff;
```

*ANSWER*

| C1 | C2 | C3 | C4 | C5 | C6 |
|---|---|---|---|---|---|
| 35 | 24 | 24 | 19 | 24 | 2 |

There are 35 rows in the STAFF table (see C1 above), but only 24 of them have non-null commission values (see C2 above). If no rows match, the COUNT returns zero - except when the SQL statement also contains a GROUP BY. In this latter case, the result is no row.

*COUNT function with and without GROUP BY*

```
SELECT 'NO GP-BY' AS c1
     , COUNT(*)   AS c2
FROM staff
WHERE id = -1
UNION
SELECT 'GROUP-BY' AS c1
     , COUNT(*)   AS c2
FROM staff
WHERE id = -1
GROUP BY dept;
```

*ANSWER*

| C1 | C2 |
|---|---|
| NO GP-BY | 0 |

### 5.1.5. COUNT_BIG

Get the number of rows or distinct values in a set of rows. Use this function if the result is too large for the COUNT function. The result is of type decimal 31. If the DISTINCT option is used both duplicate and null values are eliminated. If no rows match, the result is zero.

*COUNT_BIG function examples*

```
SELECT COUNT_BIG(*)                AS c1
     , COUNT_BIG(dept)             AS c2
     , COUNT_BIG(DISTINCT dept)    AS c3
     , COUNT_BIG(DISTINCT dept/10) AS c4
     , COUNT_BIG(DISTINCT dept)/10 AS c5
FROM STAFF;
```

*ANSWER*

| C1 | C2 | C3 | C4 | C5 |
|---|---|---|---|---|
| 35. | 35. | 8. | 7. | 0. |

### 5.1.6. COVARIANCE

Returns the covariance of a set of number pairs. The output type is float.

*COVARIANCE function examples*

```
WITH temp1(c1, c2, c3, c4) AS
(VALUES (0 , 0 , 0 , RAND(1))
 UNION ALL
 SELECT c1 + 1
      , c2 - 1
      , RAND()
      , RAND()
FROM temp1
WHERE c1 <= 1000
)
SELECT DEC(COVARIANCE(c1,c1),6,0) AS cov11
     , DEC(COVARIANCE(c1,c2),6,0) AS cov12
     , DEC(COVARIANCE(c2,c3),6,4) AS cov23
     , DEC(COVARIANCE(c3,c4),6,4) AS cov34
FROM temp1;
```

*ANSWER*

| COV11 | COV12 | COV23 | COV34 |
|---|---|---|---|
| 83666. | -83666. | -1.4689 | -0.0004 |

### 5.1.7. COVARIANCE_SAMP

Returns the sample covariance of a set of number pairs.

### 5.1.8. CUME_DIST

Returns the cumulative distribution of a row that is hypothetically inserted into a group of rows.

### 5.1.9. GROUPING

The GROUPING function is used in CUBE, ROLLUP, and GROUPING SETS statements to identify what rows come from which particular GROUPING SET. A value of 1 indicates that the corresponding data field is null because the row is from of a GROUPING SET that does not involve this row. Otherwise, the value is zero.

*GROUPING function example*

```
SELECT dept
     , AVG(salary)    AS salary
     , GROUPING(dept) AS df
FROM staff
GROUP BY ROLLUP(dept)
ORDER BY dept;
```

*ANSWER*

| DEPT | SALARY | DF |
|---|---|---|
| 10 | 83365.86 | 0 |
| 15 | 60482.33 | 0 |
| 20 | 63571.52 | 0 |
| 38 | 60457.11 | 0 |
| 42 | 49592.26 | 0 |
| 51 | 83218.16 | 0 |
| 66 | 73015.24 | 0 |
| 84 | 66536.75 | 0 |
| - | 67932.78 | 1 |

See the section titled "Group By and Having" for more information on this function.

### 5.1.10. LISTAGG

Aggregates a set of string elements into one string by concatenating the strings. Optionally, a separator string can be provided which is inserted between contiguous input strings.

### 5.1.11. MAX

Get the maximum value of a set of rows. The use of the DISTINCT option has no affect. If no rows match, the null value is returned.

*MAX function examples*

```
SELECT MAX(dept)
     , MAX(ALL dept)
     , MAX(DISTINCT dept)
     , MAX(DISTINCT dept/10)
FROM staff;
```

*ANSWER*

| 1 | 2 | 3 | 4 |
|---|---|---|---|
| 84 | 84 | 84 | 8 |

**MAX and MIN usage with Scalar Functions**

Several Db2 scalar functions convert a value from one format to another, for example from numeric to character. The function output format will not always shave the same ordering sequence as the input. This difference can affect MIN, MAX, and ORDER BY processing.

*MAX function with dates*

```
SELECT MAX(hiredate)
     , CHAR(MAX(hiredate),USA)
     , MAX(CHAR(hiredate,USA))
FROM employee;
```

*ANSWER*

| 1 | 2 | 3 |
|---|---|---|
| 2006-12-15 | 12/15/2006 | 12/15/2006 |

In the above the SQL, the second field gets the MAX before doing the conversion to character whereas the third field works the other way round. In most cases, the later is wrong. In the next example, the MAX function is used on a small integer value that has been converted to character. If the CHAR function is used for the conversion, the output is left justified, which results in an incorrect answer. The DIGITS output is correct (in this example).

*MAX function with numbers, 1 of 2*

```
SELECT MAX(id)         AS id
     , MAX(CHAR(id))   AS chr
     , MAX(DIGITS(id)) AS dig
FROM staff;
```

*ANSWER*

| ID | CHR | DIG |
|---|---|---|
| 350 | 90 | 00350 |

The DIGITS function can also give the wrong answer - if the input data is part positive and part negative. This is because this function does not put a sign indicator in the output.

*MAX function with numbers, 2 of 2*

```
SELECT MAX(id - 250)         AS id
     , MAX(CHAR(id - 250))   AS chr
     , MAX(DIGITS(id - 250)) AS dig
FROM staff;
```

*ANSWER*

| D | CHR | DIG |
|---|---|---|
| 100 | 90 | 0000000240 |

Be careful when using a column function on a field that has been converted from number to character, or from date/time to character. The result may not be what you intended.

### 5.1.12. MEDIAN

Returns the median value in a set of values.

### 5.1.13. MIN

Get the minimum value of a set of rows. The use of the DISTINCT option has no affect. If no rows match, the null value is returned.

*MIN function examples*

```
SELECT MIN(dept)
     , MIN(ALL dept)
     , MIN(DISTINCT dept)
     , MIN(DISTINCT dept/10)
FROM staff;
```

*ANSWER*

| 1 | 2 | 3 | 4 |
|---|---|---|---|
| 10 | 10 | 10 | 1 |

### 5.1.14. PERCENTILE_CONT

Returns the value that corresponds to the specified percentile given a sort specification by using a continuous distribution model.

### 5.1.15. PERCENTILE_DISC

Returns the value that corresponds to the specified percentile given a sort specification by using a discrete distribution model.

### 5.1.16. PERCENT_RANK

Returns the relative percentile rank of a row that is hypothetically inserted into a group of rows.

### 5.1.17. Regression Functions

The various regression functions support the fitting of an ordinary-least-squares regression line of the form y = a * x + b to a set of number pairs.

**REGR_AVGX** returns a quantity that than can be used to compute the validity of the regression model. The output is of type float.

**REGR_AVGY** (see REGR_AVGX).

**REGR_COUNT** returns the number of matching non-null pairs. The output is integer.

**REGR_INTERCEPT** returns the y-intercept of the regression line.

**REGR_R2** returns the coefficient of determination for the regression.

**REGR_SLOPE** returns the slope of the line.

**REGR_SXX** (see REGR_AVGX).

**REGR_SXY** (see REGR_AVGX).

**REGR_SYY** (see REGR_AVGX).

**See the IBM SQL Reference for more details on the above functions.**

*REGRESSION functions examples*

```
SELECT DEC(REGR_SLOPE(bonus,salary),7,5)     AS r_slope
     , DEC(REGR_INTERCEPT(bonus,salary),7,3) AS r_icpt
     , INT(REGR_COUNT(bonus,salary))         AS r_count
     , INT(REGR_AVGX(bonus,salary))          AS r_avgx
     , INT(REGR_AVGY(bonus,salary))          AS r_avgy
     , DEC(REGR_SXX(bonus,salary),10)        AS r_sxx
     , INT(REGR_SXY(bonus,salary))           AS r_sxy
     , INT(REGR_SYY(bonus,salary))           AS r_syy
FROM employee
WHERE workdept = 'A00';
```

*ANSWERS*

| r_slope | r_icpt | r_count | r_avgx | r_avgy | r_sxx | r_sxy | r_syy |
|---|---|---|---|---|---|---|---|
| 0.00247 | 644.862 | 5 | 70850 | 820 | 8784575000 | 21715000 | 168000 |

### 5.1.18. STDDEV

Get the standard deviation of a set of numeric values. If DISTINCT is used, duplicate values are ignored. If no rows match, the result is null. The output format is double.

*STDDEV function examples*

```
SELECT AVG(dept)                AS a1
,STDDEV(dept)                   AS s1
,DEC(STDDEV(dept),3,1)          AS s2
,DEC(STDDEV(ALL dept),3,1)      AS s3
,DEC(STDDEV(DISTINCT dept),3,1) AS s4
FROM staff;
```

*ANSWER*

| A1 | S1 | S2 | S3 | S4 |
|---|---|---|---|---|
| 41 | +2.3522355E+1 | 23.5 | 23.5 | 24.1 |

### 5.1.19. STDDEV_SAMP

The STDDEV_SAMP function returns the sample standard deviation (division by [n-1]) of a set of numbers.

### 5.1.20. SUM

Get the sum of a set of numeric values. If DISTINCT is used, duplicate values are ignored. Null values are always ignored. If no rows match, the result is null.

*SUM function examples*

```
SELECT SUM(dept)          AS s1
     , SUM(ALL dept)      AS s2
     , SUM(DISTINCT dept) AS s3
     , SUM(dept/10)       AS s4
     , SUM(dept)/10       AS s5
```

FROM staff;

*ANSWER*

| S1 | S2 | S3 | S4 | S5 |
|---|---|---|---|---|
| 1459 | 1459 | 326 | 134 | 145 |

The answers S4 and S5 above are different. This is because the division is done before the SUM in column S4, and after in column S5. In the former case, precision has been lost due to rounding of the original integer value and the result is arguably incorrect. When in doubt, use the S5 notation.

### 5.1.21. VAR or VARIANCE

Get the variance of a set of numeric values. If DISTINCT is used, duplicate values are ignored. If no rows match, the result is null. The output format is double.

*VARIANCE function examples*

```
SELECT AVG(dept)                       AS a1
     , VARIANCE(dept)                  AS s1
     , DEC(VARIANCE(dept),4,1)         AS s2
     , DEC(VARIANCE(ALL dept),4,1)     AS s3
     , DEC(VARIANCE(DISTINCT dept),4,1) AS s4
FROM staff;
```

*ANSWER*

| A1 | V1 | V2 | V3 | V4 |
|---|---|---|---|---|
| 41 | +5.533012244E+2 | 553 | 553 | 582 |

### 5.1.22. VARIANCE_SAMP

Returns the sample variance (division by [n-1]) of a set of numbers.

### 5.1.23. XMLAGG

Returns an XML sequence containing an item for each non-null value in a set of XML values.

### 5.1.24. XMLGROUP

The XMLGROUP function returns an XML value with a single XQuery document node containing one top-level element node. This is an aggregate expression that will return a single-rooted XML document from a group of rows where each row is mapped to a row subelement.

# Chapter 6. OLAP Functions

=== Introduction

Online Analytical Processing (OLAP) functions enable one to sequence and rank query rows. They are especially useful when the calling program is very simple.

## 6.1. The Bad Old Days

To really appreciate the value of the OLAP functions, one should try to do some seemingly trivial task without them. To illustrate this point, consider the following query:

*Select rows from STAFF table*

```
SELECT s1.job
     , s1.id
     , s1.salary
FROM staff s1
WHERE s1.name LIKE '%s%'
AND   s1.id   <    90
ORDER BY s1.job
       , s1.id;
```

*ANSWER*

| JOB | ID | SALARY |
|---|---|---|
| Clerk | 80 | 43504.60 |
| Mgr | 10 | 98357.50 |
| Mgr | 50 | 80659.80 |

Let us now add two fields to this query:

- A running sum of the salaries selected.
- A running count of the rows retrieved.

Adding these fields is easy - when using OLAP functions:

*Using OLAP functions to get additional fields*

```
SELECT s1.job
     , s1.id
     , s1.salary
     , SUM(salary) OVER(ORDER BY job, id)  AS sumsal
     , ROW_NUMBER() OVER(ORDER BY job, id) AS r
FROM staff s1
WHERE s1.name LIKE '%s%'
AND   s1.id   <    90
ORDER BY s1.job
       , s1.id;
```

*ANSWER*

| JOB | ID | SALARY | SUMSAL | R |
|---|---|---|---|---|
| Clerk | 80 | 43504.60 | 43504.60 | 1 |
| Mgr | 10 | 98357.50 | 141862.10 | 2 |
| Mgr | 50 | 80659.80 | 222521.90 | 3 |

### 6.1.1. Write Query without OLAP Functions

If one does not have OLAP functions, one can still get the required answer, but the code is quite tricky. The problem is that this seemingly simple query contains two nasty tricks:

- Not all of the rows in the table are selected.
- The output is ordered on two fields, the first of which is not unique.

Below is the arguably the most elegant way to write the above query without using OLAP functions. There query has the following basic characteristics:

- Define a common-table-expression with the set of matching rows.
- Query from this common-table-expression.
- For each row fetched, do two nested select statements. The first gets a running sum of the salaries, and the second gets a running count of the rows retrieved.

Now for the code:

*Running counts without OLAP functions*

```
WITH temp1 AS
(SELECT *
 FROM staff s1
 WHERE s1.name LIKE '%s%'
 AND   s1.id   <    90
)
SELECT s1.job
     , s1.id
     , s1.salary
     , (SELECT SUM(s2.salary)
        FROM temp1 s2
        WHERE (s2.job < s1.job)
        OR    (s2.job = s1.job AND s2.id <= s1.id)
       ) AS sumsal
    , (SELECT COUNT(*)
       FROM temp1 s2
       WHERE (s2.job < s1.job)
       OR    (s2.job = s1.job AND s2.id <= s1.id)
      ) AS r
FROM temp1 s1
ORDER BY s1.job
       , s1.id;
```

*ANSWER*

| JOB | ID | SALARY | SUMSAL | R |
|---|---|---|---|---|
| Clerk | 80 | 43504.60 | 43504.60 | 1 |
| Mgr | 10 | 98357.50 | 141862.10 | 2 |
| Mgr | 50 | 80659.80 | 222521.90 | 3 |

### 6.1.2. Concepts

Below are some of the basic characteristics of OLAP functions:

- OLAP functions are column functions that work (only) on the set of rows that match the predicates of the query.
- Unlike ordinarily column functions, (e.g. SUM), OLAP functions do not require that the whole answer-set be summarized. In fact, OLAP functions never change the number of rows returned by the query.
- OLAP functions work on sets of values, but the result is always a single value.
- OLAP functions are used to return individual rows from a table (e.g. about each staff member), along with related summary data (e.g. average salary in department).
- OLAP functions are often applied on some set (i.e. of a moving window) of rows that is defined relative to the current row being processed. These matching rows are classified using an ORDER

BY as being one of three types:

  - **Preceding** rows are those that have already been processed.
  - **Following** rows are those that have yet to be processed.
  - **Current row** is the one currently being processed.

- The ORDER BY used in an OLAP function is not related to the ORDER BY expression used to define the output order of the final answer set.
- OLAP functions can summarize the matching rows by a subset (i.e. partition). When this is done, it is similar to the use of a GROUP BY in an ordinary column function.

Below is a query that illustrates these concepts. It gets some individual rows from the STAFF table, while using an OLAP function to calculate a running average salary within the DEPT of the current row. The average is calculated using one preceding row (in ID order), the current row, and two following rows:

*Sample OLAP query*

```
SELECT dept
     , id
     , salary
     , DEC(AVG(salary) OVER(PARTITION BY dept
                            ORDER BY id
                            ROWS BETWEEN 1 PRECEDING
                                 AND     2 FOLLOWING)
           , 8, 2) AS avb_sal
FROM staff
WHERE dept IN (20, 38)
ORDER BY dept
       , id;
```

*ANSWER*

| DEPT | ID | SALARY | AVG_SAL |
|---|---|---|---|
| 20 | 10 | 98357.50 | 73344.45 |
| 20 | 20 | 78171.25 | 63571.52 |
| 20 | 80 | 43504.60 | 51976.20 |
| 20 | 190 | 34252.75 | 38878.67 |
| 38 | 30 | 77506.75 | 74107.01 |
| 38 | 40 | 78006.00 | 66318.95 |
| 38 | 60 | 66808.30 | 56194.70 |
| 38 | 120 | 42954.75 | 48924.26 |
| 38 | 180 | 37009.75 | 39982.25 |

*TABLE*

| DEPT | ID | SALARY | Matching? | Partition | Relation to row [38 60 66808.30 56194.70] |
|---|---|---|---|---|---|
| 15 | 110 | 42508.20 | N | - | - |
| 15 | 170 | 42258.50 | N | - | - |
| 20 | 10 | 98357.50 | Y | 1 | - |
| 20 | 20 | 78171.25 | Y | 1 | - |
| 20 | 80 | 43504.60 | Y | 1 | - |
| 20 | 190 | 34252.75 | Y | 1 | - |
| 38 | 30 | 77506.75 | Y | 2 | Preceding row |
| 38 | 40 | 78006.00 | Y | 2 | Preceding row |
| 38 | 60 | 66808.30 | Y | 2 | Current row |
| 38 | 120 | 42954.75 | Y | 2 | Following row |
| 38 | 180 | 37009.75 | Y | 2 | Following row |
| 42 | 90 | 38001.75 | N | - | - |
| 42 | 100 | 78352.80 | N | - | - |
| 42 | 130 | 40505.90 | N | - | - |

Below is another query that calculates various running averages:

*Sample OLAP query*

```
SELECT dept
     , id
     , salary
     , DEC(AVG(salary) OVER() ,8,2)                  AS avg1
     , DEC(AVG(salary) OVER(PARTITION BY dept) ,8,2) AS avg2
     , DEC(AVG(salary) OVER(PARTITION BY dept
                            ORDER BY id
                            ROWS UNBOUNDED PRECEDING)
          , 8, 2)                                    AS avg3
     , DEC(AVG(salary) OVER(PARTITION BY dept
                            ORDER BY id
                            ROWS BETWEEN 1 PRECEDING AND 2 FOLLOWING)
          , 8, 2)                                    AS avg4
FROM staff
WHERE dept IN (15,20)
AND   id   >  20
ORDER BY dept
       , id;
```

*ANSWER*

| DEPT | ID | SALARY | AVG1 | AVG2 | AVG3 | AVG4 |
|---|---|---|---|---|---|---|
| 15 | 50 | 80659.80 | 53281.11 | 60482.33 | 80659.80 | 66556.94 |
| 15 | 70 | 76502.83 | 53281.11 | 60482.33 | 78581.31 | 60482.33 |
| 15 | 110 | 42508.20 | 53281.11 | 60482.33 | 66556.94 | 53756.51 |
| 15 | 170 | 42258.50 | 53281.11 | 60482.33 | 60482.33 | 42383.35 |
| 20 | 80 | 43504.60 | 53281.11 | 38878.67 | 43504.60 | 38878.67 |
| 20 | 190 | 34252.75 | 53281.11 | 38878.67 | 38878.67 | 38878.67 |

- **AVG1**: An average of all matching rows
- **AVG2**: An average of all matching rows within a department.
- **AVG3**: An average of matching rows within a department, from the first matching row (ordered by ID), up to and including the current row.
- **AVG4**: An average of matching rows within a department, starting with one preceding row (i.e. the highest, ordered by ID), the current row, and the next two following rows.

### 6.1.3. PARTITION Expression

The PARTITION BY expression, which is optional, defines the set of rows that are used in each OLAP function calculation.

Below is a query that uses different partitions to average sets of rows:

*PARTITION BY examples*

```
SELECT id
     , dept
     , job
     , years
     , salary
     , DEC(AVG(salary) OVER(PARTITION BY dept) ,7,2)      AS dpt_avg
     , DEC(AVG(salary) OVER(PARTITION BY job) ,7,2)       AS job_avg
     , DEC(AVG(salary) OVER(PARTITION BY years/2) ,7,2)   AS yr2_avg
     , DEC(AVG(salary) OVER(PARTITION BY dept, job) ,7,2) AS d_j_avg
FROM staff
WHERE dept IN (15,20)
AND   id   >  20
ORDER BY id;
```

*ANSWER*

| ID | DEPT | JOB | YEARS | SALARY | DPT_AVG | JOB_AVG | YR2_AVG | D_J_AVG |
|---|---|---|---|---|---|---|---|---|
| 50 | 15 | Mgr | 10 | 80659.80 | 60482.33 | 80659.80 | 80659.80 | 80659.80 |
| 70 | 15 | Sales | 7 | 76502.83 | 60482.33 | 76502.83 | 76502.83 | 76502.83 |
| 80 | 20 | Clerk | - | 43504.60 | 38878.67 | 40631.01 | 43504.60 | 38878.67 |

| ID | DEPT | JOB | YEARS | SALARY | DPT_AVG | JOB_AVG | YR2_AVG | D_J_AVG |
|---|---|---|---|---|---|---|---|---|
| 110 | 15 | Clerk | 5 | 42508.20 | 60482.33 | 40631.01 | 42383.35 | 42383.35 |
| 170 | 15 | Clerk | 4 | 42258.50 | 60482.33 | 40631.01 | 42383.35 | 42383.35 |
| 190 | 20 | Clerk | 8 | 34252.75 | 38878.67 | 40631.01 | 34252.75 | 38878.67 |

### 6.1.4. PARTITION vs. GROUP BY

The PARTITION clause, when used by itself, returns a very similar result to a GROUP BY, except that like all OLAP functions, it does not remove the duplicate rows. To illustrate, below is a simple query that does a GROUP BY:

*Sample query using GROUP BY*

```
SELECT dept
     , SUM(years) AS sum
     , AVG(years) AS avg
     , COUNT(*)   AS row
FROM staff
WHERE id    BETWEEN 40 AND 120
AND   years IS NOT  NULL
GROUP BY dept;
```

*ANSWER*

| DEPT | SUM | AVG | ROW |
|---|---|---|---|
| 15 | 22 | 7 | 3 |
| 38 | 6 | 6 | 1 |
| 42 | 13 | 6 | 2 |

Below is a similar query that uses a PARTITION phrase. Observe that each value calculated is the same, but duplicate rows have not been removed:

*Sample query using PARTITION*

```
SELECT dept
     , SUM(years) OVER(PARTITION BY dept) AS sum
     , AVG(years) OVER(PARTITION BY dept) AS avg
     , COUNT(*)   OVER(PARTITION BY dept) AS row
FROM staff
WHERE id    BETWEEN 40 AND 120
AND   years IS NOT  NULL
ORDER BY dept;
```

*ANSWER*

| DEPT | SUM | AVG | ROW |
|---|---|---|---|
| 15 | 22 | 7 | 3 |
| 15 | 22 | 7 | 3 |
| 15 | 22 | 7 | 3 |
| 38 | 6 | 6 | 1 |
| 42 | 13 | 6 | 2 |
| 42 | 13 | 6 | 2 |

Below is a similar query that uses the PARTITION phrase, and then uses a DISTINCT clause to remove the duplicate rows:

*Sample query using PARTITION and DISTINCT*

```
SELECT DISTINCT dept
     , SUM(years) OVER(PARTITION BY dept) AS sum
     , AVG(years) OVER(PARTITION BY dept) AS avg
     , COUNT(*)   OVER(PARTITION BY dept) AS row
FROM staff
WHERE id    BETWEEN 40 AND 120
AND   years IS NOT  NULL
ORDER BY dept;
```

*ANSWER*

| DEPT | SUM | AVG | ROW |
|---|---|---|---|
| 15 | 22 | 7 | 3 |
| 38 | 6 | 6 | 1 |
| 42 | 13 | 6 | 2 |

Even though the above statement gives the same answer as the prior GROUP BY example, it is not the same internally. Nor is it (probably) as efficient, and it is certainly not as easy to understand. Therefore, when in doubt, use the GROUP BY syntax.

### 6.1.5. Window Definition

An OLAP function works on a "window" of matching rows. This window can be defined as:

- All matching rows.
- All matching rows within a partition.
- Some moving subset of the matching rows (within a partition, if defined).

A moving window has to have an ORDER BY clause so that the set of matching rows can be determined.

**Window Size Partitions**

- **UNBOUNDED PRECEDING**: All of the preceding rows.
- **Number PRECEDING**: The "n" preceding rows.
- **UNBOUNDED FOLLOWING**: All of the following rows.
- **Number FOLLOWING**: The "n" following rows.
- **CURRENT ROW**: Only the current row.

Defaults

- **No ORDER BY**: UNBOUNDED PRECEDING to UNBOUNDED FOLLOWING.
- **ORDER BY only**: UNBOUNDED PRECEDING to CURRENT ROW.
- **No BETWEEN**: CURRENT ROW to "n" preceding/following row or rank.
- **BETWEEN stmt**: From "n" to "n" preceding/following row or rank. The end-point must be greater than or equal to the starting point.

### 6.1.6. Sample Queries

Below is a query that illustrates some of the above concepts:

*Different window sizes_*

```
SELECT id
     , salary
     , DEC(AVG(salary) OVER()                 ,7,2) AS avg_all
     , DEC(AVG(salary) OVER(ORDER BY id)      ,7,2) AS avg_odr
     , DEC(AVG(salary)
        OVER(ORDER BY id
             ROWS BETWEEN UNBOUNDED PRECEDING
             AND UNBOUNDED FOLLOWING)         ,7,2) AS avg_p_f
     , DEC(AVG(salary)
        OVER(ORDER BY id
             ROWS BETWEEN UNBOUNDED PRECEDING
             AND CURRENT ROW)                ,7,2) AS avg_p_c
     , DEC(AVG(salary)
        OVER(ORDER BY id
             ROWS BETWEEN CURRENT ROW
             AND UNBOUNDED FOLLOWING)        ,7,2) AS avg_c_f
     , DEC(AVG(salary)
        OVER(ORDER BY id
             ROWS BETWEEN 2 PRECEDING
             AND 1 FOLLOWING)                ,7,2) AS avg_2_1
FROM staff
WHERE dept IN (15,20)
AND id > 20
ORDER BY id;
```

*ANSWER*

| ID | SALARY | AVG_ALL | AVG_ODR | AVG_P_F | AVG_P_C | AVG_C_F | AVG_2_1 |
|---|---|---|---|---|---|---|---|
| 50 | 80659.80 | 53281.11 | 80659.80 | 53281.11 | 80659.80 | 53281.11 | 78581.31 |
| 70 | 76502.83 | 53281.11 | 78581.31 | 53281.11 | 78581.31 | 47805.37 | 66889.07 |
| 80 | 43504.60 | 53281.11 | 66889.07 | 53281.11 | 66889.07 | 40631.01 | 60793.85 |
| 110 | 42508.20 | 53281.11 | 60793.85 | 53281.11 | 60793.85 | 39673.15 | 51193.53 |
| 170 | 42258.50 | 53281.11 | 57086.78 | 53281.11 | 57086.78 | 38255.62 | 40631.01 |
| 190 | 34252.75 | 53281.11 | 53281.11 | 53281.11 | 53281.11 | 34252.75 | 39673.15 |

When the BETWEEN syntax is used, the start of the range/rows must be less than or equal to the end of the range/rows.

When no BETWEEN is used, the set of rows to be evaluated goes from the current row up or down to the end value:

*Different window sizes*

```
SELECT id
, SUM(id) OVER(ORDER BY id)                          AS sum1
, SUM(id) OVER(ORDER BY id ROWS 1 PRECEDING)         AS sum2
, SUM(id) OVER(ORDER BY id ROWS UNBOUNDED PRECEDING) AS sum3
, SUM(id) OVER(ORDER BY id ROWS CURRENT ROW)         AS sum4
, SUM(id) OVER(ORDER BY id ROWS2 FOLLOWING)          AS sum6
, SUM(id) OVER(ORDER BY id ROWS UNBOUNDED FOLLOWING) AS sum6
FROM staff
WHERE id < 40
ORDER BY id;
```

*ANSWER*

| ID | SUM1 | SUM2 | SUM3 | SUM4 | SUM5 | SUM6 |
|---|---|---|---|---|---|---|
| 10 | 10 | 10 | 10 | 10 | 60 | 60 |
| 20 | 30 | 30 | 30 | 20 | 50 | 50 |
| 30 | 60 | 50 | 60 | 30 | 30 | 30 |

### 6.1.7. ROWS vs. RANGE

(OLAP, ROWS  A moving window of rows to be evaluated (relative to the current row) can be defined using either the ROW or RANGE expressions. These differ as follows:

- **ROWS**: Refers to the "n" rows before and/or after (within the partition), as defined by the ORDER BY.
- **RANGE**: Refers to those rows before and/or after (within the partition) that are within an

arithmetic range of the current row, as defined by the ORDER BY.

The next query compares the ROW and RANGE expressions:

*ROW vs. RANGE example*

```
SELECT id
, SMALLINT(SUM(id)
    OVER(ORDER BY id
         RANGE BETWEEN 10 PRECEDING AND 10 FOLLOWING))        AS rng1
, SMALLINT(SUM(id)
    OVER(ORDER BY id
         ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING))           AS row1
, SMALLINT(SUM(id)
    OVER(ORDER BY id
         RANGE BETWEEN 10 PRECEDING AND CURRENT ROW))         AS rng2
, SMALLINT(SUM(id)
    OVER(ORDER BY id
         ROWS BETWEEN 3 PRECEDING AND 1 PRECEDING))           AS row2
, SMALLINT(SUM(id)
    OVER(ORDER BY id DESC
         ROWS BETWEEN 3 PRECEDING AND 1 PRECEDING))           AS row3
, SMALLINT(SUM(id)
    OVER(ORDER BY id
         RANGE BETWEEN UNBOUNDED PRECEDING AND 20 FOLLOWING)) AS rng3
FROM staff
WHERE id < 60
ORDER BY id;
```

*ANSWER*

| ID | RNG1 | ROW1 | RNG2 | ROW2 | ROW3 | RNG3 |
|---|---|---|---|---|---|---|
| 10 | 30 | 30 | 10 | - | 90 | 60 |
| 20 | 60 | 60 | 30 | 10 | 120 | 100 |
| 30 | 90 | 90 | 50 | 30 | 90 | 150 |
| 40 | 120 | 120 | 70 | 60 | 50 | 150 |
| 50 | 90 | 90 | 90 | 90 | - | 150 |

**Usage Notes**

- An ORDER BY statement is required when using either expression.
- If no RANGE or ROWS expression was provided, the default range (assuming there was an ORDER BY) is all preceding rows – up to the current row.
- When using the RANGE expression, only one expression can be specified in the ORDER BY, and that expression must be numeric.

### 6.1.8. ORDER BY Expression

The ORDER BY phrase has several purposes:

- It defines the set of rows that make up a moving window.
- It provides a set of values to do aggregations on. Each distinct value gets a new result.
- It gives a direction to the aggregation function processing (i.e. ASC or DESC).

An ORDER BY expression is required for the RANK and DENSE_RANK functions. It is optional for all others (except of using ROWS or RANGE).

**Usage Notes**

- **ASC**: Sorts the values in ascending order. This is the default.
- **DESC**: Sorts the values in descending order.
- **NULLS**: Determines whether null values are sorted high or low, relative to the non-null values present. Note that the default option differs for ascending and descending order.
- **Sort Expression**: The sort-key expression can be any valid column, or any scalar expression is deterministic, and has no external action.
- **ORDER BY ORDER OF table-designator**: The table designator refers to a subselect or fullselect in the query and any ordering defined on columns in that subselect or fullselect (note: if there is no explicit ordering the results are unpredictable). If the subselect or fullselect ORDER BY is changed, the ordering sequence will automatically change to match. Note that the final query may have an ordering that differs from that in the subselect or fullselect.

When the table designator refers to a table in the current subselect or fullselect, as opposed to the results of a nested subselect or fullselect, the values are unpredictable.

### 6.1.9. Sample Query

In the next query, various aggregations are done on a variety of fields, and on a nested-tableexpression that contains an ORDER BY. Observe that the ascending fields sum or count up, while the descending fields sum down. Also observe that each aggregation field gets a separate result for each new set of rows, as defined in the ORDER BY phrase:

*ORDER BY example*

```
SELECT dept
     , name
     , salary
     , DEC(SUM(salary) OVER(ORDER BY dept) ,8,2)           AS sum1
     , DEC(SUM(salary) OVER(ORDER BY dept DESC) ,8,2)      AS sum2
     , DEC(SUM(salary) OVER(ORDER BY ORDER OF s1) ,8,2)    AS sum3
     , SMALLINT(RANK() OVER(ORDER BY salary, name, dept) ) AS r1
     , SMALLINT(RANK() OVER(ORDER BY ORDER OF s1)          AS r2
     , ROW_NUMBER() OVER(ORDER)                            AS w1
     , COUNT(*) OVER(ORDER BY salary)                      AS w2
FROM (SELECT *
      FROM staff
      WHERE id < 60
      ORDER BY dept
             , name
      ) AS s1
ORDER BY 1, 2;
```

*ANSWER*

| DEPT | NAME | SALARY | SUM1 | SUM2 | SUM3 | R1 | R2 | W1 | W2 |
|---|---|---|---|---|---|---|---|---|---|
| 15 | Hanes | 80659.80 | 80659.80 | 412701.30 | 80659.80 | 4 | 1 | 4 | 4 |
| 20 | Pernal | 78171.25 | 257188.55 | 332041.50 | 158831.05 | 3 | 2 | 3 | 3 |
| 20 | Sanders | 98357.50 | 257188.55 | 332041.50 | 257188.55 | 5 | 3 | 5 | 5 |
| 38 | Marenghi | 77506.75 | 412701.30 | 155512.75 | 334695.30 | 1 | 4 | 1 | 1 |
| 38 | O'Brien | 78006.00 | 412701.30 | 155512.75 | 412701.30 | 2 | 5 | 2 | 2 |

There is no relationship between the ORDER BY used in an OLAP function, and the final ordering of the answer. Both are calculated independently.

### 6.1.10. Table Designator

The next two queries illustrate referencing a table designator in a subselect. Observe that as the ORDER BY in the subselect changes, the ordering sequence changes. Note that the final query output order does match that of the subselect:

```
SELECT id
     , name
     , ROW_NUMBER()
       OVER(ORDER BY ORDER OF s) od
FROM (SELECT *
      FROM staff
      WHERE id < 50
      ORDER BY name ASC
     ) AS s
ORDER BY id ASC;
```

*ANSWER*

| ID | NAME | OD |
|---|---|---|
| 10 | Sanders | 4 |
| 20 | Pernal | 3 |
| 30 | Marenghi | 1 |
| 40 | O'Brien | 2 |

*ORDER BY table designator examples*

```
SELECT id
     , name
     , ROW_NUMBER()
       OVER(ORDER BY ORDER OF s) od
FROM (SELECT *
      FROM staff
      WHERE id < 50
      ORDER BY name DESC
     ) AS s
ORDER BY id ASC;
```

*ANSWER*

| ID | NAME | OD |
|---|---|---|
| 10 | Sanders | 1 |
| 20 | Pernal | 2 |
| 30 | Marenghi | 4 |
| 40 | O'Brien | 3 |

### 6.1.11. Nulls Processing

When writing the ORDER BY, one can optionally specify whether or not null values should be counted as high or low. The default, for an ascending field is that they are counted as high (i.e. come

last), and for a descending field, that they are counted as low:

*Overriding the default null ordering sequence*

```
SELECT id
     , years                                          AS yr
     , salary
     , DENSE_RANK() OVER(ORDER BY years ASC)          AS a
     , DENSE_RANK() OVER(ORDER BY years ASC NULLS FIRST)  AS af
     , DENSE_RANK() OVER(ORDER BY years ASC NULLS LAST )  AS al
     , DENSE_RANK() OVER(ORDER BY years DESC)         AS d
     , DENSE_RANK() OVER(ORDER BY years DESC NULLS FIRST) AS df
     , DENSE_RANK() OVER(ORDER BY years DESC NULLS LAST ) AS dl
FROM staff
WHERE id < 100
ORDER BY years
       , salary;
```

*ANSWER*

| ID | YR | SALARY | A | AF | AL | D | DF | DL |
|---|---|---|---|---|---|---|---|---|
| 30 | 5 | 77506.75 | 1 | 2 | 1 | 6 | 6 | 5 |
| 90 | 6 | 38001.75 | 2 | 3 | 2 | 5 | 5 | 4 |
| 40 | 6 | 78006.00 | 2 | 3 | 2 | 5 | 5 | 4 |
| 70 | 7 | 76502.83 | 3 | 4 | 3 | 4 | 4 | 3 |
| 10 | 7 | 98357.50 | 3 | 4 | 3 | 4 | 4 | 3 |
| 20 | 8 | 78171.25 | 4 | 5 | 4 | 3 | 3 | 2 |
| 50 | 10 | 80659.80 | 5 | 6 | 5 | 2 | 2 | 1 |
| 80 | - | 43504.60 | 6 | 1 | 6 | 1 | 1 | 6 |
| 60 | - | 66808.30 | 6 | 1 | 6 | 1 | 1 | 6 |

In general, one null value does not equal another null value. But, as is illustrated above, for purposes of assigning rank, all null values are considered equal.

### 6.1.12. Counting Nulls

The DENSE RANK and RANK functions include null values when calculating rankings. By contrast the COUNT DISTINCT statement excludes null values when counting values. Thus, as is illustrated below, the two methods will differ (by one) when they are used get a count of distinct values - if there are nulls in the target data:

*Counting distinct values - comparison*

```
SELECT COUNT(DISTINCT years) AS y#1
     , MAX(y#)               AS y#2
FROM (SELECT years
           , DENSE_RANK() OVER(ORDER BY years) AS y#
      FROM staff
      WHERE id < 100
     ) AS xxx
ORDER BY 1;
```

*ANSWER*

| Y#1 | Y#2 |
|---|---|
| 5 | 6 |

## 6.1.13. OLAP Functions

### RANK and DENSE_RANK

The RANK and DENSE_RANK functions enable one to rank the rows returned by a query. The result is of type BIGINT.

The ORDER BY phrase, which is required, is used to both sequence the values, and to tell Db2 when to generate a new value.

#### RANK vs. DENSE_RANK

The two functions differ in how they handle multiple rows with the same value:

- The RANK function returns the number of proceeding rows, plus one. If multiple rows have equal values, they all get the same rank, while subsequent rows get a ranking that counts all of the prior rows. Thus, there may be gaps in the ranking sequence.
- The DENSE_RANK function returns the number of proceeding distinct values, plus one. If multiple rows have equal values, they all get the same rank. Each change in data value causes the ranking number to be incremented by one.

### Usage Notes

- The ORDER BY expression is mandatory.
- The PARTITION BY expression is optional.

#### Compare Functions

The following query illustrates the use of the two functions:

*Ranking functions example*

```
SELECT id
     , years
     , salary
     , RANK()
        OVER(ORDER BY years) AS rank#
     , DENSE_RANK()
        OVER(ORDER BY years) AS dense#
     , ROW_NUMBER()
        OVER(ORDER BY years) AS row#
FROM staff
WHERE id    < 100
AND   years < 10
ORDER BY years;
```

*ANSWER*

| ID | YEARS | SALARY | RANK# | DENSE# | ROW# |
|---|---|---|---|---|---|
| 30 | 5 | 77506.75 | 1 | 1 | 1 |
| 40 | 6 | 78006.00 | 2 | 2 | 2 |
| 90 | 6 | 38001.75 | 2 | 2 | 3 |
| 10 | 7 | 98357.50 | 4 | 3 | 4 |
| 70 | 7 | 76502.83 | 4 | 3 | 5 |
| 20 | 8 | 78171.25 | 6 | 4 | 6 |

### 6.1.14. ORDER BY Usage

The mandatory ORDER BY phrase gives a sequence to the ranking, and also tells Db2 when to start a new rank value. The following query illustrates both uses:

*ORDER BY usage*

```
SELECT job                                                  AS job
     , years                                                AS yr
     , id                                                   AS id
     , name                                                 AS name
     , RANK() OVER(ORDER BY job ASC )                       AS a1
     , RANK() OVER(ORDER BY job ASC, years ASC)             AS a2
     , RANK() OVER(ORDER BY job ASC, years ASC ,id ASC )    AS a3
     , RANK() OVER(ORDER BY job DESC)                       AS d1
     , RANK() OVER(ORDER BY job DESC, years DESC)           AS d2
     , RANK() OVER(ORDER BY job DESC, years DESC, id DESC) AS d3
     , RANK() OVER(ORDER BY job ASC, years DESC, id ASC )   AS m1
     , RANK() OVER(ORDER BY job DESC, years ASC, id DESC)   AS m2
FROM staff
WHERE id    <   150
AND   years IN (6,7)
AND   job   >  'L'
ORDER BY job
       , years
       , id;
```

*ANSWER*

| JOB | YR | ID | NAME | A1 | A2 | A3 | D1 | D2 | D3 | M1 | M2 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Mgr | 6 | 140 | Fraye | 1 | 1 | 1 | 4 | 6 | 6 | 3 | 4 |
| Mgr | 7 | 10 | Sanders | 1 | 2 | 2 | 4 | 4 | 5 | 1 | 6 |
| Mgr | 7 | 100 | Plotz | 1 | 2 | 3 | 4 | 4 | 4 | 2 | 5 |
| Sales | 6 | 40 | O’Brien | 4 | 4 | 4 | 1 | 2 | 3 | 5 | 2 |
| Sales | 6 | 90 | Koonitz | 4 | 4 | 5 | 1 | 2 | 2 | 6 | 1 |
| Sales | 7 | 70 | Rothman | 4 | 6 | 6 | 1 | 1 | 1 | 4 | 3 |

Observe above that adding more fields to the ORDER BY phrase resulted in more ranking values being generated.

### 6.1.15. PARTITION Usage

The optional PARTITION phrase lets one rank the data by subsets of the rows returned. In the following example, the rows are ranked by salary within year:

*Values ranked by subset of rows*

```
SELECT id
     , years AS yr
     , salary
     , RANK() OVER(PARTITION BY years
                   ORDER BY salary)  AS r1
FROM staff
WHERE id     < 80
AND   years IS NOT NULL
ORDER BY years
       , salary;
```

*ANSWER*

| ID | YR | SALARY | R1 |
|---|---|---|---|
| 30 | 5 | 77506.75 | 1 |
| 40 | 6 | 78006.00 | 1 |
| 70 | 7 | 76502.83 | 1 |
| 10 | 7 | 98357.50 | 2 |
| 20 | 8 | 78171.25 | 1 |
| 50 | 0 | 80659.80 | 1 |

### 6.1.16. Multiple Rankings

One can do multiple independent rankings in the same query:

*Multiple rankings in same query*

```
SELECT id
     , years
     , salary
     , SMALLINT(RANK() OVER(ORDER BY years ASC))  AS rank_a
     , SMALLINT(RANK() OVER(ORDER BY years DESC)) AS rank_d
     , SMALLINT(RANK() OVER(ORDER BY id, years))  AS rank_iy
FROM staff
WHERE id  <  100
AND years IS NOT NULL
ORDER BY years;
```

### 6.1.17. Dumb Rankings

If one wants to, one can do some really dumb rankings. All of the examples below are fairly stupid, but arguably the dumbest of the lot is the last. In this case, the "ORDER BY 1" phrase ranks the rows returned by the constant "one", so every row gets the same rank. By contrast the "ORDER BY 1" phrase at the bottom of the query sequences the rows, and so has valid business meaning:

*Dumb rankings, SQL*

```
SELECT id
     , years
     , name
     , salary
     , SMALLINT(RANK() OVER(ORDER BY SUBSTR(name,3,2))) AS dumb1
     , SMALLINT(RANK() OVER(ORDER BY salary / 1000))    AS dumb2
     , SMALLINT(RANK() OVER(ORDER BY years * ID))       AS dumb3
     , SMALLINT(RANK() OVER(ORDER BY 1))                AS dumb4
FROM staff
WHERE id < 40
AND years IS NOT NULL
ORDER BY 1;
```

| ID | YEARS | NAME | SALARY | DUMB1 | DUMB2 | DUMB3 | DUMB4 |
|---|---|---|---|---|---|---|---|
| 10 | 7 | Sanders | 98357.50 | 1 | 3 | 1 | 1 |
| 20 | 8 | Pernal | 78171.25 | 3 | 2 | 3 | 1 |
| 30 | 5 | Marenghi | 77506.75 | 2 | 1 | 2 | 1 |

### 6.1.18. Subsequent Processing

The ranking function gets the rank of the value as of when the function was applied. Subsequent processing may mean that the rank no longer makes sense. To illustrate this point, the following query ranks the same field twice. Between the two ranking calls, some rows were removed from the answer set, which has caused the ranking results to differ:

*Subsequent processing of ranked data*

```
SELECT xxx.*
     , RANK()OVER(ORDER BY id) AS r2
FROM (SELECT id
           , name
           , RANK() OVER(ORDER BY id) AS r1
      FROM staff
      WHERE id     < 100
      AND   years IS NOT NULL
    ) AS xxx
WHERE id > 30
ORDER BY id;
```

*ANSWER*

| ID | NAME | R1 | R2 |
|---|---|---|---|
| 40 | O'Brien | 4 | 1 |
| 50 | Hanes | 5 | 2 |

| ID | NAME | R1 | R2 |
|---|---|---|---|
| 70 | Rothman | 6 | 3 |
| 90 | Koonitz | 7 | 4 |

### 6.1.19. Ordering Rows by Rank

One can order the rows based on the output of a ranking function. This can let one sequence the data in ways that might be quite difficult to do using ordinary SQL. For example, in the following query the matching rows are ordered so that all those staff with the highest salary in their respective department come first, followed by those with the second highest salary, and so on. Within each ranking value, the person with the highest overall salary is listed first:

*Ordering rows by rank, using RANK function*

```
SELECT id
     , RANK() OVER(PARTITION BY dept
                   ORDER BY salary DESC) AS r1
     , salary
     , dept AS dp
FROM staff
WHERE id     < 80
AND   years IS NOT NULL
ORDER BY r1     ASC
       , salary DESC;
```

*ANSWER*

| ID | R1 | SALARY | DP |
|---|---|---|---|
| 10 | 1 | 98357.50 | 20 |
| 50 | 1 | 80659.80 | 15 |
| 40 | 1 | 78006.00 | 38 |
| 20 | 2 | 78171.25 | 20 |
| 30 | 2 | 77506.75 | 38 |
| 70 | 2 | 76502.83 | 15 |

Here is the same query, written without the ranking function:

*Ordering rows by rank, using sub-query*

```
SELECT id
    , (SELECT COUNT(*)
       FROM staff s2
       WHERE s2.id   < 80
       AND s2.years IS NOT NULL
       AND s2.dept = s1.dept
       AND s2.salary >= s1.salary
      ) AS R1
    , salary
    , dept AS dp
FROM staff s1
WHERE id     < 80
AND   years IS NOT NULL
ORDER BY r1     ASC
       , salary DESC;
```

*ANSWER*

| ID | R1 | SALARY | DP |
|---|---|---|---|
| 10 | 1 | 98357.50 | 20 |
| 50 | 1 | 80659.80 | 15 |
| 40 | 1 | 78006.00 | 38 |
| 20 | 2 | 78171.25 | 20 |
| 30 | 2 | 77506.75 | 38 |
| 70 | 2 | 76502.83 | 15 |

The above query has all of the failings that were discussed at the beginning of this chapter:

- The nested table expression has to repeat all of the predicates in the main query, and have predicates that define the ordering sequence. Thus it is hard to read.
- The nested table expression will (inefficiently) join every matching row to all prior rows.

### 6.1.20. Selecting the Highest Value

The ranking functions can also be used to retrieve the row with the highest value in a set of rows. To do this, one must first generate the ranking in a nested table expression, and then query the derived field later in the query. The following statement illustrates this concept by getting the person, or persons, in each department with the highest salary:

*Get highest salary in each department, use RANK function*

```
SELECT id
     , salary
     , dept AS dp
FROM
   (SELECT s1.*
         , RANK() OVER(PARTITION BY dept
                       ORDER BY salary DESC) AS r1
    FROM staff s1
    WHERE id     < 80
    AND   years IS NOT NULL
   ) AS xxx
WHERE r1 = 1
ORDER BY dp;
```

*ANSWER*

| ID | SALARY | DP |
|---|---|---|
| 50 | 80659.80 | 15 |
| 10 | 98357.50 | 20 |
| 40 | 78006.00 | 38 |

Here is the same query, written using a correlated sub-query:

*Get highest salary in each department, use correlated sub-query*

```
SELECT id
     , salary
     , dept AS dp
FROM staff s1
WHERE id     < 80
AND   years IS NOT NULL
AND NOT EXISTS
  (SELECT *
   FROM staff s2
   WHERE s2.id     < 80
   AND   s2.years IS NOT NULL
   AND s2.dept = s1.dept
   AND s2.salary > s1.salary)
ORDER BY dp;
```

| ID | SALARY | DP |
|---|---|---|
| 50 | 80659.80 | 15 |
| 10 | 98357.50 | 20 |
| 40 | 78006.00 | 38 |

Here is the same query, written using an uncorrelated sub-query:

*Get highest salary in each department, use uncorrelated sub-query*

```
SELECT id
     , salary
     , dept AS dp
FROM staff
WHERE id     < 80
AND   years IS NOT NULL
AND (dept, salary) IN
  (SELECT dept, MAX(salary)
   FROM staff
   WHERE id     < 80
   AND   years IS NOT NULL
   GROUP BY dept)
ORDER BY dp;
```

*ANSWER*

| ID | SALARY | DP |
|---|---|---|
| 50 | 80659.80 | 15 |
| 10 | 98357.50 | 20 |
| 40 | 78006.00 | 38 |

Arguably, the first query above (i.e. the one using the RANK function) is the most elegant of the series because it is the only statement where the basic predicates that define what rows match are written once. With the two sub-query examples, these predicates have to be repeated, which can often lead to errors.

## 6.1.21. ROW_NUMBER

The ROW_NUMBER function lets one number the rows being returned. The result is of type BIGINT. A syntax diagram follows. Observe that unlike with the ranking functions, the ORDER BY is not required.

**ORDER BY Usage**

You don't have to provide an ORDER BY when using the ROW_NUMBER function, but not doing so can be considered to be either brave or foolish, depending on one's outlook on life. To illustrate this issue, consider the following query:

*ORDER BY example, 1 of 3*

```
SELECT id
     , name
     , ROW_NUMBER() OVER()            AS r1
     , ROW_NUMBER() OVER(ORDER BY id) AS r2
FROM staff
WHERE id     < 50
AND   years IS NOT NULL
ORDER BY id;
```

*ANSWER*

| ID | NAME | R1 | R2 |
|---|---|---|---|
| 10 | Sanders | 1 | 1 |
| 20 | Pernal | 2 | 2 |
| 30 | Marenghi | 3 | 3 |
| 40 | O'Brien | 4 | 4 |

In the above example, both ROW_NUMBER functions return the same set of values, which happen to correspond to the sequence in which the rows are returned. In the next query, the second ROW_NUMBER function purposely uses another sequence:

*ORDER BY example, 2 of 3*

```
SELECT id
     , name
     , ROW_NUMBER() OVER()              AS r1
     , ROW_NUMBER() OVER(ORDER BY name) AS r2
FROM staff
WHERE id     < 50
AND   years IS NOT NULL
ORDER BY id;
```

*ANSWER*

| ID | NAME | R1 | R2 |
|---|---|---|---|
| 10 | Sanders | 4 | 4 |
| 20 | Pernal | 3 | 3 |
| 30 | Marenghi | 1 | 1 |
| 40 | O'Brien | 2 | 2 |

Observe that changing the second function has had an impact on the first. Now lets see what happens when we add another ROW_NUMBER function:

*ORDER BY example, 3 of 3*

```
SELECT id
,name
,ROW_NUMBER() OVER()              AS r1
,ROW_NUMBER() OVER(ORDER BY ID)   AS r2
,ROW_NUMBER() OVER(ORDER BY NAME) AS r3
FROM staff
WHERE id     < 50
AND   years IS NOT NULL
ORDER BY id;
```

*ANSWER*

| ID | NAME | R1 | R2 | R3 |
|---|---|---|---|---|
| 10 | Sanders | 1 | 1 | 4 |
| 20 | Pernal | 2 | 2 | 3 |
| 30 | Marenghi | 3 | 3 | 1 |
| 40 | O'Brien | 4 | 4 | 2 |

Observe that now the first function has reverted back to the original sequence.

When not given an explicit ORDER BY, the ROW_NUMBER function, may create a value in any odd order. Usually, the sequence will reflect the order in which the rows are returned - but not always.

### 6.1.22. PARTITION Usage

The PARTITION phrase lets one number the matching rows by subsets of the rows returned. In the following example, the rows are both ranked and numbered within each JOB:

*Use of PARTITION phrase*

```
SELECT job
     , years
     , id
     , name
     , ROW_NUMBER() OVER(PARTITION BY job ORDER BY years) AS row#
     , RANK()       OVER(PARTITION BY job ORDER BY years) AS rn1#
     , DENSE_RANK() OVER(PARTITION BY job ORDER BY years) AS rn2#
FROM staff
WHERE id     < 150
AND   years IN (6,7)
AND job > 'L'
ORDER BY job,years;
```

*ANSWER*

| JOB | YEARS | ID | NAME | ROW# | RN1# | RN2# |
|---|---|---|---|---|---|---|
| Mgr | 6 | 140 | Fraye | 1 | 1 | 1 |
| Mgr | 7 | 10 | Sanders | 2 | 2 | 2 |
| Mgr | 7 | 100 | Plotz | 3 | 2 | 2 |
| Sales | 6 | 40 | O'Brien | 1 | 1 | 1 |
| Sales | 6 | 90 | Koonitz | 2 | 1 | 1 |
| Sales | 7 | 70 | Rothman | 3 | 3 | 2 |

One problem with the above query is that the final ORDER BY that sequences the rows does not identify a unique field (e.g. ID). Consequently, the rows can be returned in any sequence within a given JOB and YEAR. Because the ORDER BY in the ROW_NUMBER function also fails to identify a unique row, this means that there is no guarantee that a particular row will always give the same row number. For consistent results, ensure that both the ORDER BY phrase in the function call, and at the end of the query, identify a unique row. And to always get the rows returned in the desired row-number sequence, these phrases must be equal.

#### Selecting "n" Rows

To query the output of the ROW_NUMBER function, one has to make a nested temporary table that contains the function expression. In the following example, this technique is used to limit the query to the first three matching rows:

*Select first 3 rows, using ROW_NUMBER function*

```
SELECT *
FROM
    (SELECT id
          , name
          , ROW_NUMBER() OVER(ORDER BY id) AS r
    FROM staff
    WHERE id     < 100
    AND   years IS NOT NULL
   ) AS xxx
WHERE r <= 3
ORDER BY id;
```

*ANSWER*

| ID | NAME | R |
|---|---|---|
| 10 | Sanders | 1 |
| 20 | Pernal | 2 |
| 30 | Marenghi | 3 |

In the next query, the FETCH FIRST "n" ROWS notation is used to achieve the same result:

*Select first 3 rows, using FETCH FIRST notation*

```
SELECT id
     , name
     , ROW_NUMBER() OVER(ORDER BY id) AS r
FROM staff
WHERE id     < 100
AND   years IS NOT NULL
ORDER BY id
FETCH FIRST 3 ROWS ONLY;
```

*ANSWER*

| ID | NAME | R |
|---|---|---|
| 10 | Sanders | 1 |
| 20 | Pernal | 2 |
| 30 | Marenghi | 3 |

So far, the ROW_NUMBER and the FETCH FIRST notations seem to be about the same. But the former is much more flexible. To illustrate, the next query gets the 3rd through 6th rows:

*Select 3rd through 6th rows*

```
SELECT *
FROM
    (SELECT id
          , name
          , ROW_NUMBER() OVER(ORDER BY id) AS r
     FROM staff
     WHERE id     < 200
     AND   years IS NOT NULL
    ) AS xxx
WHERE r BETWEEN 3 AND 6
ORDER BY id;
```

*ANSWER*

| ID | NAME | R |
|---|---|---|
| 30 | Marenghi | 3 |
| 40 | O'Brien | 4 |
| 50 | Hanes | 5 |
| 70 | Rothman | 6 |

In the next query we get every 5th matching row - starting with the first:

*Select every 5th matching row*

```
SELECT *
FROM
    (SELECT id
          , name
          , ROW_NUMBER() OVER(ORDER BY id) AS r
     FROM staff
     WHERE id     < 200
     AND   years IS NOT NULL
    ) AS xxx
WHERE (r - 1) = ((r - 1) / 5) * 5
ORDER BY id;
```

*ANSWER*

| ID | NAME | R |
|---|---|---|
| 10 | Sanders | 1 |
| 70 | Rothman | 6 |
| 140 | Fraye | 11 |
| 190 | Sneider | 16 |

In the next query we get the last two matching rows:

*Select last two rows*

```
SELECT *
FROM
    (SELECT id
          , name
          , ROW_NUMBER() OVER(ORDER BY id DESC) AS r
     FROM staff
     WHERE id     < 200
     AND   years IS NOT NULL
    ) AS xxx
WHERE r <= 2
ORDER BY id;
```

*ANSWER*

| ID | NAME | R |
|---|---|---|
| 180 | Abrahams | 2 |
| 190 | Sneider | 1 |

### 6.1.23. Selecting "n" or more Rows

Imagine that one wants to fetch the first "n" rows in a query. This is easy to do, and has been illustrated above. But imagine that one also wants to keep on fetching if the following rows have the same value as the "nth". In the next example, we will get the first three matching rows in the STAFF table, ordered by years of service. However, if the 4th row, or any of the following rows, has the same YEAR as the 3rd row, then we also want to fetch them.

The query logic goes as follows:

- Select every matching row in the STAFF table, and give them all both a row-number and a ranking value. Both values are assigned according to the order of the final output. Do all of this work in a nested table expression.
- Select from the nested table expression where the rank is three or less.

The query relies on the fact that the RANK function (see RANK and DENSE_RANK) assigns the lowest common row number to each row with the same ranking:

*Select first "n" rows, or more if needed*

```
SELECT *
FROM
    (SELECT years
    , id
    , name
    , RANK() OVER(ORDER BY years)           AS rnk
    , ROW_NUMBER() OVER(ORDER BY years, id) AS row
    FROM staff
    WHERE id     < 200
    AND   years IS NOT NULL
    ) AS xxx
WHERE rnk <= 3
ORDER BY years
       , id;
```

*ANSWER*

| YEARS | ID | NAME | RNK | ROW |
|---|---|---|---|---|
| 3 | 180 | Abrahams | 1 | 1 |
| 4 | 170 | Kermisch | 2 | 2 |
| 5 | 30 | Marenghi | 3 | 3 |
| 5 | 110 | Ngan | 3 | 4 |

The type of query illustrated above can be extremely useful in certain business situations. To illustrate, imagine that one wants to give a reward to the three employees that have worked for the company the longest. Stopping the query that lists the lucky winners after three rows are fetched can get one into a lot of trouble if it happens that there are more than three employees that have

worked for the company for the same number of years.

### 6.1.24. Selecting "n" Rows - Efficiently

Sometimes, one only wants to fetch the first "n" rows, where "n" is small, but the number of matching rows is extremely large. In this section, we will discus how to obtain these "n" rows efficiently, which means that we will try to fetch just them without having to process any of the many other matching rows. Below is an invoice table. Observe that we have defined the INV# field as the primary key, which means that Db2 will build a unique index on this column:

*Performance test table - definition*

```
CREATE TABLE invoice
( inv#       INTEGER      NOT NULL
, customer#  INTEGER      NOT NULL
, sale_date  DATE         NOT NULL
, sale_value DECIMAL(9,2) NOT NULL
, CONSTRAINT ctx1 PRIMARY KEY (inv#)
, CONSTRAINT ctx2 CHECK(inv# >= 0));
```

The next SQL statement will insert 1,000,000 rows into the above table. After the rows are inserted a REORG and RUNSTATS is run, so the optimizer can choose the best access path.

*Performance test table - insert 1,000,000 rows*

```
INSERT INTO invoice
WITH temp (n, m) AS
(VALUES
  (INTEGER(0), RAND(1))
  UNION ALL
  SELECT n+1, RAND()
  FROM temp
  WHERE n+1 < 1000000
)
SELECT n                                AS inv#
     , INT(m * 1000)                    AS customer#
     , DATE('2000-11-01') + (m*40) DAYS AS sale_date
     , DECIMAL((m * m * 100),8,2)       AS sale_value
FROM temp;
```

Imagine we want to retrieve the first five rows (only) from the above table. Below are several queries that get this result. For each query, the elapsed time, as measured by Db2BATCH, is provided. Below we use the "FETCH FIRST n ROWS" notation to stop the query at the 5th row. The query scans the primary index to get first five matching rows, and thus is cheap:

*Fetch first 5 rows - 0.000 elapsed seconds*

```
SELECT s.*
FROM invoice s
ORDER BY inv#
FETCH FIRST 5 ROWS ONLY;
```

The next query is essentially the same as the prior, but this time we tell Db2 to optimize the query for fetching five rows. Nothing has changed, and all is good:

*Fetch first 5 rows - 0.000 elapsed seconds*

```
SELECT s.*
FROM invoice s
ORDER BY inv#
FETCH FIRST 5 ROWS ONLY
OPTIMIZE FOR 5 ROWS;
```

The next query is the same as the first, except that it invokes the ROW_NUMBER function to passively sequence the output. This query also uses the primary index to identify the first five matching rows, and so is cheap:

*Fetch first 5 rows+ number rows - 0.000 elapsed seconds*

```
SELECT s.*
     , ROW_NUMBER() OVER() AS row#
FROM invoice s
ORDER BY inv#
FETCH FIRST 5 ROWS ONLY;
```

The next query is the same as the previous. It uses a nested-table-expression, but no action is taken subsequently, so this code is ignored:

*Fetch first 5 rows+ number rows - 0.000 elapsed seconds*

```
SELECT *
FROM
    (SELECT s.*
          , ROW_NUMBER() OVER() AS row#
     FROM invoice s
    ) xxx
ORDER BY inv#
FETCH FIRST 5 ROWS ONLY;
```

All of the above queries processed only five matching rows. The next query will process all one million matching rows in order to calculate the ROW_NUMBER value, which is on no particular column. It will cost:

*Process and number all rows - 0.049 elapsed seconds*

```
SELECT *
FROM
    (SELECT s.*
          , ROW_NUMBER() OVER() AS row#
     FROM invoice s
    ) xxx
WHERE row# <= 5
ORDER BY inv#;
```

In the above query the "OVER()" phrase told Db2 to assign row numbers to each row. In the next query we explicitly provide the ROW_NUMBER with a target column, which happens to be the same at the ORDER BY sequence, and is also an indexed column. Db2 can use all this information to confine the query to the first "n" matching rows:

*Process and number 5 rows only - 0.000 elapsed seconds*

```
SELECT *
FROM
    (SELECT s.*
          , ROW_NUMBER() OVER(ORDER BY inv#) AS row#
     FROM invoice s
    ) xxx
WHERE row# <= 5
ORDER BY inv#;
```

Changing the above predicate to: "WHERE row# BETWEEN 1 AND 5" will get the same answer, but use a much less efficient access path.

One can also use recursion to get the first "n" rows. One begins by getting the first matching row, and then uses that row to get the next, and then the next, and so on (in a recursive join), until the required number of rows have been obtained. In the following example, we start by getting the row with the MIN invoice-number. This row is then joined to the row with the next to lowest invoice-number, which is then joined to the next, and so on. After five such joins, the cycle is stopped and the result is selected:

*Fetch first 5 rows - 0.000 elapsed seconds*

```
WITH temp (inv#, c#, sd, sv, n) AS
(SELECT inv.*
      , 1
FROM invoice inv
WHERE inv# =
    (SELECT MIN(inv#)
    FROM invoice)
UNION ALL
SELECT new.*
     , n + 1
FROM temp old
, invoice new
WHERE old.inv# < new.inv#
AND   old.n    < 5
AND   new.inv# =
    (SELECT MIN(xxx.inv#)
     FROM invoice xxx
     WHERE xxx.inv# > old.inv#)
)
SELECT *
FROM temp;
```

The above technique is nice to know, but it has several major disadvantages:

- It is not exactly easy to understand.
- It requires that all primary predicates (e.g. get only those rows where the sale-value is greater than $10,000) be repeated four times. In the above example there are none, which is unusual in the real world.
- It quickly becomes both very complicated and quite inefficient when the sequencing value is made up of multiple fields. In the above example, we sequenced by the INV# column, but imagine if we had used the sale-date, sale-value, and customer-number.
- It is extremely vulnerable to inefficient access paths. For example, if instead of joining from one (indexed) invoice-number to the next, we joined from one (non-indexed) customer-number to the next, the query would run forever.

In this section we have illustrated how minor changes to the SQL syntax can cause major changes in query performance. But to illustrate this phenomenon, we used a set of queries with 1,000,000 matching rows. In situations where there are far fewer matching rows, one can reasonably assume that this problem is not an issue.

### 6.1.25. FIRST_VALUE and LAST_VALUE

The FIRST_VALUE and LAST_VALUE functions get first or last value in the (moving) window of matching rows.

**Usage Notes**

- An expression value must be provided in the first set of parenthesis. Usually this will be a column name, but any valid scalar expression is acceptable.
- The PARTITION BY expression is optional.
- The ORDER BY expression is optional.
- See Window Definition for notes on how to define a moving-window of rows to process.
- If no explicit moving-window definition is provided, the default window size is between UNBOUNDED PRECEDING (of the partition and/or range) and the CURRENT ROW. This can sometimes cause logic errors when using the LAST_VALUE function. The last value is often simply the current row. To get the last matching value within the partition and/or range, set the upper bound to UNBOUNDED FOLLOWING.
- If IGNORE NULLS is specified, null values are ignored, unless all values are null, in which case the result is null. The default is RESPECT NULLS.

**Examples**

The following query illustrates the basics. The first matching name (in ID order) within each department is obtained:

*FIRST_VALUE function example*

```
SELECT dept
     , id
     , name
     , FIRST_VALUE(name) OVER(PARTITION BY dept
                              ORDER BY id)       AS frst
FROM staff
WHERE dept <= 15
AND   id   >  160
ORDER BY dept ,id;
```

*ANSWER*

| DEPT | ID | NAME | FRST |
|---|---|---|---|
| 10 | 210 | Lu | Lu |
| 10 | 240 | Daniels | Lu |
| 10 | 260 | Jones | Lu |
| 15 | 170 | Kermisch | Kermisch |

The next uses various ordering schemas and moving-window sizes the get a particular first or last value (within a department):

*Function examples*

```
SELECT dept
     , id
     , comm
     , FIRST_VALUE(comm) OVER(PARTITION BY dept
                              ORDER BY comm)                              AS first1
     , FIRST_VALUE(comm) OVER(PARTITION BY dept
                              ORDER BY comm NULLS FIRST)                  AS first2
     , FIRST_VALUE(comm) OVER(PARTITION BY dept
                              ORDER BY comm NULLS LAST)                   AS first3
     , FIRST_VALUE(comm) OVER(PARTITION BY dept
                              ORDER BY comm NULLS LAST
                              ROWS BETWEEN 1 PRECEDING AND CURRENT ROW) AS first4
     , LAST_VALUE(comm)  OVER(PARTITION BY dept
                              ORDER BY comm)                              AS last1
     , LAST_VALUE(comm)  OVER(PARTITION BY dept
                              ORDER BY comm NULLS FIRST
                              ROWS UNBOUNDED FOLLOWING)                   AS last2
FROM
staff
WHERE id < 100
AND dept < 30
ORDER BY dept ,comm;
```

*ANSWER*

| DEPT | ID | COMM | FIRST1 | FIRST2 | FIRST3 | FIRST4 | LAST1 | LAST2 |
|---|---|---|---|---|---|---|---|---|
| 15 | 70 | 1152.00 | 1152.00 | - | 1152.00 | 1152.00 | 1152.00 | 1152.00 |
| 15 | 50 | - | 1152.00 | - | 1152.00 | 1152.00 | - | 1152.00 |
| 20 | 80 | 128.20 | 128.20 | - | 128.20 | 128.20 | 128.20 | 612.45 |
| 20 | 20 | 612.45 | 128.20 | - | 128.20 | 128.20 | 612.45 | 612.45 |
| 20 | 10 | - | 128.20 | - | 128.20 | 612.45 | - | 612.45 |

The next query illustrates what happens when one, or all, of the matching values are null:

*Null value processing*

```
SELECT dept
     , id
     , comm
     , FIRST_VALUE(comm) OVER(PARTITION BY dept
                              ORDER BY comm)                          AS rn_lst
     , FIRST_VALUE(comm) OVER(PARTITION BY dept
                              ORDER BY comm NULLS LAST)               AS rn_ls2
     , FIRST_VALUE(comm) OVER(PARTITION BY dept
                              ORDER BY comm NULLS FIRST)              AS rn_fst
     , FIRST_VALUE(comm,'IGNORE NULLS') OVER(PARTITION BY dept
                                             ORDER BY comm NULLS FIRST) AS in_fst
FROM staff
WHERE id   BETWEEN 20 AND 160
AND   dept <= 20
ORDER BY dept ,comm;
```

*ANSWER*

| DEPT | ID | COMM | RN_LST | RN_LS2 | RN_FST | IN_FST |
|---|---|---|---|---|---|---|
| 10 | 160 | | | | | |
| 15 | 110 | 206.60 | 206.60 | 206.60 | - | 206.60 |
| 15 | 70 | 1152.00 | 206.60 | 206.60 | - | 206.60 |
| 15 | 50 | - | 206.60 | 206.60 | | |
| 20 | 80 | 128.20 | 128.20 | 128.20 | 128.20 | 128.20 |
| 20 | 20 | 612.45 | 128.20 | 128.20 | 128.20 | 128.20 |

## 6.1.26. LAG and LEAD

The LAG, and LEAD functions get the previous or next value from the (moving) window of matching rows:

- **LAG**: Get previous value. Return null if at first value.
- **LEAD**: Get next value. Return null if at last value.

**Usage Notes**

- An expression value must be provided in the first set of parenthesis. Usually this will be a column name, but any valid scalar expression is acceptable.
- The PARTITION BY expression is optional.
- The ORDER BY expression is mandatory.
- The default OFFSET value is 1. A value of zero refers to the current row. An offset that is outside of the moving-window returns null.
- If IGNORE NULLS is specified, a default (override) value must also be provided.

#### Examples

The next query uses the LAG function to illustrate what happens when one messes around with the ORDER BY expression:

*LAG and LEAD function Examples*

```
SELECT dept
     , id
     , comm
     , LAG(comm)                     OVER(PARTITION BY dept ORDER BY comm) AS lag1
     , LAG(comm,0)                   OVER(PARTITION BY dept ORDER BY comm) AS lag2
     , LAG(comm,2)                   OVER(PARTITION BY dept ORDER BY comm) AS lag3
     , LAG(comm,1,-1,'IGNORE NULLS') OVER(PARTITION BY dept ORDER BY comm) AS lag4
     , LEAD(comm)                    OVER(PARTITION BY dept ORDER BY comm) AS led1
FROM staff
WHERE id   BETWEEN 20 AND 160
AND   dept <= 20
ORDER BY dept ,comm;
```

*ANSWER*

| DEPT | ID | COMM | LAG1 | LAG2 | LAG3 | LAG4 | LED1 |
|---|---|---|---|---|---|---|---|
| 10 | 160 | - | - | - | - | -1.00 | - |
| 15 | 110 | 206.60 | - | 206.60 | - | -1.00 | 1152.00 |
| 15 | 70 | 1152.00 | 206.60 | 1152.00 | - | 206.60 | - |
| 15 | 50 | - | 1152.00 | - | 206.60 | 1152.00 | - |
| 20 | 80 | 128.20 | - | 128.20 | - | -1.00 | 612.45 |
| 20 | 20 | 612.45 | 128.20 | 612.45 | - | 128.20 | - |

### 6.1.27. Aggregation

The various aggregation functions let one do cute things like get cumulative totals or running averages. In some ways, they can be considered to be extensions of the existing Db2 column functions. The output type is dependent upon the input type.

#### Syntax Notes

Guess what - this is a complicated function. Be aware of the following:

- Any Db2 column function (e.g. AVG, SUM, COUNT), except ARRAY_AGG, can use the aggregation function.
- The OVER() usage aggregates all of the matching rows. This is equivalent to getting the current row, and also applying a column function (e.g. MAX, SUM) against all of the matching rows.
- The PARTITION BY expression is optional.
- The ORDER BY expression is mandatory if the aggregation is confined to a set of rows or range

of values. Otherwise it is optional. If a RANGE is specified (see ROWS vs. RANGE for definition), then the ORDER BY expression must be a single value that allows subtraction.

- If an ORDER BY phrase is provided, but neither a RANGE nor ROWS is specified, then the aggregation is done from the first row to the current row.
- See Window Definition for notes on how to define a moving-window of rows to process.

**Basic Usage**

In its simplest form, with just an "OVER()" phrase, an aggregation function works on all of the matching rows, running the column function specified. Thus, one gets both the detailed data, plus the SUM, or AVG, or whatever, of all the matching rows. In the following example, five rows are selected from the STAFF table. Along with various detailed fields, the query also gets sum summary data about the matching rows:

*Aggregation function, basic usage*

```
SELECT id
     , name
     , salary
     , SUM(salary) OVER() AS sum_sal
     , AVG(salary) OVER() AS avg_sal
     , MIN(salary) OVER() AS min_sal
     , MAX(salary) OVER() AS max_sal
     , COUNT(*)    OVER() AS #rows
FROM staff
WHERE id < 30
ORDER BY id;
```

*ANSWER*

| ID | NAME | SALARY | SUM_SAL | AVG_SAL | MIN_SAL | MAX_SAL | #ROWS |
|---|---|---|---|---|---|---|---|
| 10 | Sanders | 98357.50 | 254035.50 | 84678.50 | 77506.75 | 98357.50 | 3 |
| 20 | Pernal | 78171.25 | 254035.50 | 84678.50 | 77506.75 | 98357.50 | 3 |
| 30 | Marenghi | 77506.75 | 254035.50 | 84678.50 | 77506.75 | 98357.50 | 3 |

An aggregation function with just an "OVER()" phrase is logically equivalent to one that has an ORDER BY on a field that has the same value for all matching rows. To illustrate, in the following query, the four aggregation functions are all logically equivalent:

*Logically equivalent aggregation functions*

```
SELECT id
     , name
     , salary
     , SUM(salary) OVER()                                  AS sum1
     , SUM(salary) OVER(ORDER BY id * 0)                   AS sum2
     , SUM(salary) OVER(ORDER BY 'ABC')                    AS sum3
     , SUM(salary) OVER(ORDER BY 'ABC'
                        RANGE BETWEEN UNBOUNDED PRECEDING
                        AND UNBOUNDED FOLLOWING)           AS sum4
FROM staff
WHERE id < 60
ORDER BY id;
```

*ANSWER*

| ID | NAME | SALARY | SUM1 | SUM2 | SUM3 | SUM4 |
|---|---|---|---|---|---|---|
| 10 | Sanders | 98357.50 | 412701.30 | 412701.30 | 412701.30 | 412701.30 |
| 20 | Pernal | 78171.25 | 412701.30 | 412701.30 | 412701.30 | 412701.30 |
| 30 | Marenghi | 77506.75 | 412701.30 | 412701.30 | 412701.30 | 412701.30 |
| 40 | O'Brien | 78006.00 | 412701.30 | 412701.30 | 412701.30 | 412701.30 |
| 50 | Hanes | 80659.80 | 412701.30 | 412701.30 | 412701.30 | 412701.30 |

### 6.1.28. ORDER BY Usage

The ORDER BY phrase (see Order By, Group By, and Having for syntax) has two main purposes:

- It provides a set of values to do aggregations on. Each distinct value gets a new result.
- It gives a direction to the aggregation function processing (i.e. ASC or DESC).

In the next query, various aggregations are run on the DEPT field, which is not unique, and on the DEPT and NAME fields combined, which are unique (for these rows). Both ascending and descending aggregations are illustrated. Observe that the ascending fields sum or count up, while the descending fields sum down. Also observe that each aggregation field gets a separate result for each new set of rows, as defined in the ORDER BY phrase:

*Aggregation function, ORDER BY usage*

```
SELECT dept
     , name
     , salary
     , SUM(salary) OVER(ORDER BY dept)                AS sum1
     , SUM(salary) OVER(ORDER BY dept DESC)           AS sum2
     , SUM(salary) OVER(ORDER BY dept, NAME)          AS sum3
     , SUM(salary) OVER(ORDER BY dept DESC, name DESC) AS sum4
     , COUNT(*)    OVER(ORDER BY dept)                AS rw1
     , COUNT(*)    OVER(ORDER BY dept, NAME)          AS rw2
FROM staff
WHERE id < 60
ORDER BY dept
, name;
```

*ANSWER*

| DEPT | NAME | SALARY | SUM1 | SUM2 | SUM3 | SUM4 | RW1 | RW2 |
|---|---|---|---|---|---|---|---|---|
| 15 | Hanes | 80659.80 | 80659.80 | 412701.30 | 80659.80 | 412701.30 | 1 | 1 |
| 20 | Pernal | 78171.25 | 257188.55 | 332041.50 | 158831.05 | 332041.50 | 3 | 2 |
| 20 | Sanders | 98357.50 | 257188.55 | 332041.50 | 257188.55 | 253870.25 | 3 | 3 |
| 38 | Marenghi | 77506.75 | 412701.30 | 155512.75 | 334695.30 | 155512.75 | 5 | 4 |
| 38 | O’Brien | 78006.00 | 412701.30 | 155512.75 | 412701.30 | 78006.00 | 5 | 5 |

### 6.1.29. ROWS Usage

The ROWS phrase (see Window Definition for syntax) is used to limit the aggregation function to a subset of the matching rows. The set of rows to process are defined thus:

- **No ORDER BY**: UNBOUNDED PRECEDING to UNBOUNDED FOLLOWING.
- **ORDER BY only**: UNBOUNDED PRECEDING to CURRENT ROW.
- **No BETWEEN**: CURRENT ROW to "n" preceding/following row.
- **BETWEEN stmt**: From "n" to "n" preceding/following row. The end-point must be greater than or equal to the starting point.

The following query illustrates these concepts:

*ROWS usage examples*

```
SELECT id
     , years
     , AVG(years) OVER()                                                    AS "p_f"
     , AVG(years) OVER(ORDER BY id ROWS
                       BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS "p_f"
     , AVG(years) OVER(ORDER BY id)                                         AS "p_c"
     , AVG(years) OVER(ORDER BY id
                       ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)    AS "p_c"
     , AVG(years) OVER(ORDER BY id
                       ROWS UNBOUNDED PRECEDING)                            AS "p_c"
     , AVG(years) OVER(ORDER BY id
                       ROWS UNBOUNDED FOLLOWING)                            AS "c_f"
     , AVG(years) OVER(ORDER BY id
                       ROWS 2 FOLLOWING)                                    AS "c_2"
     , AVG(years) OVER(ORDER BY id
                       ROWS 1 PRECEDING)                                    AS "1_c"
     , AVG(years) OVER(ORDER BY id
                       ROWS BETWEEN 1 FOLLOWING AND 2 FOLLOWING)            AS "1_2"
FROM staff
WHERE dept IN (15,20)
AND id > 20
AND years > 1
ORDER BY id;
```

| ID | YEARS | p_f | p_f | p_c | p_c | p_c | c_f | c_2 | 1_c | 1_2 |
|---|---|---|---|---|---|---|---|---|---|---|
| 50 | 10 | 6 | 6 | 10 | 10 | 10 | 6 | 7 | 10 | 6 |
| 70 | 7 | 6 | 6 | 8 | 8 | 8 | 6 | 5 | 8 | 4 |
| 110 | 5 | 6 | 6 | 7 | 7 | 7 | 5 | 5 | 6 | 6 |
| 170 | 4 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 4 | 8 |
| 190 | 8 | 6 | 6 | 6 | 6 | 6 | 8 | 8 | 6 | - |

### 6.1.30. RANGE Usage

The RANGE phrase limits the aggregation result to a range of numeric values - defined relative to the value of the current row being processed (see Window Definition). The range is obtained by taking the value in the current row (defined by the ORDER BY expression) and adding to and/or subtracting from it, then seeing what other matching rows are in the range.

When using a RANGE, only one expression can be specified in the ORDER BY, and that expression must be numeric.

In the following example, the RANGE function adds to and/or subtracts from the DEPT field. For example, in the function that is used to populate the RG10 field, the current DEPT value is checked against the preceding DEPT values. If their value is within 10 digits of the current value, the related YEARS field is added to the SUM:

*RANGE usage*

```
SELECT dept
     , name
     , years
     , SMALLINT(SUM(years) OVER(ORDER BY dept
                                ROWS BETWEEN 1 PRECEDING AND CURRENT ROW))    AS row1
     , SMALLINT(SUM(years) OVER(ORDER BY dept
                                ROWS BETWEEN 2 PRECEDING AND CURRENT ROW))    AS row2
     , SMALLINT(SUM(years) OVER(ORDER BY dept
                                RANGE BETWEEN 1 PRECEDING AND CURRENT ROW))   AS rg01
     , SMALLINT(SUM(years) OVER(ORDER BY dept
                                RANGE BETWEEN 10 PRECEDING AND CURRENT ROW))  AS rg10
     , SMALLINT(SUM(years) OVER(ORDER BY dept
                                RANGE BETWEEN 20 PRECEDING AND CURRENT ROW))  AS rg20
     , SMALLINT(SUM(years) OVER(ORDER BY dept
                                RANGE BETWEEN 10 PRECEDING AND 20 FOLLOWING)) AS rg11
     , SMALLINT(SUM(years) OVER(ORDER BY dept
                                RANGE BETWEEN CURRENT ROW AND 20 FOLLOWING))  AS rg99
FROM staff
WHERE id < 100
AND years IS NOT NULL
ORDER BY dept
       , name;
```

ANSWER

| DEPT | NAME | YEARS | ROW1 | ROW2 | RG01 | RG10 | RG20 | RG11 | RG99 |
|---|---|---|---|---|---|---|---|---|---|
| 15 | Hanes | 10 | 10 | 10 | 17 | 17 | 17 | 32 | 32 |
| 15 | Rothman | 7 | 17 | 17 | 17 | 17 | 17 | 32 | 32 |
| 20 | Pernal | 8 | 15 | 25 | 15 | 32 | 32 | 43 | 26 |
| 20 | Sanders | 7 | 15 | 22 | 15 | 32 | 32 | 43 | 26 |
| 38 | Marengh | 5 | 12 | 20 | 11 | 11 | 26 | 17 | 17 |
| 38 | O’Brien | 6 | 11 | 18 | 11 | 11 | 26 | 17 | 17 |
| 42 | Koonitz | 6 | 12 | 17 | 6 | 17 | 17 | 17 | 6 |

Note the difference between the ROWS as RANGE expressions:

- The ROWS expression refers to the "n" rows before and/or after (within the partition), as defined by the ORDER BY.
- The RANGE expression refers to those before and/or after rows (within the partition) that are within an arithmetic range of the current row.

### 6.1.31. BETWEEN vs. ORDER BY

The BETWEEN predicate in an ordinary SQL statement is used to get those rows that have a value between the specified low-value (given first) and the high value (given last). Thus the predicate "BETWEEN 5 AND 10" may find rows, but the predicate "BETWEEN 10 AND 5" will never find any. The BETWEEN phrase in an aggregation function has a similar usage in that it defines the set of rows to be aggregated. But it differs in that the answer depends upon the related ORDER BY sequence, and a non-match returns a null value, not no-rows. Below is some sample SQL. Observe that the first two aggregations are ascending, while the last two are descending:

*BETWEEN and ORDER BY usage*

```
SELECT id
     , name
     , SMALLINT(SUM(id) OVER(ORDER BY id ASC
                             ROWS BETWEEN 1 PRECEDING AND CURRENT ROW)) AS apc
     , SMALLINT(SUM(id) OVER(ORDER BY id ASC
                             ROWS BETWEEN CURRENT ROW AND 1 FOLLOWING)) AS acf
     , SMALLINT(SUM(id) OVER(ORDER BY id DESC
                             ROWS BETWEEN 1 PRECEDING AND CURRENT ROW)) AS dpc
     , SMALLINT(SUM(id) OVER(ORDER BY id DESC
                             ROWS BETWEEN CURRENT ROW AND 1 FOLLOWING)) AS dcf
FROM staff
WHERE id     < 50
AND   years IS NOT NULL
ORDER BY id;
```

*ANSWER*

| ID | NAME | APC | ACF | DPC | DCF |
|---|---|---|---|---|---|
| 10 | Sanders | 10 | 30 | 30 | 10 |
| 20 | Pernal | 30 | 50 | 50 | 30 |
| 30 | Marenghi | 50 | 70 | 70 | 50 |
| 40 | O'Brien | 70 | 40 | 40 | 70 |

The following table illustrates the processing sequence in the above query. Each BETWEEN is applied from left to right, while the rows are read either from left to right (ORDER BY ID ASC) or right to left (ORDER BY ID DESC):

**ASC id (10,20,30,40)**

| READ ROWS, LEFT to RIGHT | 1ST-ROW | 2ND-ROW | 3RD-ROW | 4TH-ROW |
|---|---|---|---|---|
| 1 PRECEDING to CURRENT ROW | 10=10 | 10+20=30 | 20+30=40 | 30+40=70 |

| READ ROWS, LEFT to RIGHT | 1ST-ROW | 2ND-ROW | 3RD-ROW | 4TH-ROW |
|---|---|---|---|---|
| CURRENT ROW to 1 FOLLOWING | 10+20=30 | 20+30=50 | 30+40=70 | 40 =40 |

**DESC id (40,30,20,10)**

| READ ROWS, RIGHT to LEFT | 1ST-ROW | 2ND-ROW | 3RD-ROW | 4TH-ROW |
|---|---|---|---|---|
| 1 PRECEDING to CURRENT ROW | 20+10=30 | 30+20=50 | 40+30=70 | 40 =40 |
| CURRENT ROW to 1 FOLLOWING | 10 =10 | 20+10=30 | 30+20=50 | 40+30=70 |

Preceding row is always on LEFT of current row. Following row_ is always on RIGHT of current row.

The BETWEEN predicate, when used in an ordinary SQL statement, is not affected by the sequence of the input rows. But the BETWEEN phrase, when used in an aggregation function, is affected by the input sequence.

# Chapter 7. Scalar Functions

=== Introduction

Scalar functions act on a single row at a time. In this section we shall list all of the ones that come with Db2 and look in detail at some of the more interesting ones. Refer to the SQL Reference for information on those functions not fully described here.

## 7.1. Sample Data

The following self-defined view will be used throughout this section to illustrate how some of the following functions work. Observe that the view has a VALUES expression that defines the contents-three rows and nine columns.

*Sample View DDL - Scalar functions*

```
CREATE VIEW scalar
(d1, f1, s1, c1, v1, ts1, dt1, tm1, tc1) AS
WITH temp1 (n1, c1, t1) AS
( VALUES ( -2.4
         , 'ABCDEF'
         , '1996-04-22-23.58.58.123456')
       , ( +0.0
         , 'ABCD '
         , '1996-08-15-15.15.15.151515')
      , ( +1.8
        , 'AB'
        , '0001-01-01-00.00.00.000000')
)
SELECT DECIMAL(n1,3,1)
     , DOUBLE(n1)
     , SMALLINT(n1)
     , CHAR(c1,6)
     , VARCHAR(RTRIM(c1),6)
     , TIMESTAMP(t1)
     , DATE(t1)
     , TIME(t1)
     , CHAR(t1)
FROM    temp1;
```

Below are the view contents:

*Table 7. SCALAR view, contents (3 rows)*

| D1 | F1 | S1 | C1 | V1 | TS1 | DT1 | TM1 | TC1 |
|---|---|---|---|---|---|---|---|---|
| -2.4 | -2.4e+000 | -2 | ABCDEF | ABCDEF | 1996-04-22-23.58.58.123456 | 1996-04-22 | 23:58:58 | 1996-04-22-23.58.58.123456 |

| D1 | F1 | S1 | C1 | V1 | TS1 | DT1 | TM1 | TC1 |
|---|---|---|---|---|---|---|---|---|
| 0.0 | 0.0e+000 | 0 | ABCD | ABCD | 1996-08-15-15.15.15.151515 | 1996-08-15 | 15:15:15 | 1996-08-15-15.15.15.151515 |
| 1.8 | 1.8e+000 | 1 | AB | AB | 0001-01-01-00.00.00.000000 | 0001-01-01 | 00:00:00 | 0001-01-01-00.00.00.000000 |

# 7.2. Scalar Functions, Definitions

## 7.2.1. ABS or ABSVAL

Returns the absolute value of a number (e.g. -0.4 returns + 0.4). The output field type will equal the input field type (i.e. double input returns double output).

*ABS function examples*

```
SELECT d1      AS d1
     , ABS(D1) AS d2
     , f1      AS f1
     , ABS(f1) AS F2
FROM scalar;
```

*ANSWER (float output shortened)*

| D1 | D2 | F1 | F2 |
|---|---|---|---|
| -2.4 | 2.4 | -2.400e+0 | 2.400e+00 |
| 0.0 | 0.0 | 0.000e+0 | 0.000e+00 |
| 1.8 | 1.8 | 1.800e+0 | 1.800e+00 |

## 7.2.2. ACOS

Returns the arccosine of the argument as an angle expressed in radians. The output format is double.

## 7.2.3. ADD_DAYS

Returns a datetime value that represents the first argument plus a specified number of days.

## 7.2.4. ADD_HOURS

Returns a timestamp value that represents the first argument plus a specified number of hours.

### 7.2.5. ADD_MINUTES

Returns a timestamp value that represents the first argument plus a specified number of minutes.

### 7.2.6. ADD_MONTHS

Returns a datetime value that represents expression plus a specified number of months.

### 7.2.7. ADD_SECONDS

Returns a timestamp value that represents the first argument plus a specified number of seconds and fractional seconds.

### 7.2.8. ADD_YEARS

Returns a datetime value that represents the first argument plus a specified number of years.

### 7.2.9. AGE

Returns a numeric value that represents the number of full years, full months, and full days between the current timestamp and the argument.

### 7.2.10. ARRAY_DELETE

Deletes elements from an array.

### 7.2.11. ARRAY_FIRST

Returns the minimum array index value of the array.

### 7.2.12. ARRAY_LAST

Returns the maximum array index value of the array.

### 7.2.13. ARRAY_NEXT

Returns the next larger array index value for an array relative to the specified array index argument.

### 7.2.14. ARRAY_PRIOR

Returns the next smaller array index value for an array relative to the specified array index argument.

### 7.2.15. ASCII

Returns the ASCII code value of the leftmost input character. Valid input types are any valid character type up to 1 MEG. The output type is integer.

*ASCII function examples*

```
SELECT c1
     , ASCII(c1)           AS ac1
     , ASCII(SUBSTR(c1,2)) AS ac2
FROM scalar
WHERE c1 = 'ABCDEF';
```

*ANSWER*

| C1 | AC1 | AC2 |
|---|---|---|
| ABCDEF | 65 | 66 |

**The CHR function is the inverse of the ASCII function.**

### 7.2.16. ASIN

Returns the arcsine of the argument as an angle expressed in radians. The output format is double.

### 7.2.17. ATAN

Returns the arctangent of the argument as an angle expressed in radians. The output format is double.

### 7.2.18. ATAN2

Returns the arctangent of x and y coordinates, specified by the first and second arguments, as an angle, expressed in radians. The output format is double.

### 7.2.19. ATANH

Returns the hyperbolic acrctangent of the argument, where the argument is and an angle expressed in radians. The output format is double.

### 7.2.20. BIGINT

Converts the input value to bigint (big integer) format. The input can be either numeric or character. If character, it must be a valid representation of a number.

*BIGINT function example*

```
WITH temp (big) AS
(VALUES BIGINT(1)
  UNION ALL
SELECT big * 256
FROM temp
WHERE big < 1E16)
SELECT big
FROM temp;
```

*ANSWER*

| BIG |
|---|
| 1 |
| 256 |
| 65536 |
| 16777216 |
| 4294967296 |
| 1099511627776 |
| 281474976710656 |
| 72057594037927936 |

Converting certain float values to both BIGINT and decimal will result in different values being returned (see below). Both results are arguably correct, it is simply that the two functions use different rounding methods:

*Convert FLOAT to DECIMAL and BIGINT, SQL*

```
WITH temp (f1) AS
(VALUES FLOAT(1.23456789)
  UNION ALL
SELECT f1 * 100
FROM temp
WHERE f1 < 1E18
)
SELECT f1          AS float1
     , DEC(f1,19) AS decimal1
     , BIGINT(f1) AS bigint1
FROM temp;
```

| FLOAT1 | DECIMAL1 | BIGINT1 |
|---|---|---|
| +1.23456789000000E+000 | 1. | 1 |
| +1.23456789000000E+002 | 123. | 123 |

| FLOAT1 | DECIMAL1 | BIGINT1 |
|---|---|---|
| +1.23456789000000E+004 | 12345. | 12345 |
| +1.23456789000000E+006 | 1234567. | 1234567 |
| +1.23456789000000E+008 | 123456789. | 123456788 |
| +1.23456789000000E+010 | 12345678900. | 12345678899 |
| +1.23456789000000E+012 | 1234567890000. | 1234567889999 |
| +1.23456789000000E+014 | 123456789000000. | 123456788999999 |
| +1.23456789000000E+016 | 12345678900000000. | 12345678899999996 |
| +1.23456789000000E+018 | 1234567890000000000. | 1234567889999999488 |

See Floating Point Numbers for a discussion on floating-point number manipulation.

### 7.2.21. BINARY

Returns a fixed-length binary string representation of a string of any data type.

### 7.2.22. BIT Functions

There are five BIT functions:

- **BITAND** 1 if both arguments are 1.
- **BITANDNOT** Zero if bit in second argument is 1, otherwise bit in first argument.
- **BITOR** 1 if either argument is 1.
- **BITXOR** 1 if both arguments differ.
- **BITNOT** Returns opposite of the single argument.

The arguments can be SMALLINT (16 bits), INTEGER (32 bits), BIGINT (64 bits), or DECFLOAT (113 bits). The result is the same as the argument with the largest data type.

Negative numbers can be used in bit manipulation. For example the SMALLINT value -1 will have all 16 bits set to "1" (see example on Updating BIT Values). As their name implies, the BIT functions can be used to do bit-by-bit comparisons between two numbers:

*BIT functions examples*

```
WITH temp1 (b1, b2) AS
(VALUES ( 1, 0) ,( 0, 1)
      , ( 0, 0) ,( 1, 1)
      , ( 2, 1) ,(15,-7)
      , (15, 7) ,(-1, 1)
      , (15,63) ,(63,31)
      , (99,64) ,( 0,-2)
)
, temp2 (b1, b2) AS
 (SELECT SMALLINT(b1)
       , SMALLINT(b2)
  FROM temp1)
SELECT b1
     , b2
     , HEX(b1)          AS "hex1"
     , HEX(b2)          AS "hex2"
     , BITAND(b1,b2)    AS "and"
     , BITANDNOT(b1,b2) AS "ano"
     , BITOR(b1,b2)     AS "or"
     , BITXOR(b1,b2)    AS "xor"
FROM temp2;
```

*ANSWER*

| B1 | B2 | hex1 | hex2 | and | ano | or | xor |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0100 | 0000 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0000 | 0100 | 0 | 0 | 1 | 1 |
| 0 | 0 | 0000 | 0000 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0100 | 0100 | 1 | 0 | 1 | 0 |
| 2 | 1 | 0200 | 0100 | 0 | 2 | 3 | 3 |
| 15 | -7 | 0F00 | F9FF | 9 | 6 | -1 | -10 |
| 15 | 7 | 0F00 | 0700 | 7 | 8 | 15 | 8 |
| -1 | 1 | FFFF | 0100 | 1 | -2 | -1 | -2 |
| 15 | 63 | 0F00 | 3F00 | 15 | 0 | 63 | 48 |
| 63 | 31 | 3F00 | 1F00 | 31 | 32 | 63 | 32 |
| 99 | 64 | 6300 | 4000 | 64 | 35 | 99 | 35 |
| 0 | -2 | 0000 | FEFF | 0 | 0 | -2 | -2 |

### Displaying BIT Values

It can sometimes be hard to comprehend what a given BASE 10 value is in BIT format. To help, the following user-defined-function converts SMALLINT numbers to BIT values:

*Function to display SMALLINT bits*

```
CREATE FUNCTION bitdisplay(inparm SMALLINT)
RETURNS CHAR(16)
BEGIN ATOMIC
    DECLARE outstr VARCHAR(16);
    DECLARE inval INT;
    IF inparm >= 0 THEN
        SET inval = inparm;
    ELSE
        SET inval = INT(65536) + inparm;
    END IF;
    SET outstr = '';
    WHILE inval > 0 DO
        SET outstr = STRIP(CHAR(MOD(inval,2))) || outstr;
        SET inval = inval / 2;
    END WHILE;
    RETURN RIGHT(REPEAT('0',16) || outstr,16);
END!
```

Below is an example of the above function in use:

*BIT_DISPLAY function example*

```
WITH temp1 (b1) AS
(VALUES (32767)
     , (16383)
     , ( 4096)
     , ( 118)
     , ( 63)
     , ( 16)
     , ( 2)
     , ( 1)
     , ( 0)
     , ( -1)
     , ( -2)
     , ( -3)
     , ( -64)
     , (-32768)
)
, temp2 (b1) AS
(SELECT SMALLINT(b1)
FROM temp1
)
SELECT b1
     , HEX(b1)        AS "hex1"
     , BITDISPLAY(b1) AS "bit_display"
FROM temp2;
```

*ANSWER*

| B1 | hex1 | bit_display |
|---|---|---|
| 32767 | FF7F | 0111111111111111 |
| 16383 | FF3F | 0011111111111111 |
| 4096 | 0010 | 0001000000000000 |
| 118 | 7600 | 0000000001110110 |
| 63 | 3F00 | 0000000000111111 |
| 16 | 1000 | 0000000000010000 |
| 2 | 0200 | 0000000000000010 |
| 1 | 0100 | 0000000000000001 |
| 0 | 0000 | 0000000000000000 |
| -1 | FFFF | 1111111111111111 |
| -2 | FEFF | 1111111111111110 |
| -3 | FDFF | 1111111111111101 |
| -64 | C0FF | 1111111111000000 |
| -32768 | 0080 | 1000000000000000 |

#### Updating BIT Values

Use the BITXOR function to toggle targeted bits in a value. Use the BITANDNOT function to clear the same targeted bits. To illustrate, the next query uses these two functions to toggle and clear the last four bits, because the second parameter is 15, which is b"1111":

*Update bits #1*

```
WITH temp1 (b1) AS
( VALUES (32767)
       , (21845)
       , (4096)
       , (0)
       , (-1)
       , (-64)
)
, temp2 (b1, s15) AS
(SELECT SMALLINT(b1)
      , SMALLINT(15)
 FROM temp1
)
SELECT b1
     , BITDISPLAY(b1)                AS "b1_display"
     , BITXOR(b1,s15)                AS "xor"
     , BITDISPLAY(BITXOR(b1,s15))    AS "xor_display"
     , BITANDNOT(b1,s15)             AS "andnot"
     , BITDISPLAY(BITANDNOT(b1,s15)) AS "andnot_display"
FROM temp2;
```

Below is the answer:

| B1 | b1_display | xor | xor_display | andnot | andnot_display |
|---|---|---|---|---|---|
| 32767 | 0111111111111111 | 32752 | 0111111111110000 | 32752 | 0111111111110000 |
| 21845 | 0101010101010101 | 21850 | 0101010101011010 | 21840 | 0101010101010000 |
| 4096 | 0001000000000000 | 4111 | 0001000000001111 | 4096 | 0001000000000000 |
| 0 | 0000000000000000 | 15 | 0000000000001111 | 0 | 0000000000000000 |
| -1 | 1111111111111111 | -16 | 1111111111110000 | -16 | 1111111111110000 |
| -64 | 1111111111000000 | -49 | 1111111111001111 | -64 | 1111111111000000 |

The next query illustrate the use of the BITAND function to return those bits that match both parameters, and the BITNOT function to toggle all bits:

*Update bits #2, query*

```
WITH temp1 (b1) AS
(VALUES (32767)
      , (21845)
      , (4096)
      , (0)
      , (-1)
      , (-64)
)
, temp2 (b1, s15) AS
(SELECT SMALLINT(b1)
      , SMALLINT(15)
FROM temp1
)
SELECT b1
     , BITDISPLAY(b1)             AS "b1_display"
     , BITAND(b1,s15)             AS "and"
     , BITDISPLAY(BITAND(b1,s15)) AS "and_display"
     , BITNOT(b1)                 AS "not"
     , BITDISPLAY(BITNOT(b1))     AS "not_display"
FROM temp2;
```

Below is the answer:

| B1 | b1_display | and | and_display | not | not_display |
|---|---|---|---|---|---|
| 32767 | 0111111111111111 | 15 | 0000000000001111 | -32768 | 1000000000000000 |
| 21845 | 0101010101010101 | 5 | 0000000000000101 | -21846 | 1010101010101010 |
| 4096 | 0001000000000000 | 0 | 0000000000000000 | -4097 | 1110111111111111 |
| 0 | 0000000000000000 | 0 | 0000000000000000 | -1 | 1111111111111111 |
| -1 | 1111111111111111 | 15 | 0000000000001111 | 0 | 0000000000000000 |
| -64 | 1111111111000000 | 0 | 0000000000000000 | 63 | 0000000000111111 |

### 7.2.23. BLOB

Converts the input (1st argument) to a blob. The output length (2nd argument) is optional.

### 7.2.24. BOOLEAN

Returns the actual Boolean value that corresponds to a non-Boolean representation of a Boolean

value.

### 7.2.25. BTRIM

Removes the characters that are specified in a trim string from the beginning and end of a source string.

### 7.2.26. CARDINALITY

Returns a value of type BIGINT that is the number of elements in an array.

### 7.2.27. CEIL or CEILING

Returns the next smallest integer value that is greater than or equal to the input (e.g. 5.045 returns 6.000). The output field type will equal the input field type.

*CEIL function examples*

```
SELECT d1
     , CEIL(d1) AS d2
     , f1
     , CEIL(f1) AS f2
FROM scalar;
```

*ANSWER (float output shortened)*

| D1 | D2 | F1 | F2 |
|---|---|---|---|
| -2.4 | -2. | -2.400E+0 | -2.000E+0 |
| 0.0 | 0. | +0.000E+0 | +0.000E+0 |
| 1.8 | 2. | +1.800E+0 | +2.000E+0 |

Usually, when Db2 converts a number from one format to another, any extra digits on the right are truncated, not rounded. For example, the output of INTEGER(123.9) is 123. Use the CEIL or ROUND functions to avoid truncation.

### 7.2.28. CHAR

The CHAR function has a multiplicity of uses. The result is always a fixed-length character value, but what happens to the input along the way depends upon the input type:

- For character input, the CHAR function acts a bit like the SUBSTR function, except that it can only truncate starting from the left-most character. The optional length parameter, if provided, must be a constant or keyword.
- Date-time input is converted into an equivalent character string. Optionally, the external format can be explicitly specified (i.e. ISO, USA, EUR, JIS, or LOCAL).
- Integer and double input is converted into a left-justified character string.

- Decimal input is converted into a right-justified character string with leading zeros. The format of the decimal point can optionally be provided. The default decimal point is a dot. The '+' and '-' symbols are not allowed as they are used as sign indicators.

Below are some examples of the CHAR function in action:

*CHAR function examples - characters and numbers*

```
SELECT name
     , CHAR(name, 3)
     , comm
     , CHAR(comm)
     , CHAR(comm,'@')
FROM staff
WHERE id BETWEEN 80 AND 100
ORDER BY id;
```

*ANSWER*

| NAME | 2 | COMM | 4 | 5 |
|---|---|---|---|---|
| James | Jam | 128.20 | 00128.20 | 00128@20 |
| Koonitz | Koo | 1386.70 | 01386.70 | 01386@70 |
| Plotz | Plo | - | - | - |

The CHAR function treats decimal numbers quite differently from integer and real numbers. In particular, it right-justifies the former (with leading zeros), while it left-justifies the latter (with trailing blanks). The next example illustrates this point:

*CHAR function examples - positive numbers_*

```
WITH temp1 (n) AS
(VALUES (3)
  UNION ALL
 SELECT n * n
 FROM temp1
 WHERE n < 9000
)
SELECT n AS int
     , CHAR(INT(n)) AS char_int
     , CHAR(FLOAT(n)) AS char_flt
     , CHAR(DEC(n)) AS char_dec
FROM temp1;
```

*ANSWER*

| INT | CHAR_INT | CHAR_FLT | CHAR_DEC |
|---|---|---|---|
| 3 | 3 | 3.0E0 | 00000000003. |

| INT | CHAR_INT | CHAR_FLT | CHAR_DEC |
|---|---|---|---|
| 9 | 9 | 9.0E0 | 00000000009. |
| 81 | 81 | 8.1E1 | 00000000081. |
| 6561 | 6561 | 6.561E3 | 00000006561. |
| 43046721 | 43046721 | 4.3046721E7 | 00043046721. |

Negative numeric input is given a leading minus sign. This messes up the alignment of digits in the column (relative to any positive values). In the following query, a leading blank is put in front of all positive numbers in order to realign everything:

*Align CHAR function output - numbers_*

```
WITH temp1 (n1, n2) AS
(VALUES (SMALLINT(+3)
       , SMALLINT(-7))
  UNION ALL
SELECT n1 * n2
     , n2
FROM temp1
WHERE n1 < 300
)
SELECT n1
     , CHAR(n1) AS i1
     , CASE
         WHEN n1 < 0 THEN CHAR(n1)
         ELSE '+' CONCAT CHAR(n1)
       END AS i2
     , CHAR(DEC(n1)) AS d1
     , CASE
         WHEN n1 < 0 THEN CHAR(DEC(n1))
         ELSE '+' CONCAT CHAR(DEC(n1))
       END AS d2
FROM temp1;
```

*ANSWER*

| N1 | I1 | I2 | D1 | D2 |
|---|---|---|---|---|
| 3 | 3 | +3 | 00003. | +00003. |
| -21 | -21 | -21 | -00021. | -00021. |
| 147 | 147 | +147 | 00147. | +00147. |
| -1029 | -1029 | -1029 | -01029. | -01029. |
| 7203 | 7203 | +7203 | 07203. | +07203. |

Both the I2 and D2 fields above will have a trailing blank on all negative values - that was added during the concatenation operation. The RTRIM function can be used to remove it.

**DATE-TIME Conversion**

The CHAR function can be used to convert a date-time value to character. If the input is not a timestamp, the output layout can be controlled using the format option:

- **ISO**: International Standards Organization.
- **USA**: American.
- **EUR**: European, which is usually the same as ISO.
- **JIS**: Japanese Industrial Standard, which is usually the same as ISO.
- **LOCAL**: Whatever your computer is set to.

Below are some DATE examples:

*CHAR function examples - date value*

```
SELECT CHAR(CURRENT DATE,ISO) AS iso
     , CHAR(CURRENT DATE,EUR) AS eur
     , CHAR(CURRENT DATE,JIS) AS jis
     , CHAR(CURRENT DATE,USA) AS usa
FROM sysibm.sysdummy1;
```

*ANSWER*

| ISO | EUR | JIS | USA |
|---|---|---|---|
| 2005-11-30 | 30.11.2005 | 2005-11-30 | 11/30/2005 |

Below are some TIME examples: .CHAR function examples - time value

```
SELECT CHAR(CURRENT TIME,ISO) AS iso
     , CHAR(CURRENT TIME,EUR) AS eur
     , CHAR(CURRENT TIME,JIS) AS jis
     , CHAR(CURRENT TIME,USA) AS usa
FROM sysibm.sysdummy1;
```

*ANSWER*

| ISO | EUR | JIS | USA |
|---|---|---|---|
| 19.42.21 | 19.42.21 | 19:42:21 | 07:42 PM |

A timestamp cannot be formatted to anything other than ISO output:

*CHAR function example - timestamp value*

```
SELECT CHAR(CURRENT TIMESTAMP) AS TS
FROM sysibm.sysdummy1;
```

*ANSWER*

| TS |
|---|
| 2005-11-30-19.42.21.873002 |

Converting a date or time value to character, and then ordering the set of matching rows can result in unexpected orders. See CASE Checks in Wrong Sequence for details.

**CHAR vs. DIGITS - A Comparison**

Numeric input can be converted to character using either the DIGITS or the CHAR function, though the former does not support float. Both functions work differently, and neither gives perfect output. The CHAR function doesn't properly align up positive and negative numbers, while the DIGITS function looses both the decimal point and sign indicator:

*DIGITS vs. CHAR*

```
SELECT d2
    , CHAR(d2)   AS cd2
    , DIGITS(d2) AS dd2
FROM
    (SELECT DEC(d1, 4, 1) AS d2
     FROM scalar
    ) AS xxx
ORDER BY 1;
```

*ANSWER*

| D2 | CD2 | DD2 |
|---|---|---|
| -2.4 | -002.4 | 0024 |
| 0.0 | 000.0 | 0000 |
| 1.8 | 001.8 | 0018 |

Neither the DIGITS nor the CHAR function do a great job of converting numbers to characters. See Convert Number to Character for some user-defined functions that can be used instead.

## 7.2.29. CHARACTER_LENGTH

This function is similar to the LENGTH function, except that it works with different encoding schemas. The result is an integer value that is the length of the input string.

*CHARACTER_LENGTH function example*

```
WITH temp1 (c1) AS
(VALUES (CAST('ÁÉÌ' AS VARCHAR(10)))
)
SELECT c1                          AS C1
     , LENGTH(c1)                  AS LEN
     , OCTET_LENGTH(c1)            AS OCT
     , CHAR_LENGTH(c1,OCTETS)      AS L08
     , CHAR_LENGTH(c1,CODEUNITS16) AS L16
     , CHAR_LENGTH(c1,CODEUNITS32) AS L32
FROM temp1;
```

*ANSWER*

| C1 | LEN | OCT | L08 | L16 | L32 |
|---|---|---|---|---|---|
| ÁÉÌ | 6 | 6 | 6 | 3 | 3 |

### 7.2.30. CHR

Converts integer input in the range 0 through 255 to the equivalent ASCII character value. An input value above 255 returns 255. The ASCII function (see above) is the inverse of the CHR function.

*CHR function examples*

```
SELECT 'A' AS "c"
     , ASCII('A') AS "c>n"
     , CHR(ASCII('A')) AS "c>n>c"
     , CHR(333) AS "nl"
FROM staff
WHERE id = 10;
```

*ANSWER*

| C | C>N | C>N>C | NL |
|---|---|---|---|
| A | 65 | A | □ |

At present, the CHR function has a bug that results in it not returning a null value when the input value is greater than 255.

### 7.2.31. CLOB

Converts the input (1st argument) to a CLOB. The output length (2nd argument) is optional. If the input is truncated during conversion, a warning message is issued. For example, in the following example the second CLOB statement will induce a warning for the first two lines of input because they have non-blank data after the third byte:

*CLOB function examples*

```
SELECT c1
     , CLOB(c1)   AS cc1
     , CLOB(c1,3) AS cc2
FROM scalar;
```

*ANSWER*

| C1 | CC1 | CC2 |
|---|---|---|
| ABCDEF | ABCDEF | ABC |
| ABCD | ABCD | ABC |
| AB | AB | AB |

The Db2BATCH command processor dies a nasty death whenever it encounters a CLOB field in the output. If possible, convert to VARCHAR first to avoid this problem.

### 7.2.32. COALESCE

Returns the first non-null value in a list of input expressions (reading from left to right). Each expression is separated from the prior by a comma. All input expressions must be compatible. VALUE is a synonym for COALESCE.

*COALESCE function example*

```
SELECT id
     , comm
     , COALESCE(comm, 0)
FROM staff
WHERE id < 30
ORDER BY id;
```

*ANSWER*

| ID | COMM | 3 |
|---|---|---|
| 10 | - | 0.00 |
| 20 | 612.45 | 612.45 |

A CASE expression can be written to do exactly the same thing as the COALESCE function. The following SQL statement shows two logically equivalent ways to replace nulls:

*COALESCE and equivalent CASE expression*

```
WITH temp1(c1,c2,c3) AS
(VALUES (CAST(NULL AS SMALLINT)
       , CAST(NULL AS SMALLINT)
       , CAST(10   AS SMALLINT))
)
SELECT COALESCE(c1, c2, c3)          AS cc1
     , CASE
         WHEN c1 IS NOT NULL THEN c1
         WHEN c2 IS NOT NULL THEN c2
         WHEN c3 IS NOT NULL THEN c3
       END                           AS cc2
FROM temp1;
```

*ANSWER*

| CC1 | CC2 |
|---|---|
| 10 | 10 |

Be aware that a field can return a null value, even when it is defined as not null. This occurs if a column function is applied against the field, and no row is returned:

*NOT NULL field returning null value*

```
SELECT COUNT(*)             AS #rows
     , MIN(id)              AS min_id
     , COALESCE(MIN(id), -1) AS ccc_id
FROM staff
WHERE id < 5;
```

*ANSWER*

| #ROWS | MIN_ID | CCC_ID |
|---|---|---|
| 0 | - | -1 |

### 7.2.33. COLLATION_KEY

Returns a VARBINARY string that represents the collation key of the expression argument, in the specified collation.

### 7.2.34. COLLATION_KEY_BIT

Returns a VARCHAR FOR BIT DATA string that is the collation sequence of the first argument in the function. There three parameters:

- String to be evaluated.
- Collation sequence to use (must be valid).

- Length of output (optional).

The following query displays three collation sequences:

- All flavors of a given character as the same (i.e. "a" = "A" = "Ä").
- Upper and lower case characters are equal, but sort lower than accented characters.
- All variations of a character have a different collation value.

Now for the query:

*COLLATION_KEY_BIT function example*

```
WITH temp1 (c1) As
(VALUES ('a'), ('A'), ('Á'), ('Ä'), ('b'))
SELECT c1
     , COLLATION_KEY_BIT(c1, 'UCA400R1_S1', 9) AS "a=A=Á=Ä"
     , COLLATION_KEY_BIT(c1, 'UCA400R1_S2' ,9) AS "a=A<Á<Ä"
     , COLLATION_KEY_BIT(c1, 'UCA400R1_S3' ,9) AS "a<A<Á<Ä"
FROM temp1
ORDER BY COLLATION_KEY_BIT(c1, 'UCA400R1_S3');
```

Below is the answer:

| C1 | a=A=Á=Ä | a=A<Á<Ä | a<A<Á<Ä |
|---|---|---|---|
| a | x'2600' | x'26010500' | x'260105010500' |
| A | x'2600' | x'26010500' | x'260105018F00' |
| Á | x'2600' | x'2601868D00' | x'2601868D018F0500' |
| Ä | x'2600' | x'2601869D00' | x'2601869D018F0500' |
| b | x'2800' | x'28010500' | x'280105010500' |

## 7.2.35. COMPARE_DECFLOAT

Compares two DECFLOAT expressions and returns a SMALLINT number:

- **0** if both values exactly equal (i.e. no trailing-zero differences)
- **1** if the first value is less than the second value.
- **2** if the first value is greater than the second value.
- **3** if the result is unordered (i.e. either argument is NaN or sNaN).

Query

-COMPARE_DECFLOAT function example

```
WITH temp1 (d1, d2) AS
(VALUES (DECFLOAT(+1.0), DECFLOAT(+1.0))
      , (DECFLOAT(+1.0), DECFLOAT(+1.00))
      , (DECFLOAT(-1.0), DECFLOAT(-1.00))
      , (DECFLOAT(+0.0), DECFLOAT(+0.00))
      , (DECFLOAT(-0.0), DECFLOAT(-0.00))
      , (DECFLOAT(1234), +infinity)
      , (+infinity, +infinity)
      , (+infinity, -infinity)
      , (DECFLOAT(1234), -NaN)
)
SELECT COMPARE_DECFLOAT(d1, d2) AS Result
FROM temp1;
```

*ANSWER*

| Result |
|---|
| 0 |
| 2 |
| 1 |
| 2 |
| 2 |
| 1 |
| 0 |
| 2 |
| 3 |

Several values that compare as "less than" or "greater than" above are equal in the usual sense. See the section on DECFLOAT arithmetic for details (see DECFLOAT Arithmetic).

### 7.2.36. CONCAT

Joins two strings together. The CONCAT function has both "infix" and "prefix" notations. In the former case, the verb is placed between the two strings to be acted upon. In the latter case, the two strings come after the verb. Both syntax flavours are illustrated below:

*CONCAT function examples*

```
SELECT 'A' || 'B'
     , 'A' CONCAT 'B'
     , CONCAT('A','B')
     , 'A' || 'B' || 'C'
     , CONCAT(CONCAT('A','B'),'C')
FROM staff
WHERE id = 10;
```

*ANSWER*

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| AB | AB | AB | ABC | ABC |

Note that the "||" keyword can not be used with the prefix notation. This means that "||('a','b')" is not valid while "CONCAT('a','b')" is.

#### Using CONCAT with ORDER BY

When ordinary character fields are concatenated, any blanks at the end of the first field are left in place. By contrast, concatenating varchar fields removes any (implied) trailing blanks. If the result of the second type of concatenation is then used in an ORDER BY, the resulting row sequence will probably be not what the user intended. To illustrate:

*CONCAT used with ORDER BY - wrong output sequence*

```
WITH temp1 (col1, col2) AS
(VALUES
  ('A' , 'YYY')
, ('AE', 'OOO')
, ('AE', 'YYY')
)
SELECT col1
     , col2
     , col1 CONCAT col2 AS col3
FROM temp1
ORDER BY col3;
```

*ANSWER*

| COL1 | COL2 | COL3 |
|---|---|---|
| AE | OOO | AEOOO |
| AE | YYY | AEYYY |
| A | YYY | AYYY |

Converting the fields being concatenated to character gets around this problem:

*CONCAT used with ORDER BY - correct output sequence*

```
WITH temp1 (col1, col2) AS
(VALUES
  ('A' , 'YYY')
, ('AE', 'OOO')
, ('AE', 'YYY')
)
SELECT col1
     , col2
     , CHAR(col1, 2) CONCAT CHAR(col2, 3) AS col3
FROM temp1
ORDER BY col3;
```

*ANSWER*

| COL1 | COL2 | COL3 |
|---|---|---|
| A | YYY | A YYY |
| AE | OOO | AEOOO |
| AE | YYY | AEYYY |

Never do an ORDER BY on a concatenated set of variable length fields. The resulting row sequence is probably not what the user intended (see above).

### 7.2.37. COS

Returns the cosine of the argument where the argument is an angle expressed in radians. The output format is double.

*RADIAN, COS, and SIN functions example*

```
WITH temp1(n1) AS
(VALUES (0)
   UNION ALL
 SELECT n1 + 10
 FROM temp1
 WHERE n1 < 90
)
SELECT n1
     , DEC(RADIANS(n1),4,3)      AS ran
     , DEC(COS(RADIANS(n1)),4,3) AS cos
     , DEC(SIN(RADIANS(n1)),4,3) AS sin
FROM temp1;
```

*ANSWER*

| N1 | RAN | COS | SIN |
|---|---|---|---|
| 0 | 0.000 | 1.000 | 0.000 |
| 10 | 0.174 | 0.984 | 0.173 |
| 20 | 0.349 | 0.939 | 0.342 |
| 30 | 0.523 | 0.866 | 0.500 |
| 40 | 0.698 | 0.766 | 0.642 |
| 50 | 0.872 | 0.642 | 0.766 |
| 60 | 1.047 | 0.500 | 0.866 |
| 70 | 1.221 | 0.342 | 0.939 |
| 80 | 1.396 | 0.173 | 0.984 |
| 90 | 1.570 | 0.000 | 1.000 |

### 7.2.38. COSH

Returns the hyperbolic cosine for the argument, where the argument is an angle expressed in radians. The output format is double.

### 7.2.39. COT

Returns the cotangent of the argument where the argument is an angle expressed in radians. The output format is double.

### 7.2.40. CURSOR_ROWCOUNT

Returns the cumulative count of all rows fetched by the specified cursor since the cursor was opened.

### 7.2.41. DATAPARTITIONNUM

Returns the sequence number of the partition in which the row resides.

### 7.2.42. DATE

Converts the input into a date value. The nature of the conversion process depends upon the input type and length:

- Timestamp and date input have the date part extracted.
- Char or varchar input that is a valid string representation of a date or a timestamp (e.g. "1997-12-23") is converted as is.
- Char or varchar input that is seven bytes long is assumed to be a Julian date value in the format yyyynnn where yyyy is the year and nnn is the number of days since the start of the year (in the range 001 to 366).
- Numeric input is assumed to have a value which represents the number of days since the date

"0001-01-01" inclusive. All numeric types are supported, but the fractional part of a value is ignored (e.g. 12.55 becomes 12 which converts to "0001-01-12").

If the input can be null, the output will also support null. Null values convert to null output.

*DATE function example - timestamp input*

```
SELECT ts1
     , DATE(ts1) AS dt1
FROM scalar;
```

*ANSWER*

| TS1 | DT1 |
|---|---|
| 1996-04-22-23.58.58.123456 | 1996-04-22 |
| 1996-08-15-15.15.15.151515 | 1996-08-15 |
| 0001-01-01-00.00.00.000000 | 0001-01-01 |

*DATE function example - numeric input*

```
WITH temp1(n1) AS
(VALUES
  (000001)
, (728000)
, (730120)
)
SELECT n1
     , DATE(n1) AS d1
FROM temp1;
```

*ANSWER*

| N1 | D1 |
|---|---|
| 1 | 0001-01-01 |
| 728000 | 1994-03-13 |
| 730120 | 2000-01-01 |

### 7.2.43. DATE_PART

Returns a portion of a datetime based on its arguments. It extracts the subfield that is specified from the date, time, timestamp, and duration values.

### 7.2.44. DATE_TRUNC

Truncates a date, time, or timestamp value to the specified time unit.

### 7.2.45. DAY

Returns the day (as in day of the month) part of a date (or equivalent) value. The output format is integer.

*DAY function examples*

```
SELECT dt1
     , DAY(dt1) AS day1
FROM scalar
WHERE DAY(dt1) > 10;
```

*ANSWER*

| DT1 | DAY1 |
|---|---|
| 1996-04-22 | 22 |
| 1996-08-15 | 15 |

If the input is a date or timestamp, the day value must be between 1 and 31. If the input is a date or timestamp duration, the day value can ran from -99 to +99, though only -31 to +31 actually make any sense:

*DAY function, using date-duration input*

```
SELECT dt1
     , DAY(dt1)               AS day1
     , dt1 -'1996-04-30'      AS dur2
     , DAY(dt1 -'1996-04-30') AS day2
FROM scalar
WHERE DAY(dt1) > 10
ORDER BY dt1;
```

*ANSWER*

| DT1 | DAY1 | DUR2 | DAY2 |
|---|---|---|---|
| 1996-04-22 | 22 | -8. | -8 |
| 1996-08-15 | 15 | 315. | 15 |

A date-duration is what one gets when one subtracts one date from another. The field is of type decimal(8), but the value is not really a number. It has digits in the format: YYYYMMDD, so in the above query the value "315" represents 3 months, 15 days.

### 7.2.46. DAYNAME

Returns the name of the day (e.g. Friday) as contained in a date (or equivalent) value. The output format is varchar(100).

*DAYNAME function example*

```
SELECT dt1
     , DAYNAME(dt1)         AS dy1
     , LENGTH(DAYNAME(dt1)) AS dy2
FROM scalar
WHERE DAYNAME(dt1) LIKE '%a%y'
ORDER BY dt1;
```

*ANSWER*

| DT1 | DY1 | DY2 |
|---|---|---|
| 0001-01-01 | Monday | 6 |
| 1996-04-22 | Monday | 6 |
| 1996-08-15 | Thursday | 8 |

### 7.2.47. DAYOFMONTH

Returns an integer between 1 and 31 that represents the day of the month.

### 7.2.48. DAYOFWEEK

Returns a number that represents the day of the week (where Sunday is 1 and Saturday is 7) from a date (or equivalent) value. The output format is integer.

*DAYOFWEEK function example*

```
SELECT dt1
     , DAYOFWEEK(dt1) AS dwk
     , DAYNAME(dt1)   AS dnm
FROM scalar
ORDER BY dwk
       , dnm;
```

*ANSWER*

| DT1 | DWK | DNM |
|---|---|---|
| 1996-04-22 | 2 | Monday |
| 0001-01-01 | 2 | Saturday |
| 1996-08-15 | 5 | Thursday |

### 7.2.49. DAYOFWEEK_ISO

Returns an integer value that represents the day of the "ISO" week. An ISO week differs from an ordinary week in that it begins on a Monday (i.e. day-number = 1) and it neither ends nor begins at the exact end of the year. Instead, the final ISO week of the prior year will continue into the new

year. This often means that the first days of the year have an ISO week number of 52, and that one gets more than seven days in a week for ISO week 52.

*DAYOFWEEK_ISO function example*

```
WITH temp1 (n) AS
(VALUES (0)
   UNION ALL
 SELECT n+1
 FROM temp1
 WHERE n < 9
),
temp2 (dt1) AS
(VALUES(DATE('1999-12-25'))
     , (DATE('2000-12-24'))
),
temp3 (dt2) AS
(SELECT dt1 + n DAYS
 FROM temp1
    , temp2)
SELECT CHAR(dt2,ISO)             AS date
     , SUBSTR(DAYNAME(dt2),1,3) AS day
     , WEEK(dt2)                AS w
     , DAYOFWEEK(dt2)           AS d
     , WEEK_ISO(dt2)            AS wi
     , DAYOFWEEK_ISO(dt2)       AS di
FROM
temp3
ORDER BY 1;
```

*ANSWER*

| DATE | DAY | W | D | WI | DI |
|---|---|---|---|---|---|
| 1999-12-25 | Sat | 52 | 7 | 51 | 6 |
| 1999-12-26 | Sun | 53 | 1 | 51 | 7 |
| 1999-12-27 | Mon | 53 | 2 | 52 | 1 |
| 1999-12-28 | Tue | 53 | 3 | 52 | 2 |
| 1999-12-29 | Wed | 53 | 4 | 52 | 3 |
| 1999-12-30 | Thu | 53 | 5 | 52 | 4 |
| 1999-12-31 | Fri | 53 | 6 | 52 | 5 |
| 2000-01-01 | Sat | 1 | 7 | 52 | 6 |
| 2000-01-02 | Sun | 2 | 1 | 52 | 7 |
| 2000-01-03 | Mon | 2 | 2 | 1 | 1 |
| 2000-12-24 | Sun | 53 | 1 | 51 | 7 |

| DATE | DAY | W | D | WI | DI |
|---|---|---|---|---|---|
| 2000-12-25 | Mon | 53 | 2 | 52 | 1 |
| 2000-12-26 | Tue | 53 | 3 | 52 | 2 |
| 2000-12-27 | Wed | 53 | 4 | 52 | 3 |
| 2000-12-28 | Thu | 53 | 5 | 52 | 4 |
| 2000-12-29 | Fri | 53 | 6 | 52 | 5 |
| 2000-12-30 | Sat | 53 | 7 | 52 | 6 |
| 2000-12-31 | Sun | 54 | 1 | 52 | 7 |
| 2001-01-01 | Mon | 1 | 2 | 1 | 1 |
| 2001-01-02 | Tue | 1 | 3 | 1 | 2 |

### 7.2.50. DAYOFYEAR

Returns a number that is the day of the year (from 1 to 366) from a date (or equivalent) value. The output format is integer.

*DAYOFYEAR function example*

```
SELECT dt1
     , DAYOFYEAR(dt1) AS dyr
FROM scalar
ORDER BY dyr;
```

*ANSWER*

| DT1 | DYR |
|---|---|
| 0001-01-01 | 1 |
| 1996-04-22 | 113 |
| 1996-08-15 | 228 |

### 7.2.51. DAYS

Converts a date (or equivalent) value into a number that represents the number of days since the date "0001-01-01" inclusive. The output format is INTEGER.

*DAYS function example*

```
SELECT dt1
     , DAYS(dt1) AS dy1
FROM scalar
ORDER BY dy1
       , dt1;
```

*ANSWER*

| DT1 | DY1 |
|---|---|
| 0001-01-01 | 1 |
| 1996-04-22 | 728771 |
| 1996-08-15 | 728886 |

The DATE function can act as the inverse of the DAYS function. It can convert the DAYS output back into a valid date.

### 7.2.52. DAYS_BETWEEN

Returns the number of full days between the specified arguments.

### 7.2.53. DAYS_TO_END_OF_MONTH

Returns the number of days to the end of the month.

### 7.2.54. DBCLOB

Converts the input (1st argument) to a dbclob. The output length (2nd argument) is optional.

### 7.2.55. DBPARTITIONNUM

Returns the partition number of the row. The result is zero if the table is not partitioned. The output is of type integer, and is never null.

*DBPARTITIONNUM function example*

```
SELECT DBPARTITIONNUM(id) AS dbnum
FROM staff
WHERE id = 10;
```

*ANSWER*

| DBNUM |
|---|
| 0 |

The DBPARTITIONNUM function will generate a SQL error if the column/row used can not be related directly back to specific row in a real table. Therefore, one can not use this function on fields in GROUP BY statements, nor in some views. It can also cause an error when used in an outer join, and the target row failed to match in the join.

### 7.2.56. DECFLOAT

Converts a character or numeric expression to DECFLOAT.

The first parameter is the input expression. The second is the number of digits of precision (default = 34). And the third is the decimal character value (default = '.').

*DECFLOAT function example*

```
SELECT DECFLOAT(+123.4)
     , DECFLOAT(1.0, 16)
     , DECFLOAT(1.0000, 16)
     , DECFLOAT(1.2e-3, 34)
     , DECFLOAT('1.2e-3', 34)
     , DECFLOAT(-1E3, 34)
     , DECFLOAT('-1E3', 34)
     , DECFLOAT('12.5', 16)
     , DECFLOAT('12#5', 16, '#')
FROM sysibm.sysdummy1;
```

*ANSWER*

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|
| 123.4 | 1.0 | 1.0000 | 0.0011999999999999999 | 0.0012 | -1000 | -1E+3 | 12.5 | 12.5 |

The function does not always precisely convert floating-point numeric values to their DECFLOAT equivalent (see example above). Use character conversion instead.

### 7.2.57. DECFLOAT_FORMAT

Returns a DECFLOAT(34) value that is based on the interpretation of the input string using the specified format.

### 7.2.58. DEC or DECIMAL

Converts either character or numeric input to decimal. When the input is of type character, the decimal point format can be specified.

*DECIMAL function examples*

```
WITH temp1(n1, n2, c1, c2) AS
(VALUES
 (123, 1E2, '123.4', '567$8')
)
SELECT DEC(n1, 3)         AS dec1
     , DEC(n2, 4, 1)      AS dec2
     , DEC(c1, 4, 1)      AS dec3
     , DEC(c2, 4, 1, '$') AS dec4
FROM temp1;
```

*ANSWER*

| DEC1 | DEC2 | DEC3 | DEC4 |
|---|---|---|---|
| 123. | 100.0 | 123.4 | 567.8 |

Converting a floating-point number to decimal may get different results from converting the same number to integer. See Floating Point Numbers for a discussion of this issue.

### 7.2.59. DECODE

The DECODE function is a simplified form of the CASE expression. The first parameter is the expression to be evaluated. This is followed by pairs of "before" and "after" expressions. At the end is the "else" result:

*DECODE function example*

```
SELECT firstnme
     , sex
     , CASE sex
         WHEN 'F' THEN 'FEMALE'
         WHEN 'M' THEN 'MALE'
         ELSE '?'
       END AS sex2
     , DECODE(sex, 'F', 'FEMALE', 'M', 'MALE', '?') AS sex3
FROM employee
WHERE firstnme < 'D'
ORDER BY firstnme;
```

*ANSWER*

| FIRSTNME | SEX | SEX2 | SEX3 |
|---|---|---|---|
| BRUCE | M | MALE | MALE |
| CHRISTINE | F | FEMALE | FEMALE |

### 7.2.60. DECRYPT_BIN and DECRYPT_CHAR

Decrypts data that has been encrypted using the ENCRYPT function. Use the BIN function to decrypt binary data (e.g. BLOBS, CLOBS) and the CHAR function to do character data. Numeric data cannot be encrypted.

If the password is null or not supplied, the value of the encryption password special register will be used. If it is incorrect, a SQL error will be generated.

*DECRYPT_CHAR function example*

```
SELECT id
     , name
     , DECRYPT_CHAR(name2,'CLUELESS') AS name3
     , GETHINT(name2) AS hint
     , name2
FROM
(SELECT id
      , name
      , ENCRYPT(name, 'CLUELESS', 'MY BOSS') AS name2
FROM staff
WHERE id < 30
) AS xxx
ORDER BY id;
```

### 7.2.61. DEGREES

Returns the number of degrees converted from the argument as expressed in radians. The output format is double.

### 7.2.62. DEREF

Returns an instance of the target type of the argument.

### 7.2.63. DIFFERENCE

Returns the difference between the sounds of two strings as determined using the SOUNDEX function. The output (of type integer) ranges from 4 (good match) to zero (poor match).

*DIFFERENCE function example*

```
SELECT a.name AS n1
     , SOUNDEX(a.name) AS s1
     , b.name AS n2
     , SOUNDEX(b.name) AS s2
     , DIFFERENCE (a.name,b.name) AS df
FROM staff a
   , staff b
WHERE a.id = 10
AND b.id > 150
AND b.id < 250
ORDER BY df DESC
       , n2 ASC;
```

*ANSWER*

| N1 | S1 | N2 | S2 | DF |
|---|---|---|---|---|
| Sanders | S536 | Sneider | S536 | 4 |
| Sanders | S536 | Smith | S530 | 3 |
| Sanders | S536 | Lundquist | L532 | 2 |
| Sanders | S536 | Daniels | D542 | 1 |
| Sanders | S536 | Molinare | M456 | 1 |
| Sanders | S536 | Scoutten | S350 | 1 |
| Sanders | S536 | Abrahams | A165 | 0 |
| Sanders | S536 | Kermisch | K652 | 0 |
| Sanders | S536 | Lu | L000 | 0 |

The difference function returns one of five possible values. In many situations, it would be imprudent to use a value with such low granularity to rank values.

### 7.2.64. DIGITS

Converts an integer or decimal value into a character string with leading zeros. Both the sign indicator and the decimal point are lost in the translation.

*DIGITS function examples*

```
SELECT s1
     , DIGITS(s1) AS ds1
     , d1
     , DIGITS(d1) AS dd1
FROM scalar;
```

*ANSWER*

| S1 | DS1 | D1 | DD1 |
|---|---|---|---|
| 2 | 00002 | -2.4 | 024 |
| 0 | 00000 | 0.0 | 000 |
| 1 | 00001 | 1.8 | 018 |

The CHAR function can sometimes be used as alternative to the DIGITS function. Their output differs slightly - see Convert Number to Character for a comparison.

Neither the DIGITS nor the CHAR function do a great job of converting numbers to characters. See Convert Number to Character for some user-defined functions that can be used instead.

### 7.2.65. DOUBLE or DOUBLE_PRECISION

Converts numeric or valid character input to type double. This function is actually two with the same name. The one that converts numeric input is a SYSIBM function, while the other that handles character input is a SYSFUN function. The keyword DOUBLE_PRECISION has not been defined for the latter.

*DOUBLE function examples*

```
WITH temp1(c1,d1) AS
(VALUES ('12345',12.4)
      , ('-23.5',1234)
      , ('1E+45',-234)
      , ('-2e05',+2.4)
)
SELECT DOUBLE(c1) AS c1d
     , DOUBLE(d1) AS d1d
FROM temp1;
```

*ANSWER (output shortened)*

| C1D | D1D |
|---|---|
| +1.23450000E+004 | +1.24000000E+001 |
| -2.35000000E+001 | +1.23400000E+003 |
| +1.00000000E+045 | -2.34000000E+002 |
| -2.00000000E+005 | +2.40000000E+000 |

See Floating Point Numbers for a discussion on floating-point number manipulation.

### 7.2.66. EMPTY_BLOB, EMPTY_CLOB, EMPTY_DBCLOB, and EMPTY_NCLOB

These functions return a zero-length value with a data type of BLOB, CLOB, or DBCLOB.

#### ENCRYPT

Returns a encrypted rendition of the input string. The input must be char or varchar. The output is varchar for bit data.

The input values are defined as follows:

- **ENCRYPTED DATA**: A char or varchar string 32633 bytes that is to be encrypted. Numeric data must be converted to character before encryption.
- **PASSWORD**: A char or varchar string of at least six bytes and no more than 127 bytes. If the value is null or not provided, the current value of the encryption password special register will be used. Be aware that a password that is padded with blanks is not the same as one that lacks the blanks.
- **HINT**: A char or varchar string of up to 32 bytes that can be referred to if one forgets what the

password is. It is included with the encrypted string and can be retrieved using the GETHINT function.

The length of the output string can be calculated thus:

- When the hint is provided, the length of the input data, plus eight bytes, plus the distance to the next eight-byte boundary, plus thirty-two bytes for the hint.
- When the hint is not provided, the length of the input data, plus eight bytes, plus the distance to the next eight-byte boundary.

*ENCRYPT function example*

```
SELECT id
     , name
     , ENCRYPT(name,'THAT IDIOT','MY BROTHER') AS name2
FROM staff
WHERE ID < 30
ORDER BY id;
```

### 7.2.67. EVENT_MON_STATE

Returns an operational state of a particular event monitor.

### 7.2.68. EXP

Returns the exponential function of the argument. The output format is double.

*EXP function examples*

```
WITH temp1(n1) AS
(VALUES (0)
   UNION ALL
 SELECT n1 + 1
 FROM temp1
 WHERE n1 < 10
)
SELECT n1
     , EXP(n1)           AS e1
     , SMALLINT(EXP(n1)) AS e2
FROM temp1;
```

*ANSWER*

| N1 | E1 | E2 |
|---|---|---|
| 0 | +1.00000000000000E+0 | 1 |
| 1 | +2.71828182845904E+0 | 2 |
| 2 | +7.38905609893065E+0 | 7 |

| N1 | E1 | E2 |
|---|---|---|
| 3 | +2.00855369231876E+1 | 20 |
| 4 | +5.45981500331442E+1 | 54 |
| 5 | +1.48413159102576E+2 | 148 |
| 6 | +4.03428793492735E+2 | 403 |
| 7 | +1.09663315842845E+3 | 1096 |
| 8 | +2.98095798704172E+3 | 2980 |
| 9 | +8.10308392757538E+3 | 8103 |
| 10 | +2.20264657948067E+4 | 22026 |

### 7.2.69. EXTRACT

Returns a portion of a datetime based on its arguments.

### 7.2.70. FIRST_DAY

Returns a date or timestamp that represents the first day of the month of the argument.

### 7.2.71. FLOAT

Same as DOUBLE or DOUBLE_PRECISION.

### 7.2.72. FLOOR

Returns the next largest integer value that is smaller than or equal to the input (e.g. 5.945 returns 5.000). The output field type will equal the input field type.

*FLOOR function examples*

```
SELECT d1
     , FLOOR(d1) AS d2
     , f1
     , FLOOR(f1) AS f2
FROM scalar;
```

*ANSWER (float output shortened)*

| D1 | D2 | F1 | F2 |
|---|---|---|---|
| 2.4 | -3. | -2.400E+0 | -3.000E+0 |
| 0.0 | +0. | +0.000E+0 | +0.000E+0 |
| 1.8 | +1. | +1.800E+0 | +1.000E+0 |

### 7.2.73. FROM_UTC_TIMESTAMP

Returns a TIMESTAMP that is converted from Coordinated Universal Time to the time zone specified by the time zone string. FROM_UTC_TIMESTAMP is a statement deterministic function.

### 7.2.74. GENERATE_UNIQUE

Uses the system clock and node number to generate a value that is guaranteed unique (as long as one does not reset the clock). The output is of type CHAR(13) FOR BIT DATA. There are no arguments. The result is essentially a timestamp (set to universal time, not local time), with the node number appended to the back.

*GENERATE_UNIQUE function examples. Note that the second field is unprintable*

```
SELECT id
     , GENERATE_UNIQUE() AS unique_val#1
     , DEC(HEX(GENERATE_UNIQUE()),26) AS unique_val#2
FROM staff
WHERE id < 50
ORDER BY id;
```

*ANSWER*

| ID | UNIQUE_VAL#1 | UNIQUE_VAL#2 |
|---|---|---|
| 10 | | 20011017191648990521000000. |
| 20 | | 20011017191648990615000000. |
| 30 | | 20011017191648990642000000. |
| 40 | | 20011017191648990669000000. |

Observe that in the above example, each row gets a higher value. This is to be expected, and is in contrast to a CURRENT TIMESTAMP call, where every row returned by the cursor will have the same timestamp value. Also notice that the second invocation of the function on the same row got a lower value (than the first). In the prior query, the HEX and DEC functions were used to convert the output value into a number. Alternatively, the TIMESTAMP function can be used to convert the date component of the data into a valid timestamp. In a system with multiple nodes, there is no guarantee that this timestamp (alone) is unique.

### 7.2.75. Generate Unique Timestamps

The GENERATE_UNIQUE output can be processed using the TIMESTAMP function to obtain a unique timestamp value. Adding the CURRENT TIMEZONE special register to the TIMESTAMP output will convert it to local time:

*Covert GENERATE_UNIQUE output to timestamp*

```
SELECT CURRENT TIMESTAMP                          AS ts1
, TIMESTAMP(GENERATE_UNIQUE())                    AS ts2
, TIMESTAMP(GENERATE_UNIQUE()) + CURRENT TIMEZONE AS ts3
FROM sysibm.sysdummy1;
```

*ANSWER*

| TS1 | TS2 | TS3 |
|---|---|---|
| 2007-01-19-18.12.33.587000 | 2007-01-19-22.12.28.434960 | 2007-01-19-18.12.28.434953 |

This code can be useful if one is doing a multi-row insert, and one wants each row inserted to have a distinct timestamp value. However, there are a few qualifications:

- The timestamp values generated will be unique in themselves. But concurrent users may also generate the same values. There is no guarantee of absolute uniqueness.
- Converting the universal-time value to local-time does not always return a value is equal to the CURRENT TIMESTAMP special register. As is illustrated above, the result can differ by a few seconds. This may cause business problems if one is relying on the value to be the "true time" when something happened.

### Making Random

One thing that Db2 lacks is a random number generator that makes unique values. However, if we flip the characters returned in the GENERATE_UNIQUE output, we have something fairly close to what is needed. Unfortunately, Db2 also lacks a REVERSE function, so the data flipping has to be done the hard way.

*GENERATE_UNIQUE output, characters reversed to make*

```
SELECT u1
, SUBSTR(u1,20,1) CONCAT SUBSTR(u1,19,1) CONCAT
  SUBSTR(u1,18,1) CONCAT SUBSTR(u1,17,1) CONCAT
  SUBSTR(u1,16,1) CONCAT SUBSTR(u1,15,1) CONCAT
  SUBSTR(u1,14,1) CONCAT SUBSTR(u1,13,1) CONCAT
  SUBSTR(u1,12,1) CONCAT SUBSTR(u1,11,1) CONCAT
  SUBSTR(u1,10,1) CONCAT SUBSTR(u1,09,1) CONCAT
  SUBSTR(u1,08,1) CONCAT SUBSTR(u1,07,1) CONCAT
  SUBSTR(u1,06,1) CONCAT SUBSTR(u1,05,1) CONCAT
  SUBSTR(u1,04,1) CONCAT SUBSTR(u1,03,1) CONCAT
  SUBSTR(u1,02,1) CONCAT SUBSTR(u1,01,1) AS U2
FROM (SELECT HEX(GENERATE_UNIQUE()) AS u1
      FROM staff
      WHERE id < 50) AS xxx
ORDER BY u2;
```

*ANSWER*

| U1 | U2 |
|---|---|
| 20000901131649119940000000 | 04991194613110900002 |
| 20000901131649119793000000 | 39791194613110900002 |
| 20000901131649119907000000 | 70991194613110900002 |
| 20000901131649119969000000 | 96991194613110900002 |

#### Pseudo-random

Observe above that we used a nested table expression to temporarily store the results of the GENERATE_UNIQUE calls. Alternatively, we could have put a GENERATE_UNIQUE call inside each SUBSTR, but these would have amounted to separate function calls, and there is a very small chance that the net result would not always be unique.

#### Using REVERSE Function

One can refer to a user-defined reverse function (see Reversing Field Contents for the definition code) to flip the U1 value, and thus greatly simplify the query:

*GENERATE_UNIQUE output, characters reversed using function*

```
SELECT u1
     , SUBSTR(reverse(CHAR(u1)),7,20) AS u2
FROM
  (SELECT HEX(GENERATE_UNIQUE()) AS u1
   FROM staff
   WHERE ID < 50) AS xxx
ORDER BY U2;
```

### 7.2.76. GETHINT

Returns the password hint, if one is found in the encrypted data.

-GETHINT function example

```
SELECT id
     , name
     , GETHINT(name2) AS hint
FROM
  (SELECT id
        , name
        , ENCRYPT(name,'THAT IDIOT','MY BROTHER') AS name2
   FROM staff
   WHERE id < 30
  ) AS xxx
ORDER BY id;
```

*ANSWER*

| ID | NAME | HINT |
|---|---|---|
| 10 | Sanders | MY BROTHER |
| 20 | Pernal | MY BROTHER |

### 7.2.77. GRAPHIC

Converts the input (1st argument) to a graphic data type. The output length (2nd argument) is optional.

### 7.2.78. GREATEST

See MAX scalar function.

### 7.2.79. HASH

Returns a 128-bit, 160-bit, 256-bit or 512-bit hash of the input data, depending on the algorithm selected, and is intended for cryptographic purposes.

### 7.2.80. HASH4

Returns the 32-bit checksum hash of the input data. The function provides 232 distinct return values and is intended for data retrieval (lookups).

### 7.2.81. HASH8

Returns the 64-bit hash of an input string. The function provides 264 distinct return values and is intended for data retrieval (that is, lookups). The result for a particular input string differs depending on the endianness (big-endian or little-endian) of your system.

### 7.2.82. HASHEDVALUE

Returns the partition number of the row. The result is zero if the table is not partitioned. The output is of type integer, and is never null.

*HASHEDVALUE function example*

```
SELECT HASHEDVALUE(id) AS hvalue
FROM staff
WHERE id = 10;
```

*ANSWER*

| HVALUE |
|---|
| 0 |

The DBPARTITIONNUM function will generate a SQL error if the column/row used can not be related directly back to specific row in a real table. Therefore, one can not use this function on

fields in GROUP BY statements, nor in some views. It can also cause an error when used in an outer join, and the target row failed to match in the join.

### 7.2.83. HEX

Returns the hexadecimal representation of a value. All input types are supported.

*HEX function examples, numeric data*

```
WITH temp1(n1) AS
(VALUES (-3)
   UNION ALL
 SELECT n1 + 1
 FROM temp1
 WHERE n1 < 3)
SELECT SMALLINT(n1)      AS s
     , HEX(SMALLINT(n1)) AS shx
     , HEX(DEC(n1,4,0))  AS dhx
     , HEX(DOUBLE(n1))   AS fhx
FROM temp1;
```

*ANSWER*

| S | SHX | DHX | FHX |
|---|---|---|---|
| 3 | FDFF | 00003D | 00000000000008C0 |
| -2 | FEFF | 00002D | 00000000000000C0 |
| -1 | FFFF | 00001D | 000000000000F0BF |
| 0 | 0000 | 00000C | 0000000000000000 |
| 1 | 0100 | 00001C | 000000000000F03F |
| 2 | 0200 | 00002C | 0000000000000040 |
| 3 | 0300 | 00003C | 0000000000000840 |

*HEX function examples, character & varchar*

```
SELECT c1
     , HEX(c1) AS chx
     , v1
     , HEX(v1) AS vhx
FROM scalar;
```

*ANSWER*

| C1 | CHX | V1 | VHX |
|---|---|---|---|
| ABCDEF | 414243444546 | ABCDEF | 414243444546 |
| ABCD | 414243442020 | ABCD | 41424344 |

| C1 | CHX | V1 | VHX |
|---|---|---|---|
| AB | 414220202020 | AB | 4142 |

*HEX function examples, date & time*

```
SELECT dt1
     , HEX(dt1) AS dthx
     , tm1
     , HEX(tm1) AS tmhx
FROM scalar;
```

*ANSWER*

| DT1 | DTHX | TM1 | TMHX |
|---|---|---|---|
| 1996-04-22 | 19960422 | 23:58:58 | 235858 |
| 1996-08-15 | 19960815 | 15:15:15 | 151515 |
| 0001-01-01 | 00010101 | 00:00:00 | 000000 |

### 7.2.84. HEXTORAW

Returns a bit string representation of a hexadecimal character string.

### 7.2.85. HOUR

Returns the hour (as in hour of day) part of a time value. The output format is integer.

*HOUR function example*

```
SELECT tm1
     , HOUR(tm1) AS hr
FROM scalar
ORDER BY tm1;
```

*ANSWER*

| TM1 | HR |
|---|---|
| 00:00:00 | 0 |
| 15:15:15 | 15 |
| 23:58:58 | 23 |

### 7.2.86. HOURS_BETWEEN

Returns the number of full hours between the specified arguments.

### 7.2.87. IDENTITY_VAL_LOCAL

Returns the most recently assigned value (by the current user) to an identity column. The result type is decimal (31,0), regardless of the field type of the identity column. See Find Gaps in Values for detailed notes on using this function.

*IDENTITY_VAL_LOCAL function usage*

```
CREATE TABLE seq#
( ident_val INTEGER   NOT NULL GENERATED ALWAYS AS IDENTITY
, cur_ts    TIMESTAMP NOT NULL
, PRIMARY KEY (ident_val));
COMMIT;
INSERT INTO seq# VALUES(DEFAULT,CURRENT TIMESTAMP);

WITH temp (idval) AS
(VALUES (IDENTITY_VAL_LOCAL())
)
SELECT *
FROM temp;
```

*ANSWER*

| IDVAL |
| --- |
| 1. |

### 7.2.88. INITCAP

Returns a string with the first character of each word converted to uppercase, using the UPPER function semantics, and the other characters converted to lowercase, using the LOWER function semantics.

### 7.2.89. INSERT

Insert one string in the middle of another, replacing a portion of what was already there. If the value to be inserted is either longer or shorter than the piece being replaced, the remainder of the data (on the right) is shifted either left or right accordingly in order to make a good fit.

**Usage Notes**

*Acceptable input types are varchar, clob(1M), and blob(1M). The first and last parameters must always have matching field types. To insert a new value in the middle of another without removing any of what is already there, set the third parameter to zero. The varchar output is always of length 4K.

*INSERT function examples*

```
SELECT name
    , INSERT(name,3,2,'A')
    , INSERT(name,3,2,'AB')
    , INSERT(name,3,2,'ABC')
FROM staff
WHERE id < 40;
```

*ANSWER (4K output fields shortened)*

| NAME | 2 | 3 | 4 |
|---|---|---|---|
| Sanders | SaAers | SaABers | SaABCers |
| Pernal | PeAal | PeABal | PeABCal |
| Marenghi | MaAnghi | MaABnghi | MaABCnghi |

### 7.2.90. INSTR

Returns the starting position of a string (the search string) within another string (the source string). The INSTR scalar function is a synonym for the LOCATE_IN_STRING scalar function.

### 7.2.91. INSTR2

Returns the starting position, in 16-bit UTF-16 string units (CODEUNITS16), of a string within another string.

### 7.2.92. INSTR4

Returns the starting position, in 32-bit UTF-32 string units (CODEUNITS32), of a string within another string.

### 7.2.93. INSTRB

Returns the starting position, in bytes, of a string within another string.

### 7.2.94. INT or INTEGER

The INTEGER or INT function converts either a number or a valid character value into an integer. The character input can have leading and/or trailing blanks, and a sign indictor, but it can not contain a decimal point. Numeric decimal input works just fine.

*INTEGER function examples*

```
SELECT d1
     , INTEGER(d1)
     , INT('+123')
     , INT('-123')
     , INT(' 123 ')
FROM scalar;
```

*ANSWER*

| D1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| 2.4 | -2 | 123 | -123 | 123 |
| 0.0 | 0 | 123 | -123 | 123 |
| 1.8 | 1 | 123 | -123 | 123 |

### 7.2.95. INTNAND , INTNOR, INTNXOR, and INTNNOT

These bitwise functions operate on the "two's complement" representation of the integer value of the input arguments and return the result as a corresponding base 10 integer value.

### 7.2.96. JULIAN_DAY

Converts a date value into a number that represents the number of days since January the 1st, 4,713 BC. The output format is integer.

*JULIAN_DAY function example*

```
WITH temp1(dt1) AS
(VALUES ('0001-01-01-00.00.00')
      , ('1752-09-10-00.00.00')
      , ('2007-06-03-00.00.00')
      , ('2007-06-03-23.59.59')
)
SELECT DATE(dt1)       AS dt
     , DAYS(dt1)       AS dy
     , JULIAN_DAY(dt1) AS dj
FROM temp1;
```

*ANSWER*

| DT | DY | DJ |
|---|---|---|
| 0001-01-01 | 1 | 1721426 |
| 1752-09-10 | 639793 | 2361218 |
| 2007-06-03 | 732830 | 2454255 |

| DT | DY | DJ |
|---|---|---|
| 2007-06-03 | 732830 | 2454255 |

Julian Days, A History

I happen to be a bit of an Astronomy nut, so what follows is a rather extended description of Julian Days - their purpose, and history (taken from the web). The Julian Day calendar is used in Astronomy to relate ancient and modern astronomical observations. The Babylonians, Egyptians, Greeks (in Alexandria), and others, kept very detailed records of astronomical events, but they all used different calendars.
By converting all such observations to Julian Days, we can compare and correlate them. For example, a solar eclipse is said to have been seen at Ninevah on Julian day 1,442,454 and a lunar eclipse is said to have been observed at Babylon on Julian day number 1,566,839. These numbers correspond to the Julian Calendar dates -763-03-23 and -423-10-09 respectively). Thus the lunar eclipse occurred 124,384 days after the solar eclipse. The Julian Day number system was invented by Joseph Justus Scaliger (born 1540-08-05 J in Agen, France, died 1609-01-21 J in Leiden, Holland) in 1583. Although the term Julian Calendar derives from the name of Julius Caesar, the term Julian day number probably does not.
Evidently, this system was named, not after Julius Caesar, but after its inventor's father, Julius Caesar Scaliger (1484-1558). The younger Scaliger combined three traditionally recognized temporal cycles of 28, 19 and 15 years to obtain a great cycle, the Scaliger cycle, or Julian period, of 7980 years (7980 is the least common multiple of 28, 19 and 15). The length of 7,980 years was chosen as the product of 28 times 19 times 15; these, respectively, are:

- The number of years when dates recur on the same days of the week.
- The lunar or Metonic cycle, after which the phases of the Moon recur on a particular day in the solar year, or year of the seasons.
- The cycle of indiction, originally a schedule of periodic taxes or government requisitions in ancient Rome.

The first Scaliger cycle began with Year 1 on -4712-01-01 (Julian) and will end after 7980 years on 3267-12-31 (Julian), which is 3268-01-22 (Gregorian). 3268-01-01 (Julian) is the first day of Year 1 of the next Scaliger cycle. Astronomers adopted this system and adapted it to their own purposes, and they took noon GMT -4712-01-01 as their zero point. For astronomers a day begins at noon and runs until the next noon (so that the nighttime falls conveniently within one "day"). Thus they defined the Julian day number of a day as the number of days (or part of a day) elapsed since noon GMT on January 1st, 4713 B.C.E. This was not to the liking of all scholars using the Julian day number system, in particular, historians. For chronologists who start "days" at midnight, the zero point for the Julian day number system is 00:00 at the start of -4712-01-01 J, and this is day 0. This means that 200001-01 G is 2,451,545 JD. Since most days within about 150 years of the present have Julian day numbers beginning with "24", Julian day numbers within this 300-odd-year period can be abbreviated. In 1975 the convention of the modified Julian day number was adopted: Given a Julian day number JD, the modified Julian day number MJD is defined as MJD = JD - 2,400,000.5. This has two purposes:

- Days begin at midnight rather than noon.
- For dates in the period from 1859 to about 2130 only five digits need to be used to specify the

date rather than seven.

MJD 0 thus corresponds to JD 2,400,000.5, which is twelve hours after noon on JD 2,400,000 = 1858-11-16. Thus MJD 0 designates the midnight of November 16th/17th, 1858, so day 0 in the system of modified Julian day numbers is the day 1858-11-17. The following SQL statement uses the JULIAN_DAY function to get the Julian Date for certain days. The same calculation is also done using hand-coded SQL.

*JULIAN_DAY function examples*

```
SELECT bd
     , JULIAN_DAY(bd)
     , (1461 * (YEAR(bd) + 4800 +
       (MONTH(bd)-14)/12))/4    +
       ( 367 * (MONTH(bd)- 2    -
       12*((MONTH(bd)-14)/12)))/12  -
       (3 * ((YEAR(bd) + 4900 +
       (MONTH(bd)-14)/12)/100))/4 +
       DAY(bd) - 32075
FROM
(SELECT birthdate AS bd
 FROM employee
WHERE midinit = 'R'
) AS xxx
ORDER BY bd;
```

*ANSWER*

| BD | 2 | 3 |
|---|---|---|
| 1926-05-17 | 2424653 | 2424653 |
| 1936-03-28 | 2428256 | 2428256 |
| 1946-07-09 | 2432011 | 2432011 |
| 1955-04-12 | 2435210 | 2435210 |

### Julian Dates

Many computer users think of the "Julian Date" as a date format that has a layout of "yynnn" or "yyyynnn" where "yy" is the year and "nnn" is the number of days since the start of the same. A more correct use of the term "Julian Date" refers to the current date according to the calendar as originally defined by Julius Caesar - which has a leap year on every fourth year. In the US/UK, this calendar was in effect until "1752-09-14". The days between the 3rd and 13th of September in 1752 were not used in order to put everything back in sync. In the 20th and 21st centuries, to derive the Julian date one must subtract 13 days from the relevant Gregorian date (e.g.1994-01-22 becomes 1994-01-07). The following SQL illustrates how to convert a standard Db2 Gregorian Date to an equivalent Julian Date (calendar) and a Julian Date (output format):

*Julian Date outputs*

```
WITH temp1(dt1) AS
(VALUES ('2007-01-01')
      , ('2007-01-02')
      , ('2007-12-31')
)
SELECT DATE(dt1)                          AS dt
     , DATE(dt1) - 13 DAYS                AS dj1
     , YEAR(dt1) * 1000 + DAYOFYEAR(dt1) AS dj2
FROM temp1;
```

*ANSWER*

| DT | DJ1 | DJ2 |
|---|---|---|
| 2007-01-01 | 2006-12-19 | 2007001 |
| 2007-01-02 | 2006-12-20 | 2007002 |
| 2007-12-31 | 2007-12-18 | 2007365 |

Db2 does not make allowances for the days that were not used when Englishspeaking countries converted from the Julian to the Gregorian calendar in 1752.

### 7.2.97. LAST_DAY

Feturns a date or timestamp value that represents the last day of the month of the argument.

### 7.2.98. LCASE or LOWER

Converts a mixed or upper-case string to lower case. The output is the same data type and length as the input.

*LCASE function example*

```
SELECT name
     , LCASE(name) AS lname
     , UCASE(name) AS uname
FROM staff
WHERE id < 30;
```

*ANSWER*

| NAME | LNAME | UNAME |
|---|---|---|
| Sanders | sanders | SANDERS |
| Pernal | pernal | PERNAL |

### 7.2.99. LEAST

See MIN scalar function.

### 7.2.100. LEFT

The LEFT function has two arguments: The first is an input string of type char, varchar, clob, or blob. The second is a positive integer value. The output is the left most characters in the string. Trailing blanks are not removed.

*LEFT function examples*

```
WITH temp1(c1) AS
(VALUES (' ABC')
      , (' ABC ')
      , ('ABC ')
)
SELECT c1
     , LEFT(c1,4)         AS c2
     , LENGTH(LEFT(c1,4)) AS l2
FROM temp1;
```

*ANSWER*

| C1 | C2 | L2 |
|---|---|---|
| ABC | AB | 4 |
| ABC | ABC | 4 |
| ABC | ABC | 4 |

If the input is either char or varchar, the output is varchar(4000). A column this long is a nuisance to work with. Where possible, use the SUBSTR function to get around this problem.

### 7.2.101. LENGTH

Returns an integer value with the internal length of the expression (except for double-byte string types, which return the length in characters). The value will be the same for all fields in a column, except for columns containing varying-length strings.

*LENGTH function examples*

```
SELECT LENGTH(d1)
     , LENGTH(f1)
     , LENGTH(s1)
     , LENGTH(c1)
     , LENGTH(RTRIM(c1))
FROM scalar;
```

*ANSWER*

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| 2 | 8 | 2 | 6 | 6 |
| 2 | 8 | 2 | 6 | 4 |
| 2 | 8 | 2 | 6 | 2 |

### 7.2.102. LENGTH2

Returns the length of expression in 16-bit UTF-16 string units (CODEUNITS16).

### 7.2.103. LENGTH4

Returns the length of expression in 32-bit UTF-32 string units (CODEUNITS32).

### 7.2.104. LENGTHB

Returns the length of expression in bytes.

### 7.2.105. LN or LOG

Returns the natural logarithm of the argument (same as LOG). The output format is double.

*LOG function example*

```
WITH temp1(n1) AS
(VALUES (1)
      , (123)
      , (1234)
      , (12345)
      , (123456)
)
SELECT n1
     , LOG(n1) AS l1
FROM temp1;
```

*ANSWER*

| N1 | L1 |
|---|---|
| 1 | +0.00000000000000E+000 |
| 123 | +4.81218435537241E+000 |
| 1234 | +7.11801620446533E+000 |
| 12345 | +9.42100640177928E+000 |
| 123456 | +1.17236400962654E+001 |

### 7.2.106. LOCATE

Returns an integer value with the absolute starting position of the first occurrence of the first string within the second string. If there is no match, the result is zero. The optional third parameter indicates where to start the search.

The result, if there is a match, is always the absolute position (i.e. from the start of the string), not the relative position (i.e. from the starting position).

*LOCATE function examples*

```
WITH temp1 (c1) As
(VALUES ('abcdÄ')
      , ('Äbcd')
      , ('AÄ')
      , ('ÁÄ')
)
SELECT c1
     , LOCATE('Ä',c1)               AS "l1"
     , LOCATE('Ä',c1,2)             AS "l2"
     , LOCATE('Ä',c1,OCTETS)        AS "l3"
     , LOCATE('Ä',c1,CODEUNITS16)   AS "l4"
     , LOCATE('Ä',c1,2,CODEUNITS16) AS "l5"
FROM temp1;
```

*ANSWER*

| C1 | l1 | l2 | l3 | l4 | l5 |
|---|---|---|---|---|---|
| abcdÄ | 5 | 5 | 5 | 5 | 5 |
| Äbcd | 1 | 0 | 1 | 1 | 0 |
| AÄ | 2 | 2 | 2 | 2 | 2 |
| ÁÄ | 3 | 3 | 3 | 2 | 2 |

When a special character like "Á" is encountered before the find-string (see last line) the plain LOCATE returns the number of bytes searched, not the number of characters.

### 7.2.107. LOCATE_IN_STRING

Returns the starting position of a string (called the search-string ) within another string (called the source-string).

### 7.2.108. LOG10

Returns the common logarithm (base 10) of a number.

*LOG10 function example*

```
WITH temp1(n1) AS
(VALUES (1)
      , (123)
      , (1234)
      , (12345)
      , (123456)
)
SELECT n1
     , LOG10(n1) AS l1
FROM temp1;
```

*ANSWER*

| N1 | L1 |
|---|---|
| 1 | +0.00000000000000E+000 |
| 123 | +2.08990511143939E+000 |
| 1234 | +3.09131515969722E+000 |
| 12345 | +4.09149109426795E+000 |
| 123456 | +5.09151220162777E+000 |

### 7.2.109. LONG_VARCHAR

Converts the input (1st argument) to a long_varchar data type. The output length (2nd argument) is optional.

### 7.2.110. LONG_VARGRAPHIC

Converts the input (1st argument) to a long_vargraphic data type. The output length (2nd argument) is optional.

### 7.2.111. LOWER

See the description for the LCASE function.

### 7.2.112. LPAD

Pads a string on the left with a specified character string or with blanks.

### 7.2.113. LTRIM

Remove leading blanks, but not trailing blanks, from the argument.

*LTRIM function example*

```
WITH temp1(c1) AS
(VALUES (' ABC')
      , (' ABC ')
      , ('ABC ')
)
SELECT c1
     , LTRIM(c1)         AS c2
     , LENGTH(LTRIM(c1)) AS l2
FROM temp1;
```

*ANSWER*

| C1 | C2 | L2 |
|---|---|---|
| ABC | ABC | 3 |
| ABC | ABC | 4 |
| ABC | ABC | 5 |

## 7.2.114. MAX

Returns the largest item from a list that must be at least two items long:

*MAX scalar function*

```
VALUES MAX(5, 8, 4)
```

*ANSWER* ⇒ 8

One can combine the MAX scalar and column functions to get the combined MAX value of a set of rows and columns:

*Sample Views used in Join Examples*

```
SELECT MAX(MAX(salary, years, comm))
FROM staff;
```

*ANSWER* ⇒ 87654.50

Db2 knows which function is which because the MAX scalar value must have at least two input values, while the column function can only have one.

### Null Processing

The MAX and MIN scalar functions return null if any one of the input list items is null. The MAX and MIN column functions ignore null values. They do however return null when no rows match.

### 7.2.115. MAX_CARDINALITY

Returns a BIGINT value that is the maximum number of values that an array can contain.

### 7.2.116. MICROSECOND

Returns the microsecond part of a timestamp (or equivalent) value. The output is integer.

*MICROSECOND function example*

```
SELECT ts1
     , MICROSECOND(ts1)
FROM scalar
ORDER BY ts1;
```

*ANSWER*

| TS1 | 2 |
|---|---|
| 0001-01-01-00.00.00.000000 | 0 |
| 1996-04-22-23.58.58.123456 | 123456 |
| 1996-08-15-15.15.15.151515 | 151515 |

### 7.2.117. MIDNIGHT_SECONDS

Returns the number of seconds since midnight from a timestamp, time or equivalent value. The output format is integer.

*MIDNIGHT_SECONDS function example*

```
SELECT ts1
     , MIDNIGHT_SECONDS(ts1)
     , HOUR(ts1)*3600 + MINUTE(ts1)*60 + SECOND(ts1)
FROM scalar
ORDER BY ts1;
```

*ANSWER*

| TS1 | 2 | 3 |
|---|---|---|
| 0001-01-01-00.00.00.000000 | 0 | 0 |
| 1996-04-22-23.58.58.123456 | 86338 | 86338 |
| 1996-08-15-15.15.15.151515 | 54915 | 54915 |

There is no single function that will convert the MIDNIGHT_SECONDS output back into a valid time value. However, it can be done using the following SQL:

*Convert MIDNIGHT_SECONDS output back to a time value*

```
WITH temp1 (ms) AS
(SELECT MIDNIGHT_SECONDS(ts1)
 FROM scalar
)
SELECT ms
     , SUBSTR(DIGITS(ms/3600), 9)
     || ':' ||
     SUBSTR(DIGITS((ms-((MS/3600) * 3600))/60 ), 9)
     || ':' ||
     SUBSTR(DIGITS(ms-((MS/60)*60)), 9) AS tm
FROM temp1
ORDER BY 1;
```

*ANSWER*

| MS | TM |
|---|---|
| 0 | 00:00:00 |
| 54915 | 15:15:15 |
| 86338 | 23:58:58 |

The following two identical timestamp values: "2005-07-15.24.00.00" and "2005-07-16.00.00.00" will return different MIDNIGHT_SECONDS results. See the chapter titled Quirks in SQL for a detailed discussion of this issue.

### 7.2.118. MIN

Returns the smallest item from a list that must be at least two items long:

*MIN scalar function*

```
VALUES MIN(5, 8, 4)
```

*ANSWER* ⇒ 4

Null is returned if any one of the list items is null.

### 7.2.119. MINUTE

Returns the minute part of a time or timestamp (or equivalent) value. The output is integer.

*MINUTE function example*

```
SELECT ts1
     , MINUTE(ts1)
FROM scalar
ORDER BY ts1;
```

*ANSWER*

| TS1 | 2 |
|---|---|
| 0001-01-01-00.00.00.000000 | 0 |
| 1996-04-22-23.58.58.123456 | 58 |
| 1996-08-15-15.15.15.151515 | 15 |

### 7.2.120. MINUTES_BETWEEN

Returns the number of full minutes between the specified arguments.

### 7.2.121. MOD

Returns the remainder (modulus) for the first argument divided by the second. In the following example the last column uses the MOD function to get the modulus, while the second to last column obtains the same result using simple arithmetic.

*MOD function example*

```
WITH temp1(n1,n2) AS
(VALUES (-31,+11)
  UNION ALL
SELECT n1 + 13
     , n2 - 4
FROM temp1
WHERE n1 < 60
)
SELECT n1
     , n2
     , n1/n2           AS div
     , n1-((n1/n2)*n2) AS md1
     , MOD(n1,n2)      AS md2
FROM temp1
ORDER BY 1;
```

*ANSWER*

| N1 | N2 | DIV | MD1 | MD2 |
|---|---|---|---|---|
| 31 | 11 | -2 | -9 | -9 |

| N1 | N2 | DIV | MD1 | MD2 |
|---|---|---|---|---|
| -18 | 7 | -2 | -4 | -4 |
| -5 | 3 | -1 | -2 | -2 |
| 8 | -1 | -8 | 0 | 0 |
| 21 | -5 | -4 | 1 | 1 |
| 34 | -9 | -3 | 7 | 7 |
| 47 | -13 | -3 | 8 | 8 |
| 60 | -17 | -3 | 9 | 9 |

## 7.2.122. MONTH

Returns an integer value in the range 1 to 12 that represents the month part of a date or timestamp (or equivalent) value.

**MONTHNAME**

Returns the name of the month (e.g. October) as contained in a date (or equivalent) value. The output format is varchar(100).

*MONTH and MONTHNAME functions example*

```
SELECT dt1
     , MONTH(dt1)
     , MONTHNAME(dt1)
FROM scalar
ORDER BY dt1;
```

*ANSWER*

| DT1 | 2 | 3 |
|---|---|---|
| 0001-01-01 | 1 | January |
| 1996-04-22 | 4 | April |
| 1996-08-15 | 8 | August |

## 7.2.123. MONTHS_BETWEEN

Returns an estimate of the number of months between expression1 and expression2.

## 7.2.124. MULTIPLY_ALT

Returns the product of two arguments as a decimal value. Use this function instead of the multiplication operator when you need to avoid an overflow error because Db2 is putting aside too much space for the scale (i.e. fractional part of number) Valid input is any exact numeric type: decimal, integer, bigint, or smallint (but not float).

*Multiplying numbers - examples*

```
WITH temp1 (n1,n2) AS
(VALUES (DECIMAL(1234,10)
       , DECIMAL(1234,10))
)
SELECT n1
     , n2
     , n1 * n2             AS p1
     , "*"(n1,n2)          AS p2
     , MULTIPLY_ALT(n1,n2) AS p3
FROM temp1;
```

*ANSWER*

| N1 | N2 | P1 | P2 | P3 |
|---|---|---|---|---|
| 1234. | 1234. | 1522756. | 1522756. | 1522756. |

When doing ordinary multiplication of decimal values, the output precision and the scale is the sum of the two input precisions and scales - with both having an upper limit of 31. Thus, multiplying a DEC(10,5) number and a DEC(4,2) number returns a DEC(14,7) number. Db2 always tries to avoid losing (truncating) fractional digits, so multiplying a DEC(20,15) number with a DEC(20,13) number returns a DEC(31,28) number, which is probably going to be too small. The MULTIPLY_ALT function addresses the multiplication overflow problem by, if need be, truncating the output scale. If it is used to multiply a DEC(20,15) number and a DEC(20,13) number, the result is a DEC(31,19) number. The scale has been reduced to accommodate the required precision. Be aware that when there is a need for a scale in the output, and it is more than three digits, the function will leave at least three digits. Below are some examples of the output precisions and scales generated by this function:

*Table 8. Decimal multiplication - same output lengths*

| INPUT#1 | INPUT#2 | RESULT "*" OPERATOR | RESULT MULTIPLY_ALT | M_A→SCALE TRUNCATD | M_A→PRECISION TRUNCATD |
|---|---|---|---|---|---|
| DEC(05,00) | DEC(05,00) | DEC(10,00) | DEC(10,00) | NO | NO |
| DEC(10,05) | DEC(11,03) | DEC(21,08) | DEC(21,08) | NO | NO |
| DEC(20,15) | DEC(21,13) | DEC(31,28) | DEC(31,18) | YES | NO |
| DEC(26,23) | DEC(10,01) | DEC(31,24) | DEC(31,19) | YES | NO |
| DEC(31,03) | DEC(15,08) | DEC(31,11) | DEC(31,03) | YES | YES |

## 7.2.125. NCHAR

Returns a fixed-length national character string representation of a variety of data types.

### 7.2.126. NCLOB

Returns a NCLOB representation of any type of national character string.

### 7.2.127. NVARCHAR

Returns a varying-length national character string representation of a variety of data types.

### 7.2.128. NEXT_DAY

Returns a datetime value that represents the first weekday, named by string-expression, that is later than the date in expression

### 7.2.129. NEXT_MONTH

Returns the first day of the next month after the specified date.

### 7.2.130. NEXT_QUARTER

Returns the first day of the next quarter after the date specified by the input.

### 7.2.131. NEXT_WEEK

Returns the first day of the next week after the specified date. Sunday is considered the first day of that new week.

### 7.2.132. NEXT_YEAR

Returns the first day of the year follows the year containing the date specified by the input.

### 7.2.133. NORMALIZE_DECFLOAT

Removes any trailing zeros from a DECFLOAT value.

*NORMALIZE_DECFLOAT function examples*

```
WITH temp1 (d1) AS
(VALUES (DECFLOAT(1))
      , (DECFLOAT(1.0))
      , (DECFLOAT(1.00))
      , (DECFLOAT(1.000))
      , (DECFLOAT('12.3'))
      , (DECFLOAT('12.30'))
      , (DECFLOAT(1.2e4))
      , (DECFLOAT('1.2e4'))
      , (DECFLOAT(1.2e-3))
      , (DECFLOAT('1.2e-3'))
)
SELECT d1
     , NORMALIZE_DECFLOAT(d1) AS d2
FROM temp1;
```

*ANSWER*

| D1 | D2 |
|---|---|
| 1 | 1 |
| 1.0 | 1 |
| 1.00 | 1 |
| 1.000 | 1 |
| 12.3 | 12.3 |
| 12.30 | 12.3 |
| 12000 | 1.2E+4 |
| 1.2E+4 | 1.2E+4 |
| 0.001200000000000000 | 0.0012 |
| 0.0012 | 0.0012 |

### 7.2.134. NOW

Returns a timestamp based on a reading of the time-of-day clock when the SQL statement is executed at the current server.

### 7.2.135. NULLIF

Returns null if the two values being compared are equal, otherwise returns the first value.

*NULLIF function examples*

```
SELECT s1
     , NULLIF(s1, 0)
     , c1
     , NULLIF(c1, 'AB')
FROM scalar
WHERE NULLIF(0, 0) IS NULL;
```

*ANSWER*

| S1 | 2 | C1 | 4 |
|---|---|---|---|
| 2 | -2 | ABCDEF | ABCDEF |
| 0 | - | ABCD | ABCD |
| 1 | 1 | AB | - |

### 7.2.136. NVL

Same as COALESCE.

### 7.2.137. NVL2

Returns the second argument when the first argument is not NULL. If the first argument is NULL, the third argument is returned.

### 7.2.138. OCTET_LENGTH

Returns the length of the input expression in octets (bytes).

*OCTET_LENGTH example*

```
WITH temp1 (c1) AS
(VALUES (CAST('ÁÉÌ' AS VARCHAR(10)))
)
SELECT c1                          AS C1
     , LENGTH(c1)                  AS LEN
     , OCTET_LENGTH(c1)            AS OCT
     , CHAR_LENGTH(c1,OCTETS)      AS L08
     , CHAR_LENGTH(c1,CODEUNITS16) AS L16
     , CHAR_LENGTH(c1,CODEUNITS32) AS L32
FROM temp1;
```

*ANSWER*

| C1 | LEN | OCT | L08 | L16 | L32 |
|---|---|---|---|---|---|
| ÁÉÌ | 6 | 6 | 6 | 3 | 3 |

### 7.2.139. OVERLAY

Overlay (i.e. replace) some part of a string with another string. There are five parameters:

- The source string to be edited.
- The new string to be inserted. This value can be zero length, but must be provided.
- Start position for new string, and also to where start deleting. This value must be between one and the string length.
- Number of bytes in the source to be overlaid. This value is optional.
- The code unit to use.

There are two function notations. One uses keywords to separate each parameter. The other uses commas.

*OVERLAY function example*

```
WITH temp1 (txt) AS
(VALUES('abcded')
     , ('addd')
     , ('adq')
)
SELECT txt
     , OVERLAY(txt,'XX',3,1,OCTETS) AS "s3f1"
     , OVERLAY(txt,'XX',2, OCTETS)  AS "s2f0"
     , OVERLAY(txt,'XX',1,1,OCTETS) AS "s1f1"
     , OVERLAY(txt,'XX',2,2,OCTETS) AS "s2f2"
FROM temp1;
```

*ANSWER*

| TXT | s3f1 | s2f0 | s1f1 | s2f2 |
|---|---|---|---|---|
| abcded | abXXded | aXXcded | XXbcded | aXXded |
| addd | adXXd | aXXdd | XXddd | aXXd |
| adq | adXX | aXXq | XXdq | aXX |

### 7.2.140. PARAMETER

The PARAMETER function represents a position in an SQL statement where the value is provided dynamically by XQuery as part of the invocation of the db2-fn:sqlquery function.

### 7.2.141. POSITION

Returns an integer value with the absolute starting position of the first occurrence of the first string within the second string. If there is no match, the result is zero. The third parameter indicates what code-unit to use.

When a special character like "Á" is encountered before the find-string (see last two lines in next

example) the plain OCTETS search returns the number of bytes searched, not the number of characters:

*POSITION function syntax*

```
WITH temp1 (c1) As
(VALUES ('Ä')
      , ('aÄ')
      , ('ÁÄ')
      , ('ÁÁÄ')
)
SELECT c1
     , POSITION('Ä',c1,OCTETS)          AS "p1"
     , POSITION('Ä',c1,CODEUNITS16)     AS "p2"
     , POSITION('Ä',c1,CODEUNITS32)     AS "p3"
     , POSITION('Ä' IN c1 USING OCTETS) AS "p4"
FROM temp1;
```

*ANSWER*

| C1 | p1 | p2 | p3 | p4 |
|---|---|---|---|---|
| Ä | 1 | 1 | 1 | 1 |
| aÄ | 2 | 2 | 2 | 2 |
| ÁÄ | 3 | 2 | 2 | 3 |
| ÁÁÄ | 5 | 3 | 3 | 5 |

The LOCATE function (see LOCATE) is very similar to the POSITION function. It has the additional capability of being able to start the search at any position in the search string.

### 7.2.142. POSSTR

Returns the position at which the second string is contained in the first string. If there is no match the value is zero. The test is case sensitive. The output format is integer.

POSSTR function example

```
SELECT c1
     , POSSTR(c1,' ')  AS p1
     , POSSTR(c1,'CD') AS p2
     , POSSTR(c1,'cd') AS p3
FROM scalar
ORDER BY 1;
```

*ANSWER*

| C1 | P1 | P2 | P3 |
|---|---|---|---|
| AB | 3 | 0 | 0 |
| ABCD | 5 | 3 | 0 |
| ABCDEF | 0 | 3 | 0 |

**POSSTR vs. LOCATE**

The LOCATE and POSSTR functions are very similar. Both look for matching strings searching from the left. The only functional differences are that the input parameters are reversed and the LOCATE function enables one to begin the search at somewhere other than the start. When either is suitable for the task at hand, it is probably better to use the POSSTR function because it is a SYSIBM function and so should be faster.

*POSSTR vs. LOCATE functions*

```
SELECT c1
     , POSSTR(c1, ' ')    AS p1
     , LOCATE(' ', c1)    AS l1
     , POSSTR(c1, 'CD')   AS p2
     , LOCATE('CD', c1)   AS l2
     , POSSTR(c1, 'cd')   AS p3
     , LOCATE('cd', c1)   AS l3
     , LOCATE('D', c1, 2) AS l4
FROM scalar
ORDER BY 1;
```

*ANSWER*

| C1 | P1 | L1 | P2 | L2 | P3 | L3 | L4 |
|---|---|---|---|---|---|---|---|
| AB | 3 | 3 | 0 | 0 | 0 | 0 | 0 |
| ABCD | 5 | 5 | 3 | 3 | 0 | 0 | 4 |
| ABCDEF | 0 | 0 | 3 | 3 | 0 | 0 | 4 |

## 7.2.143. POWER or POW

Returns the value of the first argument to the power of the second argument

*POWER function examples*

```
WITH temp1(n1) AS
(VALUES (1)
      , (10)
      , (100)
)
SELECT n1
     , POWER(n1, 1) AS p1
     , POWER(n1, 2) AS p2
     , POWER(n1, 3) AS p3
FROM temp1;
```

*ANSWER*

| N1 | P1 | P2 | P3 |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
| 10 | 10 | 100 | 1000 |
| 100 | 100 | 10000 | 1000000 |

### 7.2.144. QUANTIZE

Convert the first input parameter to DECFLOAT, using the second parameter as a mask. The specific value of the second parameter is irrelevant. But the precision (i.e. number of digits after the decimal point) defines the precision of the DECFLOAT result:

*QUANTIZE function examples*

```
WITH temp1 (d1, d2) AS
(VALUES (+1.23,   DECFLOAT(1.0))
      , (+1.23,   DECFLOAT(1.00))
      , (-1.23,   DECFLOAT(1.000))
      , (+123,    DECFLOAT(9.8765))
      , (+123,    DECFLOAT(1E-3))
      , (+123,    DECFLOAT(1E+3))
      , (SQRT(2), DECFLOAT(0.0))
      , (SQRT(2), DECFLOAT('1E-5'))
      , (SQRT(2), DECFLOAT( 1E-5 ))
)
SELECT QUANTIZE(d1,d2) AS q
FROM temp1;
```

*ANSWER*

| Q |
|---|
| 1.2 |
| 1.23 |

| Q |
| --- |
| -1.230 |
| 123.0000 |
| 123.000 |
| 123 |
| 1.4 |
| 1.41421 |
| 1.414213562373095100000 |

Observe that the function returns a very different result when the second parameter is '1E-5' vs. 1E-5 (i.e. with no quotes). This is because the number 1E-5 is not precisely converted to the DECFLOAT value 0.00001, as the following query illustrates:

-DECFLOAT conversion example

```
WITH temp1 (d1) AS
(VALUES (DECFLOAT('1E-5'))
      , (DECFLOAT( 1E-5 ))
)
SELECT d1
FROM temp1;
```

*ANSWER*

| D1 |
| --- |
| 0.00001 |
| 0.000010000000000000001 |

### 7.2.145. QUARTER

Returns an integer value in the range 1 to 4 that represents the quarter of the year from a date or timestamp (or equivalent) value.

### 7.2.146. RADIANS

Returns the number of radians converted from the input, which is expressed in degrees. The output format is double.

### 7.2.147. RAISE_ERROR

Causes the SQL statement to stop and return a user-defined error message when invoked. There are a lot of usage restrictions involving this function, see the SQL Reference for details.

*RAISE_ERROR function example*

```
SELECT s1
     , CASE
         WHEN s1 < 1 THEN s1
         ELSE RAISE_ERROR('80001', c1)
       END AS s2
FROM scalar;
```

*ANSWER*

| S1 | S2 |
|---|---|
| -2 | -2 |
| 0 | 0 |
| **SQLSTATE=80001** | |

The SIGNAL statement (see SIGNAL Statement) is the statement equivalent of this function.

## 7.2.148. RAND

WARNING: Using the RAND function in a predicate can result in unpredictable results. See RAND in Predicate for a detailed description of this issue. To randomly sample the rows in a table reliably and efficiently, use the TABLESAMPLE feature. See Randomly Sample Data for details.

Returns a pseudo-random floating-point value in the range of zero to one inclusive. An optional seed value can be provided to get reproducible random results. This function is especially useful when one is trying to create somewhat realistic sample data.

**Usage Notes**

- The RAND function returns any one of 32K distinct floating-point values in the range of zero to one inclusive. Note that many equivalent functions in other languages (e.g. SAS) return many more distinct values over the same range.
- The values generated by the RAND function are evenly distributed over the range of zero to one inclusive.
- A seed can be provided to get reproducible results. The seed can be any valid number of type integer. Note that the use of a seed alone does not give consistent results. Two different SQL statements using the same seed may return different (but internally consistent) sets of pseudo-random numbers.
- If the seed value is zero, the initial result will also be zero. All other seed values return initial values that are not the same as the seed. Subsequent calls of the RAND function in the same statement are not affected.
- If there are multiple references to the RAND function in the same SQL statement, the seed of the first RAND invocation is the one used for all.
- If the seed value is not provided, the pseudo-random numbers generated will usually be unpredictable. However, if some prior SQL statement in the same thread has already invoked

the RAND function, the newly generated pseudo-random numbers "may" continue where the prior ones left off.

### Typical Output Values

The following recursive SQL generates 100,000 random numbers using two as the seed value. The generated data is then summarized using various Db2 column functions:

*Sample output from RAND function*

```
WITH temp (num, ran) AS
(VALUES (INT(1)
       , RAND(2))
  UNION ALL
SELECT num + 1
     , RAND()
FROM temp
WHERE num < 100000
)
SELECT COUNT(*)                                  AS #rows
     , COUNT(DISTINCT ran)                       AS #values
     , DEC(AVG(ran), 7, 6)                       AS avg_ran
     , DEC(STDDEV(ran), 7, 6)                    AS std_devm
     , DEC(MIN(ran), 7, 6)                       AS min_ran
     , DEC(MAX(ran), 7, 6)                       AS max_ran
     , DEC(MAX(ran), 7, 6) - DEC(MIN(ran), 7, 6) AS range
     , DEC(VAR(ran), 7, 6)                       AS variance
FROM temp;
```

*ANSWER*

| #ROWS | #VALUES | AVG_RAN | STD_DEVM | MIN_RAN | MAX_RAN | RANGE | VARIANCE |
|---|---|---|---|---|---|---|---|
| 100000 | 31242 | 0.499838 | 0.288706 | 0.000000 | 1.000000 | 1.000000 | 0.083351 |

Observe that less than 32K distinct numbers were generated. Presumably, this is because the RAND function uses a 2-byte carry. Also observe that the values range from a minimum of zero to a maximum of one.

Unlike most, if not all, other numeric functions in Db2, the RAND function returns different results in different flavors of Db2.

### Reproducible Random Numbers

The RAND function creates pseudo-random numbers. This means that the output looks random, but it is actually made using a very specific formula. If the first invocation of the function uses a seed value, all subsequent invocations will return a result that is explicitly derived from the initial seed. To illustrate this concept, the following statement selects six random numbers. Because of the use of the seed, the same six values will always be returned when this SQL statement is invoked (when

invoked on my machine):

*Make reproducible random numbers (use seed)*

```
SELECT deptno  AS dno
     , RAND(0) AS ran
FROM department
WHERE deptno < 'E'
ORDER BY 1;
```

*ANSWER*

| DNO | RAN |
|---|---|
| A00 | +1.15970336008789E-003 |
| B01 | +2.35572374645222E-001 |
| C01 | +6.48152104251228E-001 |
| D01 | +7.43736075930052E-002 |
| D11 | +2.70241401409955E-001 |
| D21 | +3.60026856288339E-001 |

To get random numbers that are not reproducible, simply leave the seed out of the first invocation of the RAND function. To illustrate, the following statement will give differing results with each invocation:

*Make non-reproducible random numbers (no seed)*

```
SELECT deptno AS dno
     , RAND() AS ran
FROM department
WHERE deptno < 'D'
ORDER BY 1;
```

*ANSWER*

| DNO | RAN |
|---|---|
| A00 | +2.55287331766717E-001 |
| B01 | +9.85290078432569E-001 |
| C01 | +3.18918424024171E-001 |

Use of the seed value in the RAND function has an impact across multiple SQL statements. For example, if the above two statements were always run as a pair (with nothing else run in between), the result from the second would always be the same.

### Generating Random Values

Imagine that we need to generate a set of reproducible random numbers that are within a certain range (e.g. 5 to 15). Recursive SQL can be used to make the rows, and various scalar functions can be used to get the right range of data. In the following example we shall make a list of three columns and ten rows. The first field is a simple ascending sequence. The second is a set of random numbers of type smallint in the range zero to 350 (by increments of ten). The last is a set of random decimal numbers in the range of zero to 10,000.

*Use RAND to make sample data*

```
WITH Temp1 (col1, col2, col3) AS
(VALUES (0
       , SMALLINT(RAND(2)*35)*10
       , DECIMAL(RAND()*10000, 7, 2)
       )
  UNION ALL
SELECT col1 + 1
     , SMALLINT(RAND()*35)*10
     , DECIMAL(RAND()*10000, 7, 2)
FROM temp1
WHERE col1 + 1 < 10
)
SELECT *
FROM temp1;
```

*ANSWER*

| COL1 | COL2 | COL3 |
|---|---|---|
| 0 | 0 | 9342.32 |
| 1 | 250 | 8916.28 |
| 2 | 310 | 5430.76 |
| 3 | 150 | 5996.88 |
| 4 | 110 | 8066.34 |
| 5 | 50 | 5589.77 |
| 6 | 130 | 8602.86 |
| 7 | 340 | 184.94 |
| 8 | 310 | 5441.14 |
| 9 | 70 | 9267.55 |

See the section titled "Making Sample Data" for more detailed examples of using the RAND function and recursion to make test data.

### Making Many Distinct Random Values

The RAND function generates 32K distinct random values. To get a larger set of (evenly distributed) random values, combine the result of two RAND calls in the manner shown below for the RAN2 column:

*Use RAND to make many distinct random values*

```
WITH temp1 (col1,ran1,ran2) AS
(VALUES (0,
         RAND(2),
         RAND()+(RAND()/1E5)
        )
  UNION ALL
SELECT col1 + 1
     , RAND()
     , RAND() +(RAND()/1E5)
FROM temp1
WHERE col1 + 1 < 30000
)
SELECT COUNT(*)              AS col#1
     , COUNT(DISTINCT ran1) AS ran#1
     , COUNT(DISTINCT ran2) AS ran#2
FROM temp1;
```

*ANSWER*

| COL#1 | RAN#1 | RAN#2 |
|---|---|---|
| 30000 | 19698 | 29998 |

Observe that we do not multiply the two values that make up the RAN2 column above. If we did this, it would skew the average (from 0.5 to 0.25), and we would always get a zero whenever either one of the two RAND functions returned a zero.

The GENERATE_UNIQUE function can also be used to get a list of distinct values, and actually does a better job that the RAND function. With a bit of simple data manipulation (see GENERATE_UNIQUE), these values can also be made random.

### Selecting Random Rows, Percentage

Using the RAND function in a predicate can result in unpredictable results. See RAND in Predicate for a detailed description of this issue.

Imagine that you want to select approximately 10% of the matching rows from some table. The predicate in the following query will do the job:

*Randomly select 10% of matching rows*

```
SELECT id
     , name
FROM staff
WHERE RAND() < 0.1
ORDER BY id;
```

*ANSWER*

| ID | NAME |
|---|---|
| 140 | Fraye |
| 190 | Sneider |
| 290 | Quill |

The RAND function randomly generates values in the range of zero through one, so the above query should return approximately 10% the matching rows. But it may return anywhere from zero to all of the matching rows - depending on the specific values that the RAND function generates. If the number of rows to be processed is large, then the fraction (of rows) that you get will be pretty close to what you asked for. But for small sets of matching rows, the result set size is quite often anything but what you wanted.

#### Selecting Random Rows, Number

The following query will select five random rows from the set of matching rows. It begins (in the inner-most nested table expression) by using the RAND function to assign random values to each matching row. Subsequently, the ROW_NUMBER function is used to sequence each random value. Finally, those rows with the five lowest row numbers are selected:

*Select five random rows*

```
SELECT id
     , name
FROM
  (SELECT s2.*
        , ROW_NUMBER() OVER(ORDER BY r1) AS r2
   FROM
       (SELECT s1.*
             , RAND() AS r1
        FROM staff s1
        WHERE id <= 100) AS s2
  ) as s3
WHERE r2 <= 5
ORDER BY id;
```

*ANSWER*

| ID | NAME |
|---|---|
| 10 | Sanders |
| 30 | Marenghi |
| 40 | O'Brien |
| 70 | Rothman |
| 100 | Plotz |

**Use in DML**

Imagine that in act of inspired unfairness, we decided to update a selected set of employee's salary to a random number in the range of zero to $10,000. This too is easy:

*Use RAND to assign random salaries*

```
UPDATE staff
SET salary = RAND()*10000
WHERE id < 50;
```

## 7.2.149. RAWTOHEX

Returns a hexadecimal representation of a value as a character string.

## 7.2.150. REAL

Returns a single-precision floating-point representation of a number.

*REAL and other numeric function examples*

```
SELECT n1          AS dec
     , DOUBLE(n1)  AS dbl
     , REAL(n1)    AS rel
     , INTEGER(n1) AS int
     , BIGINT(n1)  AS big
FROM
  (SELECT 1234567890.12345678901234567890 AS n1
   FROM staff
   WHERE id = 10) AS xxx;
```

*ANSWER*

| DEC | DBL | REL | INT | BIG |
|---|---|---|---|---|
| 1234567890.12345678901234567890 | 1.23456789012346e+009 | 1.234568e+009 | 1234567890 | 1234567890 |

### 7.2.151. REC2XML

Returns a string formatted with XML tags, containing column names and column data.

### 7.2.152. REGEXP_COUNT

Returns a count of the number of times that a regular expression pattern is matched in a string.

### 7.2.153. REGEXP_EXTRACT

Returns one occurrence of a substring of a string that matches the regular expression pattern.

### 7.2.154. REGEXP_INSTR

Returns the starting or ending position of the matched substring, depending on the value of the return_option argument.

### 7.2.155. REGEXP_LIKE

Returns a boolean value indicating if the regular expression pattern is found in a string. The function can be used only where a predicate is supported.

### 7.2.156. REGEXP_MATCH_COUNT

Returns a count of the number of times that a regular expression pattern is matched in a string.

### 7.2.157. REGEXP_REPLACE

Returns a modified version of the source string where occurrences of the regular expression pattern found in the source string are replaced with the specified replacement string.

### 7.2.158. REGEXP_SUBSTR

Returns one occurrence of a substring of a string that matches the regular expression pattern.

### 7.2.159. REPEAT

Repeats a character string "n" times.

*REPEAT function example*

```
SELECT id
     , CHAR(REPEAT(name, 3), 40)
FROM staff
WHERE id < 40
ORDER BY id;
```

*ANSWER*

| ID | 2 |
|---|---|
| 10 | SandersSandersSanders |
| 20 | PernalPernalPernal |
| 30 | MarenghiMarenghiMarenghi |

### 7.2.160. REPLACE

Replaces all occurrences of one string with another. The output is of type varchar(4000).

*REPLACE function examples*

```
SELECT c1
     , REPLACE(c1,'AB','XY') AS r1
     , REPLACE(c1,'BA','XY') AS r2
FROM scalar;
```

*ANSWER*

| C1 | R1 | R2 |
|---|---|---|
| ABCDEF | XYCDEF | ABCDEF |
| ABCD | XYCD | ABCD |
| AB | XY | AB |

The REPLACE function is case sensitive. To replace an input value, regardless of the case, one can nest the REPLACE function calls. Unfortunately, this technique gets to be a little tedious when the number of characters to replace is large.

*Nested REPLACE functions*

```
SELECT c1
, REPLACE(
    REPLACE(
      REPLACE(
        REPLACE(c1, 'AB', 'XY')
        , 'ab', 'XY')
      , 'Ab', 'XY')
    , 'aB', 'XY')
FROM scalar;
```

*ANSWER*

| C1 | R1 |
|---|---|
| ABCDEF | XYCDEF |
| ABCD | XYCD |

| C1 | R1 |
|---|---|
| AB | XY |

### 7.2.161. RID

Returns the RID (i.e. row identifier - of type BIGINT) for the matching row. The row identifier contains the page number, and the row number within the page. A unique table identifier must be provided.

*RID function example*

```
SELECT id
     , salary
     , RID(staff) AS staff_rid
FROM staff
WHERE id < 40
ORDER BY id;
```

*ANSWER*

| ID | SALARY | STAFF_RID |
|---|---|---|
| 10 | 98357.50 | 100663300 |
| 20 | 78171.25 | 100663301 |
| 30 | 77506.75 | 100663302 |

The RID function is similar to the RID_BIT function, but less useful (e.g. does not work in a DPF environment). All subsequent examples will refer to the RID_BIT function.

### 7.2.162. RID_BIT

Returns the row identifier, of type VARCHAR(16) FOR BIT DATA, for the row. The row identifier contains the page number, and the row number within the page. The only input value, which must be provided, is the (unique) table identifier. The table must be listed in the subsequent FROM statement.

*RID_BIT function example – single table*

```
SELECT id
     , RID_BIT(staff) AS rid_bit
FROM staff
WHERE id < 40
ORDER BY id;
```

*ANSWER*

| ID | RID_BIT |
|---|---|
| 10 | x'040000060000000000000000FCE14D' |
| 20 | x'050000060000000000000000FCE14D' |
| 30 | x'060000060000000000000000FCE14D' |

When the same table is referenced twice in the FROM, the correlation name must be used:

*RID_BIT function example – multiple tables*

```
SELECT s1.id
     , RID_BIT(s1) AS rid_bit
FROM staff s1
   , staff s2
WHERE s1.id < 40
AND   s1.id = s2.id - 10
ORDER BY s1.id;
```

*ANSWER*

| ID | RID_BIT |
|---|---|
| 10 | x'040000060000000000000000FCE14D' |
| 20 | x'050000060000000000000000FCE14D' |
| 30 | x'060000060000000000000000FCE14D' |

The RID function can be used in a predicate to uniquely identify a row. To illustrate, the following query gets the RID and ROW CHANGE TOKEN for a particular row:

*RID_BIT function example – select row to update*

```
SELECT id
     , salary
     , RID_BIT(staff)
     , ROW CHANGE TOKEN FOR staff
FROM staff
WHERE id = 20;
```

*ANSWER*

| ID | SALARY | 3 | 4 |
|---|---|---|---|
| 20 | 78171.25 | x'050000060000000000000000FCE14D' | 3999250443959009280 |

If at some subsequent point in time we want to update this row, we can use the RID value to locate it directly, and the ROW CHANGE TOKEN to confirm that it has not been changed:

*RID_BIT function example – update row*

```
UPDATE staff
SET salary = salary * 1.1
WHERE RID_BIT(staff) = x'0500000600000000000000000FCE14D'
AND ROW CHANGE TOKEN FOR staff = 3999250443959009280;
```

**Usage Notes**

- The table name provided to the RID_BIT function must uniquely identify the table being processed. If a view is referenced, the view must be deletable.
- The RID_BIT function will return a different value for a particular row a REORG is run.
- The ROW CHANGE TOKEN changes every time a row is updated, including when an update is rolled back. So after a rollback the value will be different from what it was at the beginning of the unit of work.
- The ROW CHANGE TOKEN is unique per page, not per row. So if any other row in the same page is changed, the prior update will not match. This is called a "false negative". To avoid, define a ROW CHANGE TIMESTAMP column for the table, as the value in this field is unique per row.

## 7.2.163. RIGHT

Has two arguments: The first is an input string of type char, varchar, clob, or blob. The second is a positive integer value. The output, of type varchar(4000), is the right most characters in the string.

*RIGHT function examples*

```
WITH temp1(c1) AS
(VALUES (' ABC')
      , (' ABC ')
      , ('ABC ')
)
SELECT c1
     , RIGHT(c1, 4)        AS c2
     , LENGTH(RIGHT(c1,4)) AS l2
FROM temp1;
```

*ANSWER*

| C1 | C2 | L2 |
|---|---|---|
| ABC | ABC | 4 |
| ABC | ABC | 4 |
| ABC | BC | 4 |

## 7.2.164. ROUND

Rounds the rightmost digits of number (1st argument). If the second argument is positive, it rounds

to the right of the decimal place. If the second argument is negative, it rounds to the left. A second argument of zero results rounds to integer. The input and output types are the same, except for decimal where the precision will be increased by one - if possible. Therefore, a DEC(5,2)field will be returned as DEC(6,2), and a DEC(31,2) field as DEC(31,2). To truncate instead of round, use the TRUNCATE function.

*ROUND function examples*

```
WITH temp1(d1) AS
(VALUES (123.400)
      , ( 23.450)
      , ( 3.456)
      , (   .056))
SELECT d1
     , DEC(ROUND(d1,+2),6,3) AS p2
     , DEC(ROUND(d1,+1),6,3) AS p1
     , DEC(ROUND(d1,+0),6,3) AS p0
     , DEC(ROUND(d1,-1),6,3) AS n1
     , DEC(ROUND(d1,-2),6,3) AS v
FROM temp1;
```

*ANSWER*

| D1 | P2 | P1 | P0 | N1 | N2 |
|---|---|---|---|---|---|
| 123.400 | 123.400 | 123.400 | 123.000 | 120.000 | 100.000 |
| 23.450 | 23.450 | 23.400 | 23.000 | 20.000 | 0.000 |
| 3.456 | 3.460 | 3.500 | 3.000 | 0.000 | 0.000 |
| 0.056 | 0.060 | 0.100 | 0.000 | 0.000 | 0.000 |

### 7.2.165. ROUND_TIMESTAMP

Returns a TIMESTAMP based on a provided argument (expression), rounded to the unit specified in another argument (format-string).

### 7.2.166. RPAD

Returns a string composed of string-expression padded on the right, with pad or blanks.

### 7.2.167. RTRIM

Trims the right-most blanks of a character string.

*RTRIM function example*

```
SELECT c1
     , RTRIM(c1)         AS r1
     , LENGTH(c1)        AS r2
     , LENGTH(RTRIM(c1)) AS r3
FROM scalar;
```

*ANSWER*

| C1 | R1 | R2 | R3 |
|---|---|---|---|
| ABCDEF | ABCDEF | 6 | 6 |
| ABCD | ABCD | 6 | 4 |
| AB | AB | 6 | 2 |

### 7.2.168. SECLABEL Functions

The SECLABEL, SECLABEL_BY_NAME, and SECLABEL_BY_CHAR functions are used to process security labels. See the SQL Reference for more details.

### 7.2.169. SECLABEL_BY_NAME

Returns the specified security label. The security label returned has a data type of DB2SECURITYLABEL. Use this function to insert a named security label.

### 7.2.170. SECLABEL_TO_CHAR

Accepts a security label and returns a string that contains all elements in the security label. The string is in the security label string format.

### 7.2.171. SECOND

Returns the second (of minute) part of a time or timestamp (or equivalent) value.

### 7.2.172. SECONDS_BETWEEN

Returns the number of full seconds between the specified arguments.

### 7.2.173. SIGN

Returns -1 if the input number is less than zero, 0 if it equals zero, and +1 if it is greater than zero. The input and output types will equal, except for decimal which returns double.

*SIGN function examples*

```
SELECT d1
     , SIGN(d1)
     , f1
     , SIGN(f1)
FROM scalar;
```

*ANSWER (float output shortened)*

| D1 | 2 | F1 | 4 |
|---|---|---|---|
| -2.4 | -1.000E+0 | -2.400E+0 | -1.000E+0 |
| 0.0 | +0.000E+0 | +0.000E+0 | +0.000E+0 |
| 1.8 | +1.000E+0 | +1.800E+0 | +1.000E+0 |

### 7.2.174. SIN

Returns the SIN of the argument where the argument is an angle expressed in radians. The output format is double.

*SIN function example*

```
WITH temp1(n1) AS
(VALUES (0)
   UNION ALL
 SELECT n1 + 10
 FROM temp1
 WHERE n1 < 80
)
SELECT n1
     , DEC(RADIANS(n1),4,3)      AS ran
     , DEC(SIN(RADIANS(n1)),4,3) AS sin
     , DEC(TAN(RADIANS(n1)),4,3) AS tan
FROM temp1;
```

*ANSWER*

| N1 | RAN | SIN | TAN |
|---|---|---|---|
| 0 | 0.000 | 0.000 | 0.000 |
| 10 | 0.174 | 0.173 | 0.176 |
| 20 | 0.349 | 0.342 | 0.363 |
| 30 | 0.523 | 0.500 | 0.577 |
| 40 | 0.698 | 0.642 | 0.839 |
| 50 | 0.872 | 0.766 | 1.191 |

| N1 | RAN | SIN | TAN |
|---|---|---|---|
| 60 | 1.047 | 0.866 | 1.732 |
| 70 | 1.221 | 0.939 | 2.747 |
| 80 | 1.396 | 0.984 | 5.671 |

### 7.2.175. SINH

Returns the hyperbolic sin for the argument, where the argument is an angle expressed in radians. The output format is double.

### 7.2.176. SMALLINT

Converts either a number or a valid character value into a smallint value.

*SMALLINT function examples*

```
SELECT d1
     , SMALLINT(d1)
     , SMALLINT('+123')
     , SMALLINT('-123')
     , SMALLINT(' 123 ')
FROM scalar;
```

*ANSWER*

| D1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| -2.4 | -2 | 123 | -123 | 123 |
| 0.0 | 0 | 123 | -123 | 123 |
| 1.8 | 1 | 123 | -123 | 123 |

### 7.2.177. SNAPSHOT Functions

The various SNAPSHOT functions can be used to analyze the system. They are beyond the scope of this book. Refer instead to the Db2 System Monitor Guide and Reference.

### 7.2.178. SOUNDEX

Returns a 4-character code representing the sound of the words in the argument. Use the DIFFERENCE function to convert words to soundex values and then compare.

*SOUNDEX function example*

```
SELECT a.name           AS n1
     , SOUNDEX(a.name) AS s1
     , b.name           AS n2
     , SOUNDEX(b.name) AS s2
     , DIFFERENCE(a.name,b.name) AS df
FROM staff a
   , staff b
WHERE a.id = 10
AND b.id > 150 AND b.id < 250
ORDER BY df DESC
       , n2 ASC;
```

*ANSWER*

| N1 | S1 | N2 | S2 | DF |
|---|---|---|---|---|
| Sanders | S536 | Sneider | S536 | 4 |
| Sanders | S536 | Smith | S530 | 3 |
| Sanders | S536 | Lundquist | L532 | 2 |
| Sanders | S536 | Daniels | D542 | 1 |
| Sanders | S536 | Molinare | M456 | 1 |
| Sanders | S536 | Scoutten | S350 | 1 |
| Sanders | S536 | Abrahams | A165 | 0 |
| Sanders | S536 | Kermisch | K652 | 0 |
| Sanders | S536 | Lu | L000 | 0 |

**SOUNDEX Formula**

There are several minor variations on the SOUNDEX algorithm. Below is one example:

- The first letter of the name is left unchanged.
- The letters W and H are ignored.
- The vowels, A, E, I, O, U, and Y are not coded, but are used as separators (see last item).
- The remaining letters are coded as:

| Letters | Value |
|---|---|
| B, P, F, V | 1 |
| C, G, J, K, Q, S, X, Z 2 | 2 |
| D, T | 3 |
| L | 4 |
| M, N | 5 |

| | |
|---|---|
| R | 6 |

- Letters that follow letters with same code are ignored unless a separator (see the third item above) precedes them.

The result of the above calculation is a four byte value. The first byte is a character as defined in step one. The remaining three bytes are digits as defined in steps two through four.

Output longer than four bytes is truncated If the output is not long enough, it is padded on the right with zeros. The maximum number of distinct values is 8,918.

The SOUNDEX function is something of an industry standard that was developed several decades ago. Since that time, several other similar functions have been developed. You may want to investigate writing your own Db2 function to search for similarsounding names.

### 7.2.179. SPACE

Returns a string consisting of "n" blanks. The output format is varchar(4000).

*SPACE function examples*

```
WITH temp1(n1) AS
(VALUES (1)
      , (2)
      , (3)
)
SELECT n1
     , SPACE(n1)         AS s1
     , LENGTH(SPACE(n1)) AS s2
     , SPACE(n1) || 'X'  AS s3
FROM temp1;
```

*ANSWER*

| N1 | S1 | S2 | S3 |
|---|---|---|---|
| 1 | | 1 | "X" |
| 2 | | 2 | " X" |
| 3 | | 3 | " X" |

### 7.2.180. SQRT

Returns the square root of the input value, which can be any positive number. The output format is double.

*SQRT function example*

```
WITH temp1(n1) AS
(VALUES (0.5)
      , (0.0)
      , (1.0)
      , (2.0)
)
SELECT DEC(n1,4,3)       AS n1
     , DEC(SQRT(n1),4,3) AS s1
FROM temp1;
```

*ANSWER*

| N1 | S1 |
|---|---|
| 0.500 | 0.707 |
| 0.000 | 0.000 |
| 1.000 | 1.000 |
| 2.000 | 1.414 |

### 7.2.181. STRIP

Removes leading, trailing, or both (the default), characters from a string. If no strip character is provided, leading and/or trailing blank characters are removed.

Observe in the following query that the last example removes leading "A" characters:

*STRIP function example*

```
WITH temp1(c1) AS
(VALUES (' ABC')
      , (' ABC ')
      , ('ABC ')
)
SELECT c1                       AS c1
     , STRIP(c1)                AS c2
     , LENGTH(STRIP(c1))        AS l2
     , STRIP(c1,LEADING)        AS c3
     , LENGTH(STRIP(c1,LEADING)) AS l3
     , STRIP(c1,LEADING,'A')    AS C4
FROM temp1;
```

*ANSWER*

| C1 | C2 | L2 | C3 | L3 | C4 |
|---|---|---|---|---|---|
| ABC | ABC | 3 | ABC | 3 | ABC |

| C1 | C2 | L2 | C3 | L3 | C4 |
|---|---|---|---|---|---|
| ABC | ABC | 3 | ABC | 4 | ABC |
| ABC | ABC | 3 | ABC | 5 | BC |

The TRIM function works the same way.

### 7.2.182. STRLEFT

Returns the leftmost string of string-expression of length length, expressed in the specified string unit.

### 7.2.183. STRPOS

Is a synonym for POSSTR.

### 7.2.184. STRRIGHT

Returns the right most string of string-expression of length length, expressed in the specified string unit.

### 7.2.185. SUBSTR or SUBSTRING

Returns part of a string. If the length is not provided, the output is from the start value to the end of the string.

If the length is provided, and it is longer than the field length, a SQL error results. The following statement illustrates this. Note that in this example the DAT1 field has a "field length" of 9 (i.e. the length of the longest input string).

*SUBSTR function - error because length parm too long*

```
WITH temp1 (len, dat1) AS
(VALUES ( 6,'123456789')
      , ( 4,'12345')
      , ( 16,'123')
)
SELECT len
     , dat1
     , LENGTH(dat1)         AS ldat
     , SUBSTR(dat1, 1, len) AS subdat
FROM temp1;
```

*ANSWER*

| LEN | DAT1 | LDAT | SUBDAT |
|---|---|---|---|
| 6 | 123456789 | 9 | 123456 |
| 4 | 12345 | 5 | 1234 |

The best way to avoid the above problem is to simply write good code. If that sounds too much like hard work, try the following SQL:

*SUBSTR function - avoid error using CASE (see previous)*

```
WITH temp1 (len, dat1) AS
(VALUES ( 6,'123456789')
      , ( 4,'12345')
      , ( 16,'123'
)
SELECT len
     , dat1
     , LENGTH(dat1) AS ldat
     , SUBSTR(dat1, 1,
         CASE
           WHEN len < LENGTH(dat1) THEN len
           ELSE LENGTH(dat1)
         END ) AS subdat
FROM temp1;
```

*ANSWER*

| LEN | DAT1 | LDAT | SUBDAT |
|---|---|---|---|
| 6 | 123456789 | 9 | 123456 |
| 4 | 12345 | 5 | 1234 |
| 16 | 123 | 3 | 123 |

In the above SQL a CASE statement is used to compare the LEN value against the length of the DAT1 field. If the former is larger, it is replaced by the length of the latter. If the input is varchar, and no length value is provided, the output is varchar. However, if the length is provided, the output is of type char - with padded blanks (if needed):

*SUBSTR function - fixed length output if third parm. used*

```
SELECT name
     , LENGTH(name) AS len
     , SUBSTR(name,5) AS s1
     , LENGTH(SUBSTR(name,5)) AS l1
     , SUBSTR(name,5,3) AS s2
     , LENGTH(SUBSTR(name,5,3)) AS l2
FROM staff
WHERE id < 60;
```

*ANSWER*

| NAME | LEN | S1 | L1 | S2 | L2 |
|---|---|---|---|---|---|
| Sanders | 7 | ers | 3 | ers | 3 |

| NAME | LEN | S1 | L1 | S2 | L2 |
|---|---|---|---|---|---|
| Pernal | 6 | al | 2 | al | 3 |
| Marenghi | 8 | nghi | 4 | ngh | 3 |
| O'Brien | 7 | ien | 3 | ien | 3 |
| Hanes | 5 | s | 1 | s | 3 |

### 7.2.186. SUBSTR2

Returns a substring from a string. The resulting substring starts at a specified position in the string and continues for a specified or default length. The start and length arguments are expressed in 16-bit UTF-16 string units (CODEUNITS16).

### 7.2.187. SUBSTR4

Returns a substring from a string. The resulting substring starts at a specified position in the string and continues for a specified or default length. The start and length arguments are expressed in 32-bit UTF-32 string units (CODEUNITS32).

### 7.2.188. SUBSTRB

Returns a substring of a string, beginning at a specified position in the string. Lengths are calculated in bytes.

### 7.2.189. TABLE

There isn't really a TABLE function, but there is a TABLE phrase that returns a result, one row at a time, from either an external (e.g. user written) function, or from a nested table expression. The TABLE phrase (function) has to be used in the latter case whenever there is a reference in the nested table expression to a row that exists outside of the expression. An example follows:

*Fullselect with external table reference*

```
SELECT a.id
     , a.dept
     , a.salary
     , b.deptsal
FROM staff a
, TABLE
    (SELECT b.dept
          , SUM(b.salary) AS deptsal
     FROM staff b
     WHERE b.dept = a.dept
     GROUP BY b.dept
    ) AS b
WHERE a.id < 40
ORDER BY a.id;
```

*ANSWER*

| ID | DEPT | SALARY | DEPTSAL |
|---|---|---|---|
| 10 | 20 | 98357.50 | 254286.10 |
| 20 | 20 | 78171.25 | 254286.10 |
| 30 | 38 | 77506.75 | 302285.55 |

See Table Function Usage for more details on using of the TABLE phrase in a nested table expression.

## 7.2.190. TABLE_NAME

Returns the base view or table name for a particular alias after all alias chains have been resolved. The output type is varchar(18). If the alias name is not found, the result is the input values. There are two input parameters. The first, which is required, is the alias name. The second, which is optional, is the alias schema. If the second parameter is not provided, the default schema is used for the qualifier.

TABLE_NAME function example

```
CREATE ALIAS emp1 FOR employee;
CREATE ALIAS emp2 FOR emp1;

SELECT tabschema
     , tabname
     , card
FROM syscat.tables
WHERE tabname = TABLE_NAME('emp2', 'graeme');
```

*ANSWER*

| TABSCHEMA | TABNAME | CARD |
|---|---|---|
| graeme | employee | -1 |

## 7.2.191. TABLE_SCHEMA

Returns the base view or table schema for a particular alias after all alias chains have been resolved. The output type is char(8). If the alias name is not found, the result is the input values. There are two input parameters. The first, which is required, is the alias name. The second, which is optional, is the alias schema. If the second parameter is not provided, the default schema is used for the qualifier.

**Resolving non-existent Objects**

Dependent aliases are not dropped when a base table or view is removed. After the base table or view drop, the TABLE_SCHEMA and TABLE_NAME functions continue to work fine (see the 1st output line below). However, when the alias being checked does not exist, the original input values

(explicit or implied) are returned (see the 2nd output line below).

*TABLE_SCHEMA and TABLE_NAME functions example*

```
CREATE VIEW fred1 (c1, c2, c3)
AS VALUES (11, 'AAA', 'BBB');
CREATE ALIAS fred2 FOR fred1;
CREATE ALIAS fred3 FOR fred2;

DROP VIEW fred1;

WITH temp1 (tab_sch, tab_nme) AS
(VALUES  (TABLE_SCHEMA('fred3', 'graeme')
       , TABLE_NAME('fred3'))
       , (TABLE_SCHEMA('xxxxx')
       , TABLE_NAME('xxxxx', 'xxx')))
SELECT *
FROM temp1;
```

*ANSWER*

| TAB_SCH | TAB_NME |
|---|---|
| graeme | fred1 |
| graeme | xxxxx |

### 7.2.192. TAN

Returns the tangent of the argument where the argument is an angle expressed in radians.

### 7.2.193. TANH

Returns the hyperbolic tan for the argument, where the argument is an angle expressed in radians. The output format is double.

### 7.2.194. THIS_MONTH

Returns the first day of the month in the specified date.

### 7.2.195. THIS_QUARTER

Returns the first day of the quarter that contains the specified date.

### 7.2.196. THIS_WEEK

Returns the first day of the week that contains the specified date. Sunday is considered the first day of that week.

### 7.2.197. THIS_YEAR

Returns the first day of the year in the specified date.

### 7.2.198. TIME

Converts the input into a time value.

### 7.2.199. TIMESTAMP

Converts the input(s) into a timestamp value.

**Argument Options**

If only one argument is provided, it must be (one of):

- A timestamp value.
- A character representation of a timestamp (the microseconds are optional).
- A 14 byte string in the form: YYYYMMDDHHMMSS.

If both arguments are provided:

- The first must be a date, or a character representation of a date.
- The second must be a time, or a character representation of a time.

*TIMESTAMP function examples*

```
SELECT TIMESTAMP('1997-01-11-22.44.55.000000')
     , TIMESTAMP('1997-01-11-22.44.55.000')
     , TIMESTAMP('1997-01-11-22.44.55')
     , TIMESTAMP('19970111224455')
     , TIMESTAMP('1997-01-11','22.44.55')
FROM staff WHERE id = 10;
```

### 7.2.200. TIMESTAMP_FORMAT

Takes an input string with the format: "YYYY-MM-DD HH:MM:SS" and converts it into a valid timestamp value. The VARCHAR_FORMAT function does the inverse.

*TIMESTAMP_FORMAT function example*

```
WITH temp1 (ts1) AS
(VALUES ('1999-12-31 23:59:59')
      , ('2002-10-30 11:22:33')
)
SELECT ts1
     , TIMESTAMP_FORMAT(ts1, 'YYYY-MM-DD HH24:MI:SS') AS ts2
FROM temp1
ORDER BY ts1;
```

*ANSWER*

| TS1 | TS2 |
|---|---|
| 1999-12-31 23:59:59 | 1999-12-31-23.59.59.000000 |
| 2002-10-30 11:22:33 | 2002-10-30-11.22.33.000000 |

Note that the only allowed formatting mask is the one shown.

## 7.2.201. TIMESTAMP_ISO

Returns a timestamp in the ISO format (yyyy-mm-dd hh:mm:ss.nnnnnn) converted from the IBM internal format (yyyy-mm-dd-hh.mm.ss.nnnnnn). If the input is a date, zeros are inserted in the time part. If the input is a time, the current date is inserted in the date part and zeros in the microsecond section.

*TIMESTAMP_ISO function example*

```
SELECT tm1
     , TIMESTAMP_ISO(tm1)
FROM scalar;
```

*ANSWER*

| TM1 | 2 |
|---|---|
| 23:58:58 | 2000-09-01-23.58.58.000000 |
| 15:15:15 | 2000-09-01-15.15.15.000000 |
| 00:00:00 | 2000-09-01-00.00.00.000000 |

## 7.2.202. TIMESTAMPDIFF

Returns an integer value that is an estimate of the difference between two timestamp values. Unfortunately, the estimate can sometimes be seriously out (see the example below), so this function should be used with extreme care.

**Arguments**

There are two arguments. The first argument indicates what interval kind is to be returned. Valid options are:

- **1** = Microseconds.
- **2** = Seconds.
- **4** = Minutes.
- **8** = Hours.
- **16** = Days.
- **32** = Weeks.
- **64** = Months.
- **128** = Quarters.
- **256** = Years.

The second argument is the result of one timestamp subtracted from another and then converted to character.

*TIMESTAMPDIFF function example*

```
WITH
temp1 (ts1,ts2) AS
(VALUES ('1996-03-01-00.00.01','1995-03-01-00.00.00')
      , ('1996-03-01-00.00.00','1995-03-01-00.00.01')
),
temp2 (ts1,ts2) AS
(SELECT TIMESTAMP(ts1)
      , TIMESTAMP(ts2)
 FROM temp1
),
temp3 (ts1,ts2,df) AS
(SELECT ts1
      , ts2
      , CHAR(TS1 - TS2) AS df
FROM temp2)
SELECT df
     , TIMESTAMPDIFF(16,df)  AS dif
     , DAYS(ts1) - DAYS(ts2) AS dys
FROM temp3;
```

*ANSWER*

| DF | DIF | DYS |
|---|---|---|
| 00010000000001.000000 | 365 | 366 |
| 00001130235959.000000 | 360 | 366 |

Some the interval types return estimates, not definitive differences, so should be used with care. For example, to get the difference between two timestamps in days, use the DAYS function as shown above. It is always correct.

**Roll Your Own**

The following user-defined function will get the difference, in microseconds, between two timestamp values. It can be used as an alternative to the above:

*Function to get difference between two timestamps*

```
CREATE FUNCTION ts_diff_works(in_hi TIMESTAMP, in_lo TIMESTAMP)
RETURNS BIGINT
RETURN (BIGINT(DAYS(in_hi))             * 86400000000
      + BIGINT(MIDNIGHT_SECONDS(in_hi)) * 1000000
      + BIGINT(MICROSECOND(in_hi)))
      -(BIGINT(DAYS(in_lo))             * 86400000000
      + BIGINT(MIDNIGHT_SECONDS(in_lo)) * 1000000
      + + BIGINT(MICROSECOND(in_lo)));
```

### 7.2.203. TO_CHAR

This function is a synonym for VARCHAR_FORMAT (see VARCHAR_FORMAT). It converts a timestamp value into a string using a template to define the output layout.

### 7.2.204. TO_CLOB

Returns a CLOB representation of a character string type.

### 7.2.205. TO_DATE

This function is a synonym for TIMESTAMP_FORMAT (see TIMESTAMP_FORMAT). It converts a character string value into a timestamp using a template to define the input layout.

### 7.2.206. TO_HEX

Converts a numeric expression into the hexadecimal representation.

### 7.2.207. TO_NCHAR

Returns a national character representation of an input expression that has been formatted using a character template.

### 7.2.208. TO_NCLOB

Returns any type of national character string.

### 7.2.209. TO_NUMBER

Returns a DECFLOAT(34) value that is based on the interpretation of the input string using the specified format.

### 7.2.210. TO_SINGLE_BYTE

Returns a string in which multi-byte characters are converted to the equivalent single-byte character where an equivalent character exists.

### 7.2.211. TO_TIMESTAMP

Returns a timestamp that is based on the interpretation of the input string using the specified format

### 7.2.212. TO_UTC_TIMESTAMP

Returns a TIMESTAMP that is converted to Coordinated Universal Time from the timezone that is specified by the timezone string. TO_UTC_TIMESTAMP is a a statement deterministic function.

### 7.2.213. TOTALORDER

Compares two DECFLOAT expressions and returns a SMALLINT number:

- **-1** if the first value is less than the second value.
- **0** if both values exactly equal (i.e. no trailing-zero differences)
- **+1** if the first value is greater than the second value.

Several values that compare as "less than" or "greater than" in the example below are equal in the usual sense. See the section on DECFLOAT arithmetic for details (see DECFLOAT Arithmetic).

,TOTALORDER function example

```
WITH temp1 (d1, d2) AS
(VALUES (DECFLOAT(+1.0), DECFLOAT(+1.0))
      , (DECFLOAT(+1.0), DECFLOAT(+1.00))
      , (DECFLOAT(-1.0), DECFLOAT(-1.00))
      , (DECFLOAT(+0.0), DECFLOAT(+0.00))
      , (DECFLOAT(-0.0), DECFLOAT(-0.00))
      , (DECFLOAT(1234), +infinity)
      , (+infinity, +infinity)
      , (+infinity, -infinity)
      , (DECFLOAT(1234), -NaN)
)
SELECT TOTALORDER(d1,d2)
FROM temp1;
```

*ANSWER*

| 1 |
|---|
| 0 |
| 1 |
| -1 |
| 1 |
| 1 |
| -1 |
| 0 |
| 1 |
| 1 |

### 7.2.214. TRANSLATE

Converts individual characters in either a character or graphic input string from one value to another. It can also convert lower case data to upper case.

**Usage Notes**

- The use of the input string alone generates upper case output.
- When "from" and "to" values are provided, each individual "from" character in the input string is replaced by the corresponding "to" character (if there is one).
- If there is no "to" character for a particular "from" character, those characters in the input string that match the "from" are set to blank (if there is no substitute value).
- A fourth, optional, single-character parameter can be provided that is the substitute character to be used for those "from" values having no "to" value.
- If there are more "to" characters than "from" characters, the additional "to" characters are ignored.

*TRANSLATE function examples*

```
SELECT 'abcd'
      , TRANSLATE('abcd')
      , TRANSLATE('abcd', '', 'a')
      , TRANSLATE('abcd', 'A', 'A')
      , TRANSLATE('abcd', 'A', 'a')
      , TRANSLATE('abcd', 'A', 'ab')
      , TRANSLATE('abcd', 'A', 'ab', ' ')
      , TRANSLATE('abcd', 'A', 'ab', 'z')
      , TRANSLATE('abcd', 'AB', 'a')
FROM staff WHERE id = 10;
```

| ANS. | NOTES |
|---|---|
| abcd | No change |

| ANS. | NOTES |
|---|---|
| ABCD | Make upper case |
| bcd | 'a'⇒' ' |
| abcd | 'A'⇒'A' |
| Abcd | 'a'⇒'A' |
| A cd | 'a'⇒'A','b'⇒' ' |
| A cd | 'a'⇒'A','b'⇒' ' |
| Azcd | 'a'⇒'A','b'⇒'z' |
| Abcd | 'a'⇒'A' |

**REPLACE vs. TRANSLATE - A Comparison**

Both the REPLACE and the TRANSLATE functions alter the contents of input strings. They differ in that the REPLACE converts whole strings while the TRANSLATE converts multiple sets of individual characters. Also, the "to" and "from" strings are back to front.

*REPLACE vs. TRANSLATE*

```
SELECT c1
     , REPLACE(c1, 'AB', 'XY')
     , REPLACE(c1, 'BA', 'XY')
     , TRANSLATE(c1, 'XY', 'AB')
     , TRANSLATE(c1, 'XY', 'BA')
FROM scalar
WHERE c1 = 'ABCD';
```

*ANSWER*

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| ABCD | XYCD | ABCD | XYCD | YXCD |

## 7.2.215. TRIM

See STRIP function.

## 7.2.216. TRIM_ARRAY

Deletes elements from the end of an array.

## 7.2.217. TRUNC_TIMESTAMP

Returns a TIMESTAMP that is an argument (expression) truncated to the unit specified by another argument (format-string).

### 7.2.218. TRUNC or TRUNCATE

Truncates (not rounds) the rightmost digits of an input number (1st argument). If the second argument is positive, it truncates to the right of the decimal place. If the second value is negative, it truncates to the left. A second value of zero truncates to integer. The input and output types will equal. To round instead of truncate, use the ROUND function.

*TRUNCATE function examples*

```
WITH temp1(d1) AS
(VALUES (123.400)
      , ( 23.450)
      , ( 3.456)
      , ( .056)
)
SELECT d1
     , DEC(TRUNC(d1,+2), 6, 3) AS pos2
     , DEC(TRUNC(d1,+1), 6, 3) AS pos1
     , DEC(TRUNC(d1,+0), 6, 3) AS zero
     , DEC(TRUNC(d1,-1), 6, 3) AS neg1
     , DEC(TRUNC(d1,-2), 6, 3) AS neg2
FROM temp1
ORDER BY 1 DESC;
```

*ANSWER*

| D1 | POS2 | POS1 | ZERO | NEG1 | NEG2 |
|---|---|---|---|---|---|
| 123.400 | 123.400 | 123.400 | 123.000 | 120.000 | 100.000 |
| 23.450 | 23.440 | 23.400 | 23.000 | 20.000 | 0.000 |
| 3.456 | 3.450 | 3.400 | 3.000 | 0.000 | 0.000 |
| 0.056 | 0.050 | 0.000 | 0.000 | 0.000 | 0.000 |

### 7.2.219. TYPE_ID

Returns the internal type identifier of the dynamic data type of the expression.

### 7.2.220. TYPE_NAME

Returns the unqualified name of the dynamic data type of the expression.

### 7.2.221. TYPE_SCHEMA

Returns the schema name of the dynamic data type of the expression.

### 7.2.222. UCASE or UPPER

Converts a mixed or lower-case string to upper case. The output is the same data type and length as

the input.

*UCASE function example*

```
SELECT name
     , LCASE(name) AS lname
     , UCASE(name) AS uname
FROM staff
WHERE id < 30;
```

*ANSWER*

| NAME | LNAME | UNAME |
|---|---|---|
| Sanders | sanders | SANDERS |
| Pernal | pernal | PERNAL |

### 7.2.223. VALUE

Same as COALESCE.

### 7.2.224. VARBINARY

Returns a VARBINARY (varying-length binary string) representation of a string of any data type.

### 7.2.225. VARCHAR

Converts the input (1st argument) to a varchar data type. The output length (2nd argument) is optional. Trailing blanks are not removed.

,VARCHAR function examples

```
SELECT c1
     , LENGTH(c1) AS l1
     , VARCHAR(c1) AS v2
     , LENGTH(VARCHAR(c1)) AS l2
     , VARCHAR(c1,4) AS v3
FROM scalar;
```

*ANSWER*

| C1 | L1 | V2 | L2 | V3 |
|---|---|---|---|---|
| ABCDEF | 6 | ABCDEF | 6 | ABCD |
| ABCD | 6 | ABCD | 6 | ABCD |
| AB | 6 | AB | 6 | AB |

### 7.2.226. VARCHAR_BIT_FORMAT

Returns a VARCHAR bit-data representation of a character string. See the SQL Reference for more details.

### 7.2.227. VARCHAR_FORMAT

Converts a timestamp value into a string with the format: "YYYY-MM-DD HH:MM:SS". The TIMESTAMP_FORMAT function does the inverse.

*VARCHAR_FORMAT function example*

```
WITH temp1 (ts1) AS
(VALUES (TIMESTAMP('1999-12-31-23.59.59'))
      , (TIMESTAMP('2002-10-30-11.22.33'))
)
SELECT ts1
     , VARCHAR_FORMAT(ts1,'YYYY-MM-DD HH24:MI:SS') AS ts2
FROM temp1
ORDER BY ts1;
```

*ANSWER*

| TS1 | TS2 |
|---|---|
| 1999-12-31-23.59.59.000000 | 1999-12-31 23:59:59 |
| 2002-10-30-11.22.33.000000 | 2002-10-30 11:22:33 |

Note that the only allowed formatting mask is the one shown.

### 7.2.228. VARCHAR_FORMAT_BIT

Returns a VARCHAR representation of a character bit-data string. See the SQL Reference for more details.

### 7.2.229. VARGRAPHIC

Converts the input (1st argument) to a VARGRAPHIC data type. The output length (2nd argument) is optional.

### 7.2.230. VERIFY_GROUP_FOR_USER

Returns a value that indicates whether the groups associated with the authorization ID identified by the SESSION_USER special register are in the group names specified by the list of group-name-expression arguments.

### 7.2.231. VERIFY_ROLE_FOR_USER

Returns a value that indicates whether any of the roles associated with the authorization ID

identified by the SESSION_USER special register are in (or contain any of) the role names specified by the list of role-name-expression arguments.

### 7.2.232. VERIFY_TRUSTED_CONTEXT_ROLE_FOR_USER

Returns a value that indicates whether the authorization ID identified by the SESSION_USER special register has acquired a role under a trusted connection associated with some trusted context and that role is in (or part of) the role names specified by the list of role-name-expression arguments.

### 7.2.233. WEEK

Returns a value in the range 1 to 53 or 54 that represents the week of the year, where a week begins on a Sunday, or on the first day of the year. Valid input types are a date, a timestamp, or an equivalent character value. The output is of type integer.

*WEEK function examples*

```
SELECT WEEK(DATE('2000-01-01')) AS w1
     , WEEK(DATE('2000-01-02')) AS w2
     , WEEK(DATE('2001-01-02')) AS w3
     , WEEK(DATE('2000-12-31')) AS w4
     , WEEK(DATE('2040-12-31')) AS w5
FROM sysibm.sysdummy1;
```

*ANSWER*

| W1 | W2 | W3 | W4 | W5 |
|---|---|---|---|---|
| 1 | 2 | 1 | 54 | 53 |

Both the first and last week of the year may be partial weeks. Likewise, from one year to the next, a particular day will often be in a different week (see Comparing Weeks).

### 7.2.234. WEEK_ISO

Returns an integer value, in the range 1 to 53, that is the "ISO" week number. An ISO week differs from an ordinary week in that it begins on a Monday and it neither ends nor begins at the exact end of the year. Instead, week 1 is the first week of the year to contain a Thursday. Therefore, it is possible for up to three days at the beginning of the year to appear in the last week of the previous year. As with ordinary weeks, not all ISO weeks contain seven days.

WEEK_ISO function example

```
WITH temp1 (n) AS
(VALUES (0)
   UNION ALL
 SELECT n+1
 FROM temp1
 WHERE n < 10
), temp2 (dt2) AS
(SELECT DATE('1998-12-27') + y.n YEARS + d.n DAYS
 FROM temp1 y
    , temp1 d
 WHERE y.n IN (0,2)
)
SELECT CHAR(dt2,ISO)              AS dte
     , SUBSTR(DAYNAME(dt2),1,3)   AS dy
     , WEEK(dt2)                  AS wk
     , DAYOFWEEK(dt2)             AS dy
     , WEEK_ISO(dt2)              AS wi
     , DAYOFWEEK_ISO(dt2)         AS di
FROM temp2
ORDER BY 1;
```

*ANSWER*

| DTE | DY | WK | DY | WI | DI |
|---|---|---|---|---|---|
| 1998-12-27 | Sun | 53 | 1 | 52 | 7 |
| 1998-12-28 | Mon | 53 | 2 | 53 | 1 |
| 1998-12-29 | Tue | 53 | 3 | 53 | 2 |
| 1998-12-30 | Wed | 53 | 4 | 53 | 3 |
| 1998-12-31 | Thu | 53 | 5 | 53 | 4 |
| 1999-01-01 | Fri | 1 | 6 | 53 | 5 |
| 1999-01-02 | Sat | 1 | 7 | 53 | 6 |
| 1999-01-03 | Sun | 2 | 1 | 53 | 7 |
| 1999-01-04 | Mon | 2 | 2 | 1 | 1 |
| 1999-01-05 | Tue | 2 | 3 | 1 | 2 |
| 1999-01-06 | Wed | 2 | 4 | 1 | 3 |
| 2000-12-27 | Wed | 53 | 4 | 52 | 3 |
| 2000-12-28 | Thu | 53 | 5 | 52 | 4 |
| 2000-12-29 | Fri | 53 | 6 | 52 | 5 |
| 2000-12-30 | Sat | 53 | 7 | 52 | 6 |
| 2000-12-31 | Sun | 54 | 1 | 52 | 7 |
| 2001-01-01 | Mon | 1 | 2 | 1 | 1 |

| DTE | DY | WK | DY | WI | DI |
|---|---|---|---|---|---|
| 2001-01-02 | Tue | 1 | 3 | 1 | 2 |
| 2001-01-03 | Wed | 1 | 4 | 1 | 3 |
| 2001-01-04 | Thu | 1 | 5 | 1 | 4 |
| 2001-01-05 | Fri | 1 | 6 | 1 | 5 |
| 2001-01-06 | Sat | 1 | 7 | 1 | 6 |

### 7.2.235. WEEKS_BETWEEN

Returns the number of full weeks between the specified arguments

### 7.2.236. WIDTH_BUCKET

Is used to create equal-width histograms.

### 7.2.237. XMLATTRIBUTES

Constructs XML attributes from the arguments.

### 7.2.238. XMLCOMMENT

Returns an XML value with a single XQuery comment node with the input argument as the content.

### 7.2.239. XMLCONCAT

Returns a sequence containing the concatenation of a variable number of XML input arguments

### 7.2.240. XMLDOCUMENT

Returns an XML value with a single XQuery document node with zero or more children nodes.

### 7.2.241. XMLELEMENT

Returns an XML value that is an XQuery element node.

### 7.2.242. XMLFOREST

Returns an XML value that is a sequence of XQuery element nodes.

### 7.2.243. XMLNAMESPACES

Constructs namespace declarations from the arguments.

### 7.2.244. XMLPARSE

Parses the argument as an XML document and returns an XML value.

### 7.2.245. XMLPI

Returns an XML value with a single XQuery processing instruction node.

### 7.2.246. XMLQUERY

Returns an XML value from the evaluation of an XQuery expression possibly using specified input arguments as XQuery variables.

### 7.2.247. XMLROW

Returns an XML value with a single XQuery document node containing one top-level element node.

### 7.2.248. XMLSERIALIZE

Returns a serialized XML value of the specified data type generated from the XML-expression argument.

### 7.2.249. XMLTEXT

Returns an XML value with a single XQuery text node having the input argument as the content.

### 7.2.250. XMLVALIDATE

Returns a copy of the input XML value augmented with information obtained from XML schema validation, including default values.

### 7.2.251. XMLXSROBJECTID

Returns the XSR object identifier of the XML schema used to validate the XML document specified in the argument. The XSR object identifier is returned as a BIGINT value and provides the key to a single row in SYSCAT.XSROBJECTS.

### 7.2.252. XSLTRANSFORM

Use XSLTRANSFORM to convert XML data into other formats, including the conversion of XML documents that conform to one XML schema into documents that conform to another schema.

### 7.2.253. YEAR

Returns a four-digit year value in the range 0001 to 9999 that represents the year (including the century). The input is a date or timestamp (or equivalent) value. The output is integer.

*YEAR and WEEK functions example*

```
SELECT dt1
     , YEAR(dt1) AS yr
     , WEEK(dt1) AS wk
FROM scalar;
```

*ANSWER*

| DT1 | YR | WK |
|---|---|---|
| 1996-04-22 | 1996 | 17 |
| 1996-08-15 | 1996 | 33 |
| 0001-01-01 | 1 | 1 |

### 7.2.254. YEARS_BETWEEN

Returns the number of full years between the specified arguments.

### 7.2.255. YMD_BETWEEN

Returns a numeric value that specifies the number of full years, full months, and full days between two datetime values.

### 7.2.256. "+" PLUS

The PLUS function is same old plus sign that you have been using since you were a kid. One can use it the old fashioned way, or as if it were normal a Db2 function - with one or two input items. If there is a single input item, then the function acts as the unary "plus" operator. If there are two items, the function adds them:

*PLUS function examples*

```
SELECT id
     , salary
     , "+"(salary)     AS s2
     , "+"(salary, id) AS s3
FROM staff
WHERE id < 40
ORDER BY id;
```

*ANSWER*

| ID | SALARY | S2 | S3 |
|---|---|---|---|
| 10 | 98357.50 | 98357.50 | 98367.50 |
| 20 | 78171.25 | 78171.25 | 78191.25 |
| 30 | 77506.75 | 77506.75 | 77536.75 |

Both the PLUS and MINUS functions can be used to add and subtract numbers, and also date and time values. For the latter, one side of the equation has to be a date/time value, and the other either a date or time duration (a numeric representation of a date/time), or a specified date/time type. To illustrate, below are three different ways to add one year to a date:

*Adding one year to date value*

```
SELECT empno
     , CHAR(birthdate,ISO)                            AS bdate1
     , CHAR(birthdate + 1 YEAR,ISO)                   AS bdate2
     , CHAR("+"(birthdate,DEC(00010000,8)),ISO)       AS bdate3
     , CHAR("+"(birthdate,DOUBLE(1),SMALLINT(1)),ISO) AS bdate4
FROM employee
WHERE empno < '000040'
ORDER BY empno;
```

*ANSWER*

| EMPNO | BDATE1 | BDATE2 | BDATE3 | BDATE4 |
|---|---|---|---|---|
| 000010 | 1933-08-24 | 1934-08-24 | 1934-08-24 | 1934-08-24 |
| 000020 | 1948-02-02 | 1949-02-02 | 1949-02-02 | 1949-02-02 |
| 000030 | 1941-05-11 | 1942-05-11 | 1942-05-11 | 1942-05-11 |

### 7.2.257. "-" MINUS

The MINUS works the same way as the PLUS function, but does the opposite:

*MINUS function examples*

```
SELECT id
     , salary
     , "-"(salary)     AS s2
     , "-"(salary, id) AS s3
FROM staff
WHERE id < 40
ORDER BY id;
```

*ANSWER*

| ID | SALARY | S2 | S3 |
|---|---|---|---|
| 10 | 98357.50 | -98357.50 | 98347.50 |
| 20 | 78171.25 | -78171.25 | 78151.25 |
| 30 | 77506.75 | -77506.75 | 77476.75 |

### 7.2.258. "*" MULTIPLY

The MULTIPLY function is used to multiply two numeric values:

*MULTIPLY function examples*

```
SELECT id
     , salary
     , salary * id     AS s2
     , "*"(salary, id) AS s3
FROM staff
WHERE id < 40
ORDER BY id;
```

*ANSWER*

| ID | SALARY | S2 | S3 |
|---|---|---|---|
| 10 | 98357.50 | 983575.00 | 983575.00 |
| 20 | 78171.25 | 1563425.00 | 1563425.00 |
| 30 | 77506.75 | 2325202.50 | 2325202.50 |

### 7.2.259. "/" DIVIDE

The DIVIDE function is used to divide two numeric values:

*DIVIDE function examples*

```
SELECT id
     , salary
     , salary / id     AS s2
     , "/"(salary, id) AS s3
FROM staff
WHERE id < 40
ORDER BY id;
```

*ANSWER*

| ID | SALARY | S2 | S3 |
|---|---|---|---|
| 10 | 98357.50 | 9835.75 | 9835.75 |
| 20 | 78171.25 | 3908.56 | 3908.56 |
| 30 | 77506.75 | 2583.55 | 2583.55 |

### 7.2.260. "||" CONCAT

Same as the CONCAT function:

*CONCAT function examples*

```
SELECT id
     , name || 'Z'      AS n1
     , name CONCAT 'Z'  AS n2
     , "||"(name,'Z')   AS n3
     , CONCAT(name,'Z') AS n4
FROM staff
WHERE LENGTH(name) < 5
ORDER BY id;
```

*ANSWER*

| ID | N1 | N2 | N3 | N4 |
|---|---|---|---|---|
| 110 | NganZ | NganZ | NganZ | NganZ |
| 210 | LuZ | LuZ | LuZ | LuZ |
| 270 | LeaZ | LeaZ | LeaZ | LeaZ |

# 7.3. User Defined Functions

Many problems that are really hard to solve using raw SQL become surprisingly easy to address, once one writes a simple function. This chapter will cover some of the basics of user defined functions. These can be very roughly categorized by their input source, their output type, and the language used:

- External scalar functions use an external process (e.g. a C program), and possibly also an external data source, to return a single value.
- External table functions use an external process, and possibly also an external data source, to return a set of rows and columns.
- Internal sourced functions are variations of an existing Db2 function
- Internal scalar functions use compound SQL code to return a single value.
- Internal table functions use compound SQL code to return a set of rows and columns.

This chapter will briefly go over the last three types of function listed above. See the official Db2 documentation for more details.

As of the time of writing, there is a known bug in Db2 that causes the prepare cost of a dynamic SQL statement to go up exponentially when a user defined function that is written in the SQL language is referred to multiple times in a single SQL statement.

## 7.3.1. Sourced Functions

A sourced function is used to redefine an existing Db2 function so as to in some way restrict or enhance its applicability. Below is a scalar function that is a variation on the standard DIGITS

function, but which only works on small integer fields:

*Create sourced function*

```
CREATE FUNCTION digi_int (SMALLINT)
RETURNS CHAR(5)
SOURCE SYSIBM.DIGITS(SMALLINT);
```

Here is an example of the function in use:

*Using sourced function - works*

```
SELECT id           AS ID
     , DIGITS(id)   AS I2
     , digi_int(id) AS I3
FROM staff
WHERE id < 40
ORDER BY id;
```

*ANSWER*

| ID | I2 | I3 |
|---|---|---|
| 10 | 00010 | 00010 |
| 20 | 00020 | 00020 |
| 30 | 00030 | 00030 |

By contrast, the following statement will fail because the input is an integer field:

*Using sourced function - fails*

```
SELECT id
     , digi_int(INT(id))
FROM staff
WHERE id < 50;
```

*ANSWER*: Error

Sourced functions are especially useful when one has created a distinct (data) type, because these do not come with any of the usual Db2 functions. To illustrate, in the following example a distinct type is created, then a table using the type, then two rows are inserted:

Create distinct type and test table

```
CREATE DISTINCT TYPE us_dollars AS DEC(7,2) WITH COMPARISONS;
CREATE TABLE customers
  ( ID      SMALLINT   NOT NULL
  , balance us_dollars NOT NULL);

INSERT INTO customers VALUES (1, 111.11), (2, 222.22);

SELECT *
FROM customers
ORDER BY ID;
```

*ANSWER*

| ID | balance |
|---|---|
| 1 | 111.11 |
| 2 | 222.22 |

The next query will fail because there is currently no multiply function for "us_dollars":

Do multiply - fails

```
SELECT id
     , balance * 10
FROM customers
ORDER BY id;
```

*ANSWER*: Error

The enable the above, we have to create a sourced function:

*Create sourced function*

```
CREATE FUNCTION "*" (us_dollars, INT)
RETURNS us_dollars
SOURCE SYSIBM."*"(DECIMAL, INT);
```

Now we can do the multiply:

*Do multiply - works*

```
SELECT id
     , balance * 10 AS newbal
FROM customers
ORDER BY id;
```

*ANSWER*

| ID | NEWBAL |
|---|---|
| 1 | 1111.10 |
| 2 | 2222.20 |

For the record, here is another way to write the same:

*Do multiply - works*

```
SELECT id
     , "*"(balance,10) AS newbal
FROM customers
ORDER BY id;
```

*ANSWER*

| ID | NEWBAL |
|---|---|
| 1 | 1111.10 |
| 2 | 2222.20 |

### 7.3.2. Scalar Functions

A scalar function has as input a specific number of values (i.e. not a table) and returns a single output item.

If a function returns a value (as opposed to a table), that value will always be nullable, regardless of whether or not the returned value can ever actually be null. This may cause problems if one is not prepared to handle a null indicator. To illustrate, the following function will return a nullable value that never be null:

*Function returns nullable, but never null, value*

```
CREATE FUNCTION Test()
RETURNS CHAR(5)
RETURN 'abcde';
```

#### Input and Output Limits

One can have multiple scalar functions with the same name and different input/output data types, but not with the same name and input/output types, but with different lengths. So if one wants to support all possible input/output lengths for, say, varchar data, one has to define the input and output lengths to be the maximum allowed for the field type. For varchar input, one would need an output length of 32,672 bytes to support all possible input values. But this is a problem, because it is very close to the maximum allowable table (row) length in Db2, which is 32,677 bytes. Decimal field types are even more problematic, because one needs to define both a length and a scale. To illustrate, imagine that one defines the input as being of type decimal(31,12). The following input values would be treated thus:

- A decimal(10,5) value would be fine.
- A decimal(31,31) value would lose precision.
- A decimal(31,0) value may fail because it is too large.
- See Convert Number to Character for a detailed description of this problem.

*Examples*

Below is a very simple scalar function - that always returns zero:

*Simple function usage*

```
CREATE FUNCTION returns_zero() RETURNS SMALLINT RETURN 0;

SELECT id             AS id
     , returns_zero() AS zz
FROM staff
WHERE id = 10;
```

*ANSWER*

| ID | ZZ |
|---|---|
| 10 | 0 |

Two functions can be created with the same name. Which one is used depends on the input type that is provided:

*Two functions with same name*

```
CREATE FUNCTION calc(inval SMALLINT) RETURNS INT RETURN inval * 10;
CREATE FUNCTION calc(inval INTEGER)  RETURNS INT RETURN inval * 5;

SELECT id                AS id
     , calc(SMALLINT(id)) AS c1
     , calc(INTEGER (id)) AS C2
FROM staff
WHERE id < 30
ORDER BY id;
```

*ANSWER*

| ID | C1 | C2 |
|---|---|---|
| 10 | 100 | 50 |
| 20 | 200 | 100 |

```
DROP FUNCTION calc(SMALLINT);
DROP FUNCTION calc(INTEGER);
```

Below is an example of a function that is not deterministic, which means that the function result can not be determined based on the input:

*Not deterministic function*

```
CREATE FUNCTION rnd(inval INT)
RETURNS SMALLINT
NOT DETERMINISTIC
RETURN RAND() * 50;

SELECT id     AS id
     , rnd(1) AS RND
FROM staff
WHERE id < 40
ORDER BY id;
```

*ANSWER*

| ID | RND |
|---|---|
| 10 | 37 |
| 20 | 8 |
| 30 | 42 |

The next function uses a query to return a single row/column value:

*Function using query*

```
CREATE FUNCTION get_sal(inval SMALLINT)
RETURNS DECIMAL(7, 2)
RETURN SELECT salary
       FROM staff
       WHERE id = inval;

SELECT id          AS id
     , get_sal(id) AS salary
FROM staff
WHERE id < 40
ORDER BY id;
```

*ANSWER*

| ID | SALARY |
|---|---|
| 10 | 98357.50 |

| ID | SALARY |
|---|---|
| 20 | 78171.25 |
| 30 | 77506.75 |

More complex SQL statements are also allowed - as long as the result (in a scalar function) is just one row/column value. In the next example, the either the maximum salary in the same department is obtained, or the maximum salary for the same year - whatever is higher:

*Function using common table expression*

```
CREATE FUNCTION max_sal(inval SMALLINT)
RETURNS DECIMAL(7, 2)
RETURN
WITH ddd (max_sal) AS
(SELECT MAX(S2.salary)
 FROM staff S1
    , staff S2
 WHERE S1.id   = inval
 AND   S1.dept = s2.dept
)
, yyy (max_sal) AS
(SELECT MAX(S2.salary)
 FROM staff S1
    , staff S2
 WHERE S1.id    = inval
 AND   S1.years = s2.years
)
SELECT
  CASE
    WHEN ddd.max_sal > yyy.max_sal THEN ddd.max_sal
    ELSE yyy.max_sal
  END
FROM ddd, yyy;

SELECT id          AS id
     , salary      AS SAL1
     , max_sal(id) AS SAL2
FROM staff
WHERE id < 40
ORDER BY id;
```

*ANSWER*

| ID | SAL1 | SAL2 |
|---|---|---|
| 10 | 98357.50 | 98357.50 |
| 20 | 78171.25 | 98357.50 |
| 30 | 77506.75 | 79260.25 |

A scalar or table function cannot change any data, but it can be used in a DML statement. In the next example, a function is used to remove all "e" characters from the name column:

*Function used in update*

```
CREATE FUNCTION remove_e(instr VARCHAR(50))
RETURNS VARCHAR(50)
RETURN replace(instr,'e','');

UPDATE staff
SET name = remove_e(name)
WHERE id < 40;
```

### 7.3.3. Compound SQL Usage

A function can use compound SQL, with the following limitations:

- The statement delimiter, if needed, cannot be a semi-colon.
- No DML statements are allowed.

Below is an example of a scalar function that uses compound SQL to reverse the contents of a text string:

*Function using compound SQL*

```
--#SET DELIMITER !

CREATE FUNCTION reverse(instr VARCHAR(50))
RETURNS VARCHAR(50)
BEGIN ATOMIC
    DECLARE outstr VARCHAR(50) DEFAULT '';
    DECLARE curbyte SMALLINT
    DEFAULT 0;
    SET curbyte = LENGTH(RTRIM(instr));
    WHILE curbyte >= 1 DO
        SET outstr = outstr || SUBSTR(instr,curbyte,1);
        SET curbyte = curbyte - 1;
    END WHILE;
    RETURN outstr;
END!

SELECT id AS id
     , name AS name1
     , reverse(name) AS name2
FROM staff
WHERE id < 40
ORDER BY id!
```

*ANSWER*

| ID | NAME1 | NAME2 |
|---|---|---|
| 10 | Sanders | srednaS |
| 20 | Pernal | lanreP |
| 30 | Marenghi | ihgneraM |

This example uses an "!" as the stmt delimiter.

Because compound SQL is a language with basic logical constructs, one can add code that does different things, depending on what input is provided. To illustrate, in the next example the possible output values are as follows:

- If the input is null, the output is set to null.
- If the length of the input string is less than 6, an error is flagged.
- If the length of the input string is less than 7, the result is set to -1.
- Otherwise, the result is the length of the input string.

Now for the code:

*Function with error checking logic*

```
--#SET DELIMITER !
CREATE FUNCTION check_len(instr VARCHAR(50))
RETURNS SMALLINT
BEGIN ATOMIC
    IF instr IS NULL THEN
        RETURN NULL;
    END IF;
    IF length(instr) < 6 THEN
        SIGNAL SQLSTATE '75001'
        SET MESSAGE_TEXT = 'Input string is < 6';
    ELSEIF length(instr) < 7 THEN
        RETURN -1;
    END IF;
    RETURN length(instr);
END!

SELECT id              AS id
     , name            AS name1
     , check_len(name) AS name2
FROM staff
WHERE id < 60
ORDER BY id!
```

This example uses an "!" as the stmt delimiter.

*ANSWER*

| ID | NAME1 | NAME2 |
|---|---|---|
| 10 | Sanders | 7 |
| 20 | Pernal | -1 |
| 30 | Marenghi | 8 |
| 40 | O’Brien | 7 |

The above query failed when it got to the name "Hanes", which is less than six bytes long.

# Chapter 8. Table Functions

A table function is very similar to a scalar function, except that it returns a set of rows and columns, rather than a single value. Here is an example:

*Simple table function*

```
CREATE FUNCTION get_staff()
RETURNS TABLE (  ID   SMALLINT
               , name VARCHAR(9)
               , YR   SMALLINT)
RETURN
  SELECT id
  , name
  , years
  FROM staff;

SELECT *
FROM TABLE(get_staff()) AS s
WHERE id < 40
ORDER BY id;
```

*ANSWER*

| ID | NAME | YR |
|---|---|---|
| 10 | Sanders | 7 |
| 20 | Pernal | 8 |
| 30 | Marenghi | 5 |

Note the following:

- The TABLE keyword, the function name (obviously), the two sets of parenthesis , and a correlation name, are all required.
- If the function has input parameters, they are all required, and their type must match.
- Optionally, one can list all of the columns that are returned by the function, giving each an assigned name.

Below is an example of a function that uses all of the above features:

*Table function with parameters*

```
CREATE FUNCTION get_st(inval INTEGER)
RETURNS
  TABLE (  id   SMALLINT
         , name VARCHAR(9)
         , yr   SMALLINT)
RETURN
  SELECT id
       , name
       , years
  FROM staff
  WHERE id = inval;

SELECT *
FROM TABLE(get_st(30)) AS sss (id, nnn, yy) *
```

*ANSWER*

| ID | NNN | YY |
|---|---|---|
| 30 | Marenghi | 5 |

Examples

A table function returns a table, but it doesn't have to touch a table. To illustrate, the following function creates the data on the fly:

*Table function that creates data*

```
CREATE FUNCTION make_data()
RETURNS
  TABLE ( KY SMALLINT
        , DAT CHAR(5))
RETURN
  WITH temp1 (k#) AS
  (VALUES (1),(2),(3))
  SELECT k#
       , DIGITS(SMALLINT(k#))
  FROM temp1;

SELECT *
FROM TABLE(make_data()) AS ttt;
```

*ANSWER*

| KY | DAT |
|---|---|
| 1 | 00001 |
| 2 | 00002 |

| KY | DAT |
|---|---|
| 3 | 00003 |

The next example uses compound SQL to first flag an error if one of the input values is too low, then find the maximum salary and related ID in the matching set of rows, then fetch the same rows - returning the two previously found values at the same time:

This example uses an "!" as the stmt delimiter.

*Table function with compound SQL*

```
CREATE FUNCTION staff_list(lo_key INTEGER
                         , lo_sal INTEGER)
RETURNS
  TABLE ( id      SMALLINT
        , salary  DECIMAL(7,2)
        , max_sal DECIMAL(7,2)
        , id_max SMALLINT)
LANGUAGE SQL
READS SQL DATA
EXTERNAL ACTION
DETERMINISTIC
BEGIN ATOMIC
  DECLARE hold_sal DECIMAL(7,2) DEFAULT 0;
  DECLARE hold_key SMALLINT;
  IF lo_sal < 0 THEN
    SIGNAL SQLSTATE '75001'
    SET MESSAGE_TEXT = 'Salary too low';
  END IF;
  FOR get_max AS
    SELECT id     AS in_key
         , salary As in_sal
    FROM staff
    WHERE id >= lo_key
  DO
    IF in_sal > hold_sal THEN
      SET hold_sal = in_sal;
      SET hold_key = in_key;
    END IF;
  END FOR;
RETURN
  SELECT id
       , salary
       , hold_sal
       , hold_key
  FROM staff
  WHERE id >= lo_key;
END!


SELECT *
FROM TABLE(staff_list(66,1)) AS ttt
WHERE id < 111
ORDER BY id!
```

*ANSWER*

| ID | SALARY | MAX_SAL | ID_MAX |
|---|---|---|---|
| 70 | 76502.83 | 91150.00 | 140 |
| 80 | 43504.60 | 91150.00 | 140 |
| 90 | 38001.75 | 91150.00 | 140 |
| 100 | 78352.80 | 91150.00 | 140 |
| 110 | 42508.20 | 91150.00 | 140 |

## 8.1. BASE_TABLE

Returns both the object name and schema name of the object found after any alias chains have been resolved.

## 8.2. UNNEST

Returns a result table that includes a row for each element of the specified array. If there are multiple ordinary array arguments specified, the number of rows will match the array with the largest cardinality.

## 8.3. XMLTABLE

Returns a result table from the evaluation of XQuery expressions, possibly using specified input arguments as XQuery variables. Each sequence item in the result sequence of the row XQuery expression represents a row of the result table.

# Chapter 9. Useful User-Defined Functions

In this section we will describe some simple functions that are generally useful, and that people have asked for over the years. In addition to the functions listed here, there are also the following elsewhere in this book:

- Check character input is a numeric value - Check Numeric function
- Convert numeric data to character (right justified) - User-defined functions - convert integer to character.
- Like-column predicate evaluation - LIKE_COLUMN Function.
- Locate string in input, a block at a time - LOCATE_BLOCK user defined function.
- Pause SQL statement (by looping) for "n" seconds - Function to Pause for "n" Seconds.
- Sort Character Field Contents.

## 9.1. Julian Date Functions

The function below converts a Db2 date into a Julian date (format) value:

*Convert Date into Julian Date*

```
CREATE FUNCTION julian_out(inval DATE)
RETURNS
  CHAR(7)
RETURN
  RTRIM(CHAR(YEAR(inval)))
        ||
        SUBSTR(DIGITS(DAYOFYEAR(inval)),8);

SELECT empno
     , CHAR(hiredate, ISO)  AS h_date
     , JULIAN_OUT(hiredate) AS j_date
FROM employee
WHERE empno < '000050'
ORDER BY empno;
```

*ANSWER*

| EMPNO | H_DATE | J_DATE |
|---|---|---|
| 000010 | 1995-01-01 | 1995001 |
| 000020 | 2003-10-10 | 2003283 |
| 000030 | 2005-04-05 | 2005095 |

The next function does the opposite:

*Convert Julian Date into Date*

```
CREATE FUNCTION julian_in(inval CHAR(7))
RETURNS
  DATE
RETURN
     DATE('0001-01-01')
  + (INT(SUBSTR(inval,1,4)) - 1) YEARS
  + (INT(SUBSTR(inval,5,3)) - 1) DAYS;
```

## 9.2. Get Prior Date

Imagine that one wanted to get all rows where some date is for the prior year - relative to the current year. This is easy to code:

*Select rows where hire-date = prior year*

```
SELECT empno
     , hiredate
FROM employee
WHERE YEAR(hiredate) = YEAR(CURRENT DATE) - 1;
```

## 9.3. Get Prior Month

One can use the DAYS function to get the same data for the prior day. But one cannot use the MONTH function to do the equivalent for the prior month because at the first of the year the month number goes back to one.

One can address this issue by writing a simple function that multiplies the year-number by 12, and then adds the month-number:

*Create year-month function*

```
CREATE FUNCTION year_month(inval DATE)
RETURNS
  INTEGER
RETURN
  (YEAR(inval) * 12) + MONTH(inval);
```

We can use this function thus:

*Select rows where hire-date = prior month*

```
SELECT empno
     , hiredate
FROM employee
WHERE YEAR_MONTH(hiredate) = YEAR_MONTH(CURRENT DATE) - 1;
```

## 9.4. Get Prior Week

Selecting rows for the prior week is complicated by the fact that both the US and ISO definitions of a week begin at one at the start of the year (see Comparing Weeks). If however we choose to define a week as a set of seven contiguous days, regardless of the date, we can create a function to do the job. In the example below we shall assume that a week begins on a Sunday:

*Create week-number function*

```
CREATE FUNCTION sunday_week(inval DATE)
RETURNS
  INTEGER
RETURN
  DAYS(inval) / 7;
```

The next function assumes that a week begins on a Monday:

*Create week-number function*

```
CREATE FUNCTION monday_week(inval DATE)
RETURNS
  INTEGER
RETURN
  (DAYS(inval) - 1) / 7;
```

Both the above functions convert the input date into a day-number value, then subtract (if needed) to get to the right day of the week, then divide by seven to get a week-number. The result is the number of weeks since the beginning of the current era. The next query shows the two functions in action:

*Use week-number functions*

```
WITH temp1 (num,dt) AS
(VALUES (1, DATE('2004-12-29'))
   UNION ALL
 SELECT num + 1
      , dt + 1 DAY
 FROM temp1
 WHERE num < 15
),
temp2 (dt, dy) AS
(SELECT dt
      , SUBSTR(DAYNAME(dt),1,3)
 FROM temp1
)
SELECT CHAR(dt, ISO)   AS date
     , dy              AS day
     , WEEK(dt)        AS wk
     , WEEK_ISO(dt)    AS is
     , sunday_week(dt) AS sun_wk
     , monday_week(dt) AS mon_wk
FROM temp2
ORDER BY 1;
```

*ANSWER*

| DATE | DAY | WK | IS | SUN_WK | MON_WK |
|---|---|---|---|---|---|
| 2004-12-29 | Wed | 53 | 53 | 104563 | 104563 |
| 2004-12-30 | Thu | 53 | 53 | 104563 | 104563 |
| 2004-12-31 | Fri | 53 | 53 | 104563 | 104563 |
| 2005-01-01 | Sat | 1 | 53 | 104563 | 104563 |
| 2005-01-02 | Sun | 2 | 53 | 104564 | 104563 |
| 2005-01-03 | Mon | 2 | 1 | 104564 | 104564 |
| 2005-01-04 | Tue | 2 | 1 | 104564 | 104564 |
| 2005-01-05 | Wed | 2 | 1 | 104564 | 104564 |
| 2005-01-06 | Thu | 2 | 1 | 104564 | 104564 |
| 2005-01-07 | Fri | 2 | 1 | 104564 | 104564 |
| 2005-01-08 | Sat | 2 | 1 | 104564 | 104564 |
| 2005-01-09 | Sun | 3 | 1 | 104565 | 104564 |
| 2005-01-10 | Mon | 3 | 2 | 104565 | 104565 |
| 2005-01-11 | Tue | 3 | 2 | 104565 | 104565 |
| 2005-01-12 | Wed | 3 | 2 | 104565 | 104565 |

### 9.4.1. Generating Numbers

The next function returns a table of rows. Each row consists of a single integer value , starting at zero, and going up to the number given in the input. At least one row is always returned. If the input value is greater than zero, the number of rows returned equals the input value plus one:

*Create num-list function*

```
CREATE FUNCTION NumList(max_num INTEGER)
RETURNS
  TABLE(num INTEGER)
LANGUAGE SQL
RETURN
  WITH temp1 (num) AS
  (VALUES (0)
     UNION ALL
   SELECT num + 1
   FROM temp1
   WHERE num < max_num
  )
  SELECT num
  FROM temp1;
```

Below are some queries that use the above function:

```
SELECT *
FROM TABLE(NumList(-1)) AS xxx;
```

*ANSWER*

| **1** |
| --- |
| 0 |

```
SELECT *
FROM TABLE(NumList(+0)) AS xxx;
```

| **1** |
| --- |
| 0 |

```
SELECT *
FROM TABLE(NumList(+3)) AS xxx;
```

| **1** |
| --- |
| 0 |

| 1 |
|---|
| 1 |
| 2 |
| 3 |

*Using num-list function*

```
SELECT *
FROM TABLE(NumList(CAST(NULL AS INTEGER))) AS xxx;
```

| 1 |
|---|
| 0 |

If this function did not always return one row, we might have to use a left-outer-join when joining to it. Otherwise the calling row might disappear from the answer-set because no row was returned.

To illustrate the function's usefulness, consider the following query, which returns the start and end date for a given set of activities:

*Select activity start & end date*

```
SELECT actno
     , emstdate
     , emendate
     , DAYS(emendate) - DAYS(emstdate) AS #days
FROM emp_act act
WHERE empno   = '000260'
AND   projno  = 'AD3113'
AND   actno   < 100
AND   emptime = 0.5
ORDER BY actno;
```

*ANSWER*

| ACTNO | EMSTDATE | EMENDATE | #DAYS |
|---|---|---|---|
| 70 | 2002-06-15 | 2002-07-01 | 16 |
| 80 | 2002-03-01 | 2002-04-15 | 45 |

Imagine that we wanted take the above output, and generate a row for each day between the start and end dates. To do this we first have to calculate the number of days between a given start and end, and then join to the function using that value:

*Generate one row per date between start & end dates (1 of 2)*

```
SELECT actno
     , #days
     , num
     , emstdate + num DAYS AS new_date
FROM
  (SELECT actno
        , emstdate
        , emendate
        , DAYS(emendate) - DAYS(emstdate) AS #days
   FROM emp_act act
   WHERE empno   = '000260'
   AND   projno  = 'AD3113'
   AND   actno   < 100
   AND   emptime = 0.5
) AS aaa
, TABLE(NumList(#days)) AS ttt
ORDER BY actno
       , num;
```

*ANSWER*

| ACTNO | #DAYS | NUM | NEW_DATE |
|---|---|---|---|
| 70 | 16 | 0 | 2002-06-15 |
| 70 | 16 | 1 | 2002-06-16 |
| 70 | 16 | 2 | 2002-06-17 |
| 70 | 16 | 3 | 2002-06-18 |
| 70 | 16 | 4 | 2002-06-19 |
| 70 | 16 | 5 | 2002-06-20 |
| 70 | 16 | 6 | 2002-06-21 |
| 70 | 16 | 7 | 2002-06-22 |
| 70 | 16 | 8 | 2002-06-23 |
| 70 | 16 | 9 | 2002-06-24 |
| 70 | 16 | 10 | 2002-06-25 |
| | | | etc... |

In the above query the #days value equals the number of days between the start and end dates. If the two dates equal, the #days value will be zero. In this case we will still get a row because the function will return a single zero value. If this were not the case (i.e. the function returned no rows if the input value was less than one), we would have to code a left-outer-join with a fake ON statement:

*Generate one row per date between start & end dates (2 of 2)*

```
SELECT actno
     , #days
     , num
     , emstdate + num DAYS AS new_date
FROM
  (SELECT actno
        , emstdate
        , emendate
        , DAYS(emendate)- DAYS(emstdate) AS #days
   FROM emp_act act
   WHERE empno   = '000260'
   AND   projno  = 'AD3113'
   AND   actno   < 100
   AND   emptime = 0.5
) AS aaa
LEFT OUTER JOIN
TABLE(NumList(#days)) AS ttt
ON 1 = 1
ORDER BY actno
       , num;
```

| ACTNO | #DAYS | NUM | NEW_DATE |
|---|---|---|---|
| 70 | 16 | 0 | 2002-06-15 |
| 70 | 16 | 1 | 2002-06-16 |
| 70 | 16 | 2 | 2002-06-17 |
| 70 | 16 | 3 | 2002-06-18 |
| 70 | 16 | 4 | 2002-06-19 |
| 70 | 16 | 5 | 2002-06-20 |
| 70 | 16 | 6 | 2002-06-21 |
| 70 | 16 | 7 | 2002-06-22 |
| 70 | 16 | 8 | 2002-06-23 |
| 70 | 16 | 9 | 2002-06-24 |
| 70 | 16 | 10 | 2002-06-25 |
| | | | etc... |

## 9.5. Check Data Value Type

The following function checks to see if an input value is character, where character is defined as meaning that all bytes are "A" through "Z" or blank. It converts (if possible) all bytes to blank using the TRANSLATE function, and then checks to see if the result is blank:

*Check if input value is character*

```
CREATE FUNCTION ISCHAR (inval VARCHAR(250))
RETURNS
  SMALLINT
LANGUAGE SQL
RETURN
  CASE
    WHEN TRANSLATE(UPPER(inval),' ','ABCDEFGHIJKLMNOPQRSTUVWXYZ') = ' ' THEN 1
    ELSE 0
  END;
```

The next function is similar to the prior, except that it looks to see if all bytes in the input are in the range of "0" through "9", or blank:

*Check if input value is numeric*

```
CREATE FUNCTION ISNUM (inval VARCHAR(250))
RETURNS
  SMALLINT
LANGUAGE SQL
RETURN
  CASE
    WHEN TRANSLATE(inval,' ','01234567890') = ' ' THEN 1
    ELSE 0
  END;
```

Below is an example of the above two functions in action:

*Example of functions in use*

```
WITH temp (indata) AS
(VALUES ('ABC')
      , ('123')
      , ('3.4')
      , ('-44')
      , ('A1 ')
      , ('  ')
)
SELECT indata         AS indata
     , ISCHAR(indata) AS c
     , ISNUM(indata) AS n
FROM temp;
```

*ANSWER*

| INDATA | C | N |
|---|---|---|
| ABC | 1 | 0 |

| INDATA | C | N |
|---|---|---|
| 123 | 0 | 1 |
| 3.4 | 0 | 0 |
| -44 | 0 | 0 |
| A1 | 0 | 0 |
| | 1 | 1 |

The above ISNUM function is a little simplistic. It doesn't check for all-blanks, or embedded blanks, decimal input, or sign indicators. The next function does all of this, and also indicates what type of number was found:

*Check if input value is numeric*

```
CREATE FUNCTION ISNUM2 (inval VARCHAR(255))
RETURNS
  CHAR(4)
LANGUAGE SQL
RETURN
  CASE
    WHEN inval = ' '                                        THEN ' '
    WHEN LOCATE(' ',RTRIM(LTRIM(inval))) > 0                THEN ' '
    WHEN TRANSLATE(inval,' ','01234567890') = inval         THEN ' '
    WHEN TRANSLATE(inval,' ','01234567890') = ' '           THEN 'INT '
    WHEN TRANSLATE(inval,' ','+01234567890') = ' '
     AND LOCATE('+',LTRIM(inval)) = 1
     AND LENGTH(REPLACE(inval,'+','')) = LENGTH(inval) - 1  THEN 'INT+'
    WHEN TRANSLATE(inval,' ','-01234567890') = ' '
     AND LOCATE('-',LTRIM(inval)) = 1
     AND LENGTH(REPLACE(inval,'-','')) = LENGTH(inval) - 1  THEN 'INT-'
    WHEN TRANSLATE(inval,' ','.01234567890') = ' '
     AND LENGTH(REPLACE(inval,'.','')) = LENGTH(inval) - 1  THEN 'DEC '
    WHEN TRANSLATE(inval,' ','+.01234567890') = ' '
     AND LOCATE('+',LTRIM(inval)) = 1
     AND LENGTH(REPLACE(inval,'+','')) = LENGTH(inval) - 1
     AND LENGTH(REPLACE(inval,'.','')) = LENGTH(inval) - 1  THEN 'DEC+'
    WHEN TRANSLATE(inval,' ','-.01234567890') = ' '
     AND LOCATE('-',LTRIM(inval)) = 1
     AND LENGTH(REPLACE(inval,'-','')) = LENGTH(inval) - 1
     AND LENGTH(REPLACE(inval,'.','')) = LENGTH(inval) - 1  THEN 'DEC-'
    ELSE ' '
  END;
```

The first three WHEN checks above are looking for non-numeric input:

- The input is blank.
- The input has embedded blanks.
- The input does not contain any digits.

The final five WHEN checks look for a specific types of numeric input. They are all similar in design, so we can use the last one (looking of negative decimal input) to illustrate how they all work:

- Check that the input consists only of digits, dots, the minus sign, and blanks.
- Check that the minus sign is the left-most non-blank character.
- Check that there is only one minus sign in the input.
- Check that there is only one dot in the input.

Below is an example of the above function in use:

*Example of function in use*

```
WITH temp (indata) AS
(VALUES ('ABC')
      , ('123')
      , ('3.4')
      , ('-44')
      , ('+11')
      , ('-1-')
      , ('12+')
      , ('+.1')
      , ('-0.')
      , ('  ')
      , ('1 1')
      , (' . ')
)
SELECT indata         AS indata
     , ISNUM2(indata) AS type
     , CASE
         WHEN ISNUM2(indata) <> '' THEN DEC(indata,5,2)
         ELSE NULL
       END           AS number
FROM temp;
```

*ANSWER*

| INDATA | TYPE | NUMBER |
|---|---|---|
| ABC | | - |
| 123 | INT | 123.00 |
| 3.4 | DEC | |
| 3.40 | -44 | INT |
| -44.00 | +11 | INT+ |
| 11.00 | -1- | |
| - | 12+ | |

| INDATA | TYPE | NUMBER |
|---|---|---|
| - | +.1 | DEC+ |
| 0.10 | -0. | DEC |
| 0.00 | | |
| - | 1 1 | |
| - | . | |

## 9.6. Hash Function

The following hash function is a little crude, but it works. It accepts a VARCHAR string as input, then walks the string and, one byte at a time, manipulates a floating point number. At the end of the process the floating point value is translated into BIGINT.

This example uses an "!" as the stmt delimiter.

*Create HASH_STRING function*

```
CREATE FUNCTION HASH_STRING (instr VARCHAR(30000))
RETURNS
  BIGINT
DETERMINISTIC
CONTAINS SQL
NO EXTERNAL ACTION
BEGIN ATOMIC
  DECLARE inlen SMALLINT;
  DECLARE curbit SMALLINT DEFAULT 1;
  DECLARE outnum DOUBLE DEFAULT 0;
  SET inlen = LENGTH(instr);
  WHILE curbit <= inlen
  DO
    SET outnum = (outnum * 123) + ASCII(SUBSTR(instr,curbit));
    IF outnum > 1E10 THEN
      SET outnum = outnum / 1.2345E6;
    END IF;
    SET curbit = curbit + 1;
  END WHILE;
  RETURN BIGINT(TRANSLATE(CHAR(outnum),'01','.E'));
END!
```

Below is an example of the function in use:

*HASH_STRING function usage*

```
SELECT id
     , name
     , HASH_STRING(name) AS hash_val
FROM staff s
WHERE id < 70
ORDER BY id!
```

*ANSWER*

| ID | NAME | HASH_VAL |
|---|---|---|
| 10 | Sanders | 203506538768383718 |
| 20 | Pernal | 108434258721263716 |
| 30 | Marenghi | 201743899927085914 |
| 40 | O'Brien | 202251277018590318 |
| 50 | Hanes | 103496977706763914 |
| 60 | Quigley | 202990889019520318 |

One way to judge a hash function is to look at the number of distinct values generated for a given number of input strings. Below is a very simple test:

*HASH_FUNCTION test*

```
WITH
temp1 (col1) AS
(VALUES (1)
   UNION ALL
 SELECT col1 + 1
 FROM temp1
 WHERE col1 < 100000
)
SELECT COUNT(*)                                    AS #rows
     , COUNT(DISTINCT HASH_STRING(CHAR(col1)))   AS #hash1
     , COUNT(DISTINCT HASH_STRING(DIGITS(col1))) AS #hash2
FROM temp1!
```

ANSWER

| #ROWS | #HASH1 | #HASH2 |
|---|---|---|
| 100000 | 100000 | 100000 |

# Chapter 10. Order By, Group By, and Having

## 10.1. Order By

The ORDER BY statement is used to sequence output rows.

**Notes**

One can order on any one of the following:

- A named column, or an expression, neither of which need to be in the select list.
- An unnamed column - identified by its number in the list of columns selected.
- The ordering sequence of a specific nested subselect.
- For an insert, the order in which the rows were inserted (see Insert Examples).

**Also note**:

- One can have multiple ORDER BY statements in a query, but only one per subselect.
- Specifying the same field multiple times in an ORDER BY list is allowed, but silly. Only the first specification of the field will have any impact on the output order.
- If the ORDER BY column list does not uniquely identify each row, any rows with duplicate values will come out in random order. This is almost always the wrong thing to do when the data is being displayed to an end-user.
- Use the TRANSLATE function to order data regardless of case. Note that this trick may not work consistently with some European character sets.
- NULL values sort high.

**Sample Data**

The following view is used throughout this section:

*ORDER BY sample data definition*

```
CREATE VIEW SEQ_DATA(col1,col2) AS
VALUES ('ab','xy')
     , ('AB','xy')
     , ('ac','XY')
     , ('AB','XY')
     , ('Ab','12');
```

### 10.1.1. Order by Examples

The following query presents the output in ascending order:

*Simple ORDER BY*

```
SELECT col1
     , col2
FROM seq_data
ORDER BY col1 ASC
       , col2;
```

**SEQ_DATA**

| COL1 | COL2 |
|---|---|
| ab | xy |
| AB | xy |
| ac | XY |
| AB | XY |
| Ab | 12 |

*ANSWER*

| COL1 | COL2 |
|---|---|
| AB | XY |
| AB | xy |
| Ab | 12 |
| ab | xy |
| ac | XY |

In the above example, all of the lower case data comes before any of the upper case data. Use the TRANSLATE function to display the data in case-independent order:

*Case insensitive ORDER BY*

```
SELECT col1
     , col2
FROM seq_data
ORDER BY TRANSLATE(col1) ASC
       , TRANSLATE(col2) ASC
```

*ANSWER*

| COL1 | COL2 |
|---|---|
| Ab | 12 |
| ab | xy |
| AB | xy |

| COL1 | COL2 |
|---|---|
| AB | XY |
| ac | XY |

One does not have to specify the column in the ORDER BY in the select list though, to the end-user, the data may seem to be random order if one leaves it out:

*ORDER BY on not-displayed column*

```
SELECT col2
FROM seq_data
ORDER BY col1
       , col2;
```

*ANSWER*

| COL2 |
|---|
| XY |
| xy |
| 12 |
| xy |
| XY |

In the next example, the data is (primarily) sorted in descending sequence, based on the second byte of the first column:

*ORDER BY second byte of first column*

```
SELECT col1
     , col2
FROM seq_data
ORDER BY SUBSTR(col1,2) DESC
       , col2
       , 1;
```

*ANSWER*

| COL1 | COL2 |
|---|---|
| ac | XY |
| Ab | 12 |
| ab | xy |
| AB | XY |
| AB | xy |

The standard ASCII collating sequence defines upper-case characters as being lower than lower-case (i.e. 'A' < 'a'), so upper-case characters display first if the data is ascending order. In the next example, this is illustrated using the HEX function is used to display character data in bit-data order:

*ORDER BY in bit-data sequence*

```
SELECT col1
     , HEX(col1) AS hex1
     , col2
     , HEX(col2) AS hex2
FROM seq_data
ORDER BY HEX(col1)
       , HEX(col2)
```

*ANSWER*

| COL1 | HEX1 | COL2 | HEX2 |
|---|---|---|---|
| AB | 4142 | XY | 5859 |
| AB | 4142 | xy | 7879 |
| Ab | 4162 | 12 | 3132 |
| ab | 6162 | xy | 7879 |
| ac | 6163 | XY | 5859 |

### 10.1.2. ORDER BY subselect

One can order by the result of a nested ORDER BY, thus enabling one to order by a column that is not in the input - as is done below:

-ORDER BY nested ORDER BY

```
SELECT col1
FROM
  (SELECT col1
   FROM seq_data
   ORDER BY col2
  ) AS xxx
ORDER BY ORDER OF xxx;
```

*ANSWER*

| COL1 |
|---|
| Ab |
| ac |

| COL1 |
| --- |
| AB |
| ab |
| AB |

In the next example the ordering of the innermost subselect is used, in part, to order the final output. This is done by first referring it to directly, and then indirectly:

*Multiple nested ORDER BY statements*

```
SELECT *
FROM
  (SELECT *
   FROM
     (SELECT *
      FROM seq_data
      ORDER BY col2
     ) AS xxx
   ORDER BY ORDER OF xxx
         , SUBSTR(col1, 2)
  ) AS yyy
ORDER BY ORDER OF yyy
               , col1;
```

*ANSWER*

| COL1 | COL2 |
| --- | --- |
| Ab | 12 |
| AB | XY |
| ac | XY |
| AB | xy |
| ac | xy |

### 10.1.3. ORDER BY inserted rows

One can select from an insert statement (see Insert Examples) to see what was inserted. Order by the INSERT SEQUENCE to display the rows in the order that they were inserted:

*ORDER BY insert input sequence*

```
SELECT empno
     , projno AS prj
     , actno AS act
     , ROW_NUMBER() OVER() AS r#
FROM
  FINAL TABLE
   (INSERT INTO emp_act (empno, projno, actno)
    VALUES ('400000','ZZZ',999)
         , ('400000','VVV',111)
   )
ORDER BY INPUT SEQUENCE;
```

*ANSWER*

| EMPNO | PRJ | ACT | R# |
|---|---|---|---|
| 400000 | ZZZ | 999 | 1 |
| 400000 | VVV | 111 | 2 |

The INPUT SEQUENCE phrase only works in an insert statement. It can be listed in the ORDER BY part of the statement, but not in the SELECT part. The select cannot be a nested table expression.

### 10.1.4. Group By and Having

The GROUP BY and GROUPING SETS statements are used to group individual rows into combined sets based on the value in one, or more, columns. The related ROLLUP and CUBE statements are short-hand forms of particular types of GROUPING SETS statement.

**Rules and Restrictions**

- There can only be one GROUP BY per SELECT. Multiple select statements in the same query can each have their own GROUP BY.
- Every field in the SELECT list must either be specified in the GROUP BY, or must have a column function applied against it.
- The result of a simple GROUP BY is always a distinct set of rows, where the unique identifier is whatever fields were grouped on.
- Only expressions returning constant values (e.g. a column name, a constant) can be referenced in a GROUP BY. For example, one cannot group on the RAND function as its result varies from one call to the next. To reference such a value in a GROUP BY, resolve it beforehand using a nested-table-expression.
- Variable length character fields with differing numbers on trailing blanks are treated as equal in the GROUP. The number of trailing blanks, if any, in the result is unpredictable.
- When grouping, all null values in the GROUP BY fields are considered equal.

- There is no guarantee that the rows resulting from a GROUP BY will come back in any particular order. If this is a problem, use an ORDER BY.

### 10.1.5. GROUP BY Flavors

A typical GROUP BY that encompasses one or more fields is actually a subset of the more general GROUPING SETS command. In a grouping set, one can do the following:

- Summarize the selected data by the items listed such that one row is returned per unique combination of values. This is an ordinary GROUP BY.
- Summarize the selected data using multiple independent fields. This is equivalent to doing multiple independent GROUP BY statements - with the separate results combined into one using UNION ALL statements.
- Summarize the selected data by the items listed such that one row is returned per unique combination of values, and also get various sub-totals, plus a grand-total. Depending on what exactly is wanted, this statement can be written as a ROLLUP, or a CUBE.

To illustrate the above concepts, imagine that we want to group some company data by team, department, and division. The possible sub-totals and totals that we might want to get are:

*Possible groupings*

```
GROUP BY division, department, team
GROUP BY division, department
GROUP BY division
GROUP BY division, team
GROUP BY department, team
GROUP BY department
GROUP BY team
GROUP BY ()   <= grand-total
```

If we wanted to get the first three totals listed above, plus the grand-total, we could write the statement one of three ways:

*Three ways to write the same GROUP BY*

```
GROUP BY division, department, team
UNION ALL
GROUP BY division, department
UNION ALL
GROUP BY division
UNION ALL
GROUP BY ()

GROUP BY GROUPING SETS ((division, department, team)
                      , (division, department)
                      , (division)
                      , ())

GROUP BY ROLLUP (division, department, team)
```

**Usage Warnings**

Before we continue, be aware of the following:

- Single vs. double parenthesis is a very big deal in grouping sets. When using the former, one is listing multiple independent groupings, while with the latter one is listing the set of items in a particular grouping.
- Repetition matters - sometimes. In an ordinary GROUP BY duplicate references to the same field has no impact on the result. By contrast, in a GROUPING SET, ROLLUP, or CUBE statement, duplicate references can often result in the same set of data being retrieved multiple times.

### 10.1.6. GROUP BY Sample Data

The following view will be used throughout this section:

*GROUP BY Sample Data*

```
CREATE VIEW employee_view (d1, dept, sex, salary) AS
VALUES ('A', 'A00', 'F', 52750)
     , ('A', 'A00', 'M', 29250)
     , ('A', 'A00', 'M', 46500)
     , ('B', 'B01', 'M', 41250)
     , ('C', 'C01', 'F', 23800)
     , ('C', 'C01', 'F', 28420)
     , ('C', 'C01', 'F', 38250)
     , ('D', 'D11', 'F', 21340)
     , ('D', 'D11', 'F', 22250)
     , ('D', 'D11', 'F', 29840)
     , ('D', 'D11', 'M', 18270)
     , ('D', 'D11', 'M', 20450)
     , ('D', 'D11', 'M', 24680)
     , ('D', 'D11', 'M', 25280)
     , ('D', 'D11', 'M', 27740)
     , ('D', 'D11', 'M', 32250);
```

**VIEW CONTENTS**

| D1 | DEPT | SEX | SALARY |
|---|---|---|---|
| A | A00 | F | 52750 |
| A | A00 | M | 29250 |
| A | A00 | M | 46500 |
| B | B01 | M | 41250 |
| C | C01 | F | 23800 |
| C | C01 | F | 28420 |
| C | C01 | F | 38250 |
| D | D11 | F | 21340 |
| D | D11 | F | 22250 |
| D | D11 | F | 29840 |
| D | D11 | M | 18270 |
| D | D11 | M | 20450 |
| D | D11 | M | 24680 |
| D | D11 | M | 25280 |
| D | D11 | M | 27740 |
| D | D11 | M | 32250 |

### 10.1.7. Simple GROUP BY Statements

A simple GROUP BY is used to combine individual rows into a distinct set of summary rows.

#### Sample Queries

In this first query we group our sample data by the leftmost three fields in the view:

*Simple GROUP BY*

```
SELECT d1
     , dept
     , sex
     , SUM(salary)        AS salary
     , SMALLINT(COUNT(*)) AS #rows
FROM employee_view
WHERE dept <> 'ABC'
GROUP BY d1
       , dept
       , sex
HAVING dept      >  'A0'
AND (SUM(salary) >  100
OR MIN(salary)   >  10
OR COUNT(*)      <> 22)
ORDER BY d1
       , dept
       , sex;
```

*ANSWER*

| D1 | DEPT | SEX | SALARY | #ROWS |
|---|---|---|---|---|
| A | A00 | F | 52750 | 1 |
| A | A00 | M | 75750 | 2 |
| B | B01 | M | 41250 | 1 |
| C | C01 | F | 90470 | 3 |
| D | D11 | F | 73430 | 3 |
| D | D11 | M | 148670 | 6 |

There is no need to have a field in the GROUP BY in the SELECT list, but the answer really doesn't make much sense if one does this:

*GROUP BY on non-displayed field*

```
SELECT sex
     , SUM(salary)        AS salary
     , SMALLINT(COUNT(*)) AS #rows
FROM employee_view
WHERE sex IN ('F','M')
GROUP BY dept
       , sex
ORDER BY sex;
```

*ANSWER*

| SEX | SALARY | #ROWS |
|---|---|---|
| F | 52750 | 1 |
| F | 90470 | 3 |
| F | 73430 | 3 |
| M | 75750 | 2 |
| M | 41250 | 1 |
| M | 148670 | 6 |

One can also do a GROUP BY on a derived field, which may, or may not be, in the statement SELECT list. This is an amazingly stupid thing to do:

*GROUP BY on derived field, not shown*

```
SELECT SUM(salary)        AS salary
     , SMALLINT(COUNT(*)) AS #rows
FROM employee_view
WHERE d1 <> 'X'
GROUP BY SUBSTR(dept,3,1)
HAVING COUNT(*) <> 99;
```

*ANSWER*

| SALARY | #ROWS |
|---|---|
| 128500 | 3 |
| 353820 | 13 |

One can not refer to the name of a derived column in a GROUP BY statement. Instead, one has to repeat the actual derivation code. One can however refer to the new column name in an ORDER BY:

*GROUP BY on derived field, shown*

```
SELECT SUBSTR(dept, 3, 1) AS wpart
     , SUM(salary) AS salary
     , SMALLINT(COUNT(*)) AS #rows
FROM employee_view
GROUP BY SUBSTR(dept, 3, 1)
ORDER BY wpart DESC;
```

*ANSWER*

| WPART | SALARY | #ROWS |
|---|---|---|
| 1 | 353820 | 13 |
| 0 | 128500 | 3 |

### 10.1.8. GROUPING SETS Statement

The GROUPING SETS statement enables one to get multiple GROUP BY result sets using a single statement. It is important to understand the difference between nested (i.e. in secondary parenthesis), and non-nested GROUPING SETS sub-phrases:

- A nested list of columns works as a simple GROUP BY.
- A non-nested list of columns works as separate simple GROUP BY statements, which are then combined in an implied UNION ALL.

*GROUPING SETS in parenthesis vs. not*

```
GROUP BY GROUPING SETS ((A,B,C)) is equivalent to GROUP BY A , B , C
GROUP BY GROUPING SETS (A,B,C) is equivalent to GROUP BY A UNION ALL
GROUP BY B UNION ALL GROUP BY C
GROUP BY GROUPING SETS (A,(B,C)) is equivalent to GROUP BY A UNION ALL
GROUP BY B , C
```

Multiple GROUPING SETS in the same GROUP BY are combined together as if they were simple fields in a GROUP BY list:

*Multiple GROUPING SETS*

```
GROUP BY GROUPING SETS (A) is equivalent to GROUP BY A
       , GROUPING SETS (B)                         , B
       , GROUPING SETS (C)                         , C

GROUP BY GROUPING SETS (A)      is equivalent to  GROUP BY A
       , GROUPING SETS ((B,C))                            , B
                                                          , C

GROUP BY GROUPING SETS (A)    is equivalent to  GROUP BY A
       , GROUPING SETS (B,C)                            , B
                                                UNION ALL
                                                GROUP BY A
                                                       , C
```

One can mix simple expressions and GROUPING SETS in the same GROUP BY:

*Simple GROUP BY expression and GROUPING SETS combined*

```
GROUP BY A                       is equivalent to GROUP BY A
       , GROUPING SETS ((B,C))                            , B
                                                          , C
```

Repeating the same field in two parts of the GROUP BY will result in different actions depending on the nature of the repetition. The second field reference is ignored if a standard GROUP BY is being made, and used if multiple GROUP BY statements are implied:

*Mixing simple GROUP BY expressions and GROUPING SETS*

```
GROUP BY A                         is equivalent to GROUP BY A
       , B                                                , B
       , GROUPING SETS ((B,C))                            , C

GROUP BY A                         is equivalent to GROUP BY A
       , B                                                , B
       , GROUPING SETS (B,C)                              , C
                                                UNION ALL
                                                GROUP BY A
                                                       , B

GROUP BY A                         is equivalent to GROUP BY A
       , B                                                , B
       , C                                                , C
       , GROUPING SETS (B,C)                    UNION ALL
                                                GROUP BY A
                                                       , B
                                                       , C
```

A single GROUPING SETS statement can contain multiple sets of (implied) GROUP BY phrases. These are combined using implied UNION ALL statements:

*GROUPING SETS with multiple components*

```
GROUP BY GROUPING SETS ((A,B,C)          is equivalent to GROUP BY A
                      , (A,B)                             , B
                      , (C))                              , C
                                                UNION ALL
                                                GROUP BY A
                                                       , B
                                                UNION ALL
                                                GROUP BY C

GROUP BY GROUPING SETS ((A)              is equivalent to GROUP BY A
                      , (B,C)                           UNION ALL
                      , (A)                             GROUP BY B
                      , A                                      , C
                      , ((C)))                          UNION ALL
                                                        GROUP BY A
                                                        UNION ALL
                                                        GROUP BY A
                                                        UNION ALL
                                                        GROUP BY C
```

The null-field list "( )" can be used to get a grand total. This is equivalent to not having the GROUP BY at all.

*GROUPING SET with multiple components, using grand-total*

```
GROUP BY GROUPING SETS ((A,B,C)          is equivalent to GROUP BY A
                      , (A,B)                             , B
                      , (A)                               , C
                      , ())                         UNION ALL
                                                    GROUP BY A
                                                           , B
is equivalent to                                    UNION ALL
                                                    GROUP BY A
                                                    UNION ALL
ROLLUP(A, B, C)                                     grand-totl
```

The above GROUPING SETS statement is equivalent to a ROLLUP(A,B,C), while the next is equivalent to a CUBE(A,B,C):

*GROUPING SET with multiple components, using grand-total*

```
GROUP BY GROUPING SETS ((A,B,C)
                      , (A,B)
                      , (A,C)
                      , (B,C)
                      , (A)
                      , (B)
                      , (C)
                      , ())


is equivalent to


CUBE(A,B,C)
```

```
is equivalent to GROUP BY A
                        , B
                        , C
                  UNION ALL
                  GROUP BY A
                        , B
                  UNION ALL
                  GROUP BY A
                        , C
                  UNION ALL
                  GROUP BY B
                        , C
                  UNION ALL
                  GROUP BY A
                  UNION ALL
                  GROUP BY B
                  UNION ALL
                  GROUP BY C
                  UNION ALL
                  grand-totl
```

### 10.1.9. SQL Examples

This first example has two GROUPING SETS. Because the second is in nested parenthesis, the result is the same as a simple three-field group by:

*Multiple GROUPING SETS, making one GROUP BY*

```
SELECT d1
     , dept
     , sex
     , SUM(salary)        AS sal
     , SMALLINT(COUNT(*)) AS #r
     , GROUPING(d1)       AS f1
     , GROUPING(dept)     AS fd
     , GROUPING(sex)      AS fs
     FROM employee_view
     GROUP BY GROUPING SETS (d1)
            , GROUPING SETS ((dept, sex))
     ORDER BY d1
            , dept
            , sex;
```

*ANSWER*

| D1 | DEPT | SEX | SAL | #R | DF | WF | SF |
|---|---|---|---|---|---|---|---|
| A | A00 | F | 52750 | 1 | 0 | 0 | 0 |

| D1 | DEPT | SEX | SAL | #R | DF | WF | SF |
|---|---|---|---|---|---|---|---|
| A | A00 | M | 75750 | 2 | 0 | 0 | 0 |
| B | B01 | M | 41250 | 1 | 0 | 0 | 0 |
| C | C01 | F | 90470 | 3 | 0 | 0 | 0 |
| D | D11 | F | 73430 | 3 | 0 | 0 | 0 |
| D | D11 | M | 148670 | 6 | 0 | 0 | 0 |

The GROUPING(field-name) column function is used in these examples to identify what rows come from which particular GROUPING SET. A value of 1 indicates that the corresponding data field is null because the row is from of a GROUPING SET that does not involve this row. Otherwise, the value is zero.

In the next query, the second GROUPING SET is not in nested-parenthesis. The query is therefore equivalent to GROUP BY D1, DEPT UNION ALL GROUP BY D1, SEX:

*Multiple GROUPING SETS, making two GROUP BY results*

```
SELECT d1
     , dept
     , sex
     , SUM(salary)         AS sal
     , SMALLINT(COUNT(*)) AS #r
     , GROUPING(d1)        AS f1
     , GROUPING(dept)      AS fd
     , GROUPING(sex)       AS fs
FROM employee_view
GROUP BY GROUPING SETS (d1)
       , GROUPING SETS (dept, sex)
ORDER BY d1
       , dept
       , sex;
```

*ANSWER*

| D1 | DEPT | SEX | SAL | #R | F1 | FD | FS |
|---|---|---|---|---|---|---|---|
| A | A00 | - | 128500 | 3 | 0 | 0 | 1 |
| A | - | F | 52750 | 1 | 0 | 1 | 0 |
| A | - | M | 75750 | 2 | 0 | 1 | 0 |
| B | B01 | - | 41250 | 1 | 0 | 0 | 1 |
| B | - | M | 41250 | 1 | 0 | 1 | 0 |
| C | C01 | - | 90470 | 3 | 0 | 0 | 1 |
| C | - | F | 90470 | 3 | 0 | 1 | 0 |
| D | D11 | - | 222100 | 9 | 0 | 0 | 1 |

| D1 | DEPT | SEX | SAL | #R | F1 | FD | FS |
|---|---|---|---|---|---|---|---|
| D | - | F | 73430 | 3 | 0 | 1 | 0 |
| D | - | M | 148670 | 6 | 0 | 1 | 0 |

It is generally unwise to repeat the same field in both ordinary GROUP BY and GROUPING SETS statements, because the result is often rather hard to understand. To illustrate, the following two queries differ only in their use of nested-parenthesis. Both of them repeat the DEPT field:

- In the first, the repetition is ignored, because what is created is an ordinary GROUP BY on all three fields.
- In the second, repetition is important, because two GROUP BY statements are implicitly generated. The first is on D1 and DEPT. The second is on D1, DEPT, and SEX.

*Repeated field essentially ignored*

```
SELECT d1
     , dept
     , sex
     , SUM(salary)        AS sal
     , SMALLINT(COUNT(*)) AS #r
     , GROUPING(d1)       AS f1
     , GROUPING(dept)     AS fd
     , GROUPING(sex)      AS fs
FROM employee_view
GROUP BY d1
       , dept
       , GROUPING SETS ((dept, sex))
ORDER BY d1
       , dept
       , sex;
```

*ANSWER*

| D1 | DEPT | SEX | SAL | #R | F1 | FD | FS |
|---|---|---|---|---|---|---|---|
| A | A00 | F | 52750 | 1 | 0 | 0 | 0 |
| A | A00 | M | 75750 | 2 | 0 | 0 | 0 |
| B | B01 | M | 41250 | 1 | 0 | 0 | 0 |
| C | C01 | F | 90470 | 3 | 0 | 0 | 0 |
| D | D11 | F | 73430 | 3 | 0 | 0 | 0 |
| D | D11 | M | 148670 | 6 | 0 | 0 | 0 |

*Repeated field impacts query result*

```
SELECT d1
     , dept
     , sex
     , SUM(salary) AS sal
     , SMALLINT(COUNT(*)) AS #r
     , GROUPING(d1) AS f1
     , GROUPING(dept) AS fd
     , GROUPING(sex) AS fs
FROM employee_view
GROUP BY d1
       , dept
       , GROUPING SETS (dept, sex)
ORDER BY d1
       , dept
       , sex;
```

*ANSWER*

| D1 | DEPT | SEX | SAL | #R | F1 | FD | FS |
|---|---|---|---|---|---|---|---|
| A | A00 | F | 52750 | 1 | 0 | 0 | 0 |
| A | A00 | M | 75750 | 2 | 0 | 0 | 0 |
| A | A00 | | 128500 | 3 | 0 | 0 | 1 |
| B | B01 | M | 41250 | 1 | 0 | 0 | 0 |
| B | B01 | | 41250 | 1 | 0 | 0 | 1 |
| C | C01 | F | 90470 | 3 | 0 | 0 | 0 |
| C | C01 | | 90470 | 3 | 0 | 0 | 1 |
| D | D11 | | 73430 | 3 | 0 | 0 | 0 |
| D | D11 | M | 148670 | 6 | 0 | 0 | 0 |
| D | D11 | | 222100 | 9 | 0 | 0 | 1 |

The above two queries can be rewritten as follows:

*Repeated field impacts query result*

```
GROUP BY d1                    is equivalent to GROUP BY d1
       , dept                                         , dept
       , GROUPING SETS ((dept, sex))                  , sex

GROUP BY d1                    is equivalent to GROUP BY d1
       , dept                                         , dept
       , GROUPING SETS (dept, sex)                    , sex
                                                UNION ALL
                                                GROUP BY d1
                                                       , dept
                                                       , dept
```

Repetitions of the same field in a GROUP BY (as is done above) are ignored during query processing. Therefore GROUP BY D1, DEPT, DEPT, SEX is the same as GROUP BY D1, DEPT, SEX.

### 10.1.10. ROLLUP Statement

A  expression displays sub-totals for the specified fields. This is equivalent to doing the original GROUP BY, and also doing more groupings on sets of the left-most columns.

*ROLLUP vs. GROUPING SETS*

```
GROUP BY ROLLUP(A,B,C) ===> GROUP BY GROUPING SETS((A,B,C)
                                                  , (A,B)
                                                  , (A)
                                                  , ())

GROUP BY ROLLUP(C,B)   ===> GROUP BY GROUPING SETS((C,B)
                                                  , (C)
                                                  , ())

GROUP BY ROLLUP(A)     ===> GROUP BY GROUPING SETS((A)
                                                  , ())
```

Imagine that we wanted to GROUP BY, but not ROLLUP one field in a list of fields. To do this, we simply combine the field to be removed with the next more granular field:

*ROLLUP vs. GROUPING SETS*

```
GROUP BY ROLLUP(A,(B,C))      ===> GROUP BY GROUPING SETS((A,B,C)
                                                        , (A)
                                                        , ())
```

Multiple ROLLUP statements in the same GROUP BY act independently of each other:

*ROLLUP vs. GROUPING SETS*

```
GROUP BY ROLLUP(A)              ===> GROUP BY GROUPING SETS((A,B,C)
       , ROLLUP(B,C)                                      , (A,B)
                                                          , (A)
                                                          , (B, C)
                                                          , (B)
                                                          , ())
```

One way to understand the above is to convert the two ROLLUP statement into equivalent grouping sets, and them "multiply" them - ignoring any grand-totals except when they are on both sides of the equation:

*Multiplying GROUPING SETS*

```
ROLLUP(A) * ROLLUP(B,C)                 = GROUPING SETS((A,B,C)
                                                      , (A,B)
                                                      , (A)
GROUPING SETS((A) * GROUPING SETS((B,C) =             , (B,C)
           , ())              , (B)                   , (B)
                              , ())                   , (())
```

**SQL Examples**

Here is a standard GROUP BY that gets no sub-totals:

*Simple GROUP BY*

```
SELECT dept
     , SUM(salary) AS salary
     , SMALLINT(COUNT(*)) AS #rows
     , GROUPING(dept) AS fd
FROM employee_view
GROUP BY dept
ORDER BY dept;
```

*ANSWER*

| DEPT | SALARY | #ROWS | FD |
|---|---|---|---|
| A00 | 128500 | 3 | 0 |
| B01 | 41250 | 1 | 0 |
| C01 | 0470 | 3 | 0 |
| D11 | 222100 | 9 | 0 |

Imagine that we wanted to also get a grand total for the above. Below is an example of using the ROLLUP statement to do this:

*GROUP BY with ROLLUP*

```
SELECT dept
, SUM(salary) AS salary
, SMALLINT(COUNT(*)) AS #rows
, GROUPING(dept) AS FD
FROM employee_view
GROUP BY ROLLUP(dept)
ORDER BY dept;
```

*ANSWER*

| DEPT | SALARY | #ROWS | FD |
|---|---|---|---|
| A00 | 128500 | 3 | 0 |
| B01 | 41250 | 1 | 0 |
| C01 | 90470 | 3 | 0 |
| D11 | 222100 | 9 | 0 |
| - | 482320 | 16 | 1 |

The GROUPING(field-name) function that is selected in the above example returns a one when the output row is a summary row, else it returns a zero.

Alternatively, we could do things the old-fashioned way and use a UNION ALL to combine the original GROUP BY with an all-row summary:

*ROLLUP done the old-fashioned way*

```
SELECT dept
     , SUM(salary)           AS salary
     , SMALLINT(COUNT(*))    AS #rows
     , GROUPING(dept)        AS fd
FROM employee_view
GROUP BY dept
UNION ALL
SELECT CAST(NULL AS CHAR(3)) AS dept
     , SUM(salary)           AS salary
     , SMALLINT(COUNT(*))    AS #rows
     , CAST(1 AS INTEGER)    AS fd
FROM employee_view
ORDER BY dept;
```

*ANSWER*

| DEPT | SALARY | #ROWS | FD |
|---|---|---|---|
| A00 | 128500 | 3 | 0 |

| DEPT | SALARY | #ROWS | FD |
|---|---|---|---|
| B01 | 41250 | 1 | 0 |
| C01 | 90470 | 3 | 0 |
| D11 | 222100 | 9 | 0 |
| - | 482320 | 16 | 1 |

Specifying a field both in the original GROUP BY, and in a ROLLUP list simply results in every data row being returned twice. In other words, the result is garbage:

*Repeating a field in GROUP BY and ROLLUP (error)*

```
SELECT dept
     , SUM(salary) AS salary
     , SMALLINT(COUNT(*)) AS #rows
     , GROUPING(dept) AS fd
FROM employee_view
GROUP BY dept
       , ROLLUP(dept)
ORDER BY dept;
```

*ANSWER*

| DEPT | SALARY | #ROWS | FD |
|---|---|---|---|
| A00 | 128500 | 3 | 0 |
| A00 | 128500 | 3 | 0 |
| B01 | 41250 | 1 | 0 |
| B01 | 41250 | 1 | 0 |
| C01 | 90470 | 3 | 0 |
| C01 | 90470 | 3 | 0 |
| D11 | 222100 | 9 | 0 |
| D11 | 222100 | 9 | 0 |

Below is a graphic representation of why the data rows were repeated above. Observe that two GROUP BY statements were, in effect, generated:

*Repeating a field, explanation*

```
GROUP BY dept           => GROUP BY dept                    => GROUP BY dept
       , ROLLUP(dept)             , GROUPING SETS((dept)       UNION ALL
                                                 , ())       GROUP BY dept
                                                                    , ()
```

In the next example the GROUP BY, is on two fields, with the second also being rolled up:

*GROUP BY on 1st field, ROLLUP on 2nd*

```
SELECT dept
     , sex
     , SUM(salary)        AS salary
     , SMALLINT(COUNT(*)) AS #rows
     , GROUPING(dept)     AS fd
     , GROUPING(sex)      AS fs
FROM employee_view
GROUP BY dept
       , ROLLUP(sex)
ORDER BY dept
       , sex;
```

*ANSWER*

| DEPT | SEX | SALARY | #ROWS | FD | FS |
|---|---|---|---|---|---|
| A00 | F | 52750 | 1 | 0 | 0 |
| A00 | M | 75750 | 2 | 0 | 0 |
| A00 | - | 128500 | 3 | 0 | 1 |
| B01 | M | 41250 | 1 | 0 | 0 |
| B01 | - | 41250 | 1 | 0 | 1 |
| C01 | F | 90470 | 3 | 0 | 0 |
| C01 | - | 90470 | 3 | 0 | 1 |
| D11 | F | 73430 | 3 | 0 | 0 |
| D11 | M | 148670 | 6 | 0 | 0 |
| D11 | - | 222100 | 9 | 0 | 1 |

The next example does a ROLLUP on both the DEPT and SEX fields, which means that we will get rows for the following:

- The work-department and sex field combined (i.e. the original raw GROUP BY).
- A summary for all sexes within an individual work-department.
- A summary for all work-departments (i.e. a grand-total).

*ROLLUP on DEPT, then SEX*

```
SELECT dept
     , sex
     , SUM(salary) AS salary
     , SMALLINT(COUNT(*)) AS #rows
     , GROUPING(dept) AS fd
     , GROUPING(sex) AS fs
FROM employee_view
GROUP BY ROLLUP(dept , sex)
ORDER BY dept , sex;
```

*ANSWER*

| DEPT | SEX | SALARY | #ROWS | FD | FS |
|---|---|---|---|---|---|
| A00 | F | 52750 | 1 | 0 | 0 |
| A00 | M | 75750 | 2 | 0 | 0 |
| A00 | - | 128500 | 3 | 0 | 1 |
| B01 | M | 41250 | 1 | 0 | 0 |
| B01 | - | 41250 | 1 | 0 | 1 |
| C01 | F | 90470 | 3 | 0 | 0 |
| C01 | - | 90470 | 3 | 0 | 1 |
| D11 | F | 73430 | 3 | 0 | 0 |
| D11 | M | 148670 | 6 | 0 | 0 |
| D11 | - | 222100 | 9 | 0 | 1 |
| - | - | 482320 | 16 | 1 | 1 |

In the next example we have reversed the ordering of fields in the ROLLUP statement. To make things easier to read, we have also altered the ORDER BY sequence. Now get an individual row for each sex and work-department value, plus a summary row for each sex:, plus a grand-total row:

*ROLLUP on SEX, then DEPT*

```
SELECT sex
     , dept
     , SUM(salary)        AS salary
     , SMALLINT(COUNT(*)) AS #rows
     , GROUPING(dept)     AS fd
     , GROUPING(sex)      AS fs
FROM employee_view
GROUP BY ROLLUP(sex
              , dept)
ORDER BY sex
       , dept;
```

*ANSWER*

| SEX | DEPT | SALARY | #ROWS | FD | FS |
|---|---|---|---|---|---|
| F | A00 | 52750 | 1 | 0 | 0 |
| F | C01 | 90470 | 3 | 0 | 0 |
| F | D11 | 73430 | 3 | 0 | 0 |
| F | | 216650 | 7 | 1 | 0 |
| M | A00 | 75750 | 2 | 0 | 0 |
| M | B01 | 41250 | 1 | 0 | 0 |
| M | D11 | 148670 | 6 | 0 | 0 |
| M | - | 265670 | 9 | 1 | 0 |
| - | - | 482320 | 16 | 1 | 1 |

The next statement is the same as the prior, but it uses the logically equivalent GROUPING SETS syntax:

*ROLLUP on SEX, then DEPT*

```
SELECT sex
     , dept
     , SUM(salary) AS salary
     , SMALLINT(COUNT(*)) AS #rows
     , GROUPING(dept) AS fd
     , GROUPING(sex) AS fs
FROM employee_view
GROUP BY GROUPING SETS ((sex, dept)
                      , (sex)
                      , ())
ORDER BY sex
       , dept;
```

*ANSWER*

| SEX | DEPT | SALARY | #ROWS | FD | FS |
|---|---|---|---|---|---|
| F | A00 | 52750 | 1 | 0 | 0 |
| F | C01 | 90470 | 3 | 0 | 0 |
| F | D11 | 73430 | 3 | 0 | 0 |
| F | - | 216650 | 7 | 1 | 0 |
| M | A00 | 75750 | 2 | 0 | 0 |
| M | B01 | 41250 | 1 | 0 | 0 |
| M | D11 | 148670 | 6 | 0 | 0 |
| M | - | 265670 | 9 | 1 | 0 |

| SEX | DEPT | SALARY | #ROWS | FD | FS |
|---|---|---|---|---|---|
| - | - | 482320 | 16 | 1 | 1 |

The next example has two independent rollups:

- The first generates a summary row for each sex.
- The second generates a summary row for each work-department.

The two together make a (single) combined summary row of all matching data. This query is the same as a UNION of the two individual rollups, but it has the advantage of being done in a single pass of the data. The result is the same as a CUBE of the two fields:

*Two independent ROLLUPS*

```
SELECT sex
     , dept
     , SUM(salary)        AS salary
     , SMALLINT(COUNT(*)) AS #rows
     , GROUPING(dept)     AS fd
     , GROUPING(sex)      AS fs
FROM employee_view
GROUP BY ROLLUP(sex)
      , ROLLUP(dept)
ORDER BY sex
       , dept;
```

*ANSWER*

| SEX | DEPT | SALARY | #ROWS | FD | FS |
|---|---|---|---|---|---|
| F | A00 | 52750 | 1 | 0 | 0 |
| F | C01 | 90470 | 3 | 0 | 0 |
| F | D11 | 73430 | 3 | 0 | 0 |
| F | - | 216650 | 7 | 1 | 0 |
| M | A00 | 75750 | 2 | 0 | 0 |
| M | B01 | 41250 | 1 | 0 | 0 |
| M | D11 | 148670 | 6 | 0 | 0 |
| M | - | 265670 | 9 | 1 | 0 |
| - | A00 | 128500 | 3 | 0 | 1 |
| - | B01 | 41250 | 1 | 0 | 1 |
| - | C01 | 90470 | 3 | 0 | 1 |
| - | D11 | 222100 | 9 | 0 | 1 |
| - | - | 482320 | 16 | 1 | 1 |

Below we use an inner set of parenthesis to tell the ROLLUP to treat the two fields as one, which causes us to only get the detailed rows, and the grand-total summary:

*Combined-field ROLLUP*

```
SELECT dept
     , sex
     , SUM(salary)        AS salary
     , SMALLINT(COUNT(*)) AS #rows
     , GROUPING(dept)     AS fd
     , GROUPING(sex)      AS fs
FROM employee_view
GROUP BY ROLLUP((dept,sex))
ORDER BY dept
       , sex;
```

*ANSWER*

| DEPT | SEX | SALARY | #ROWS | FD | FS |
|---|---|---|---|---|---|
| A00 | F | 52750 | 1 | 0 | 0 |
| A00 | M | 75750 | 2 | 0 | 0 |
| B01 | M | 41250 | 1 | 0 | 0 |
| C01 | F | 90470 | 3 | 0 | 0 |
| D11 | F | 73430 | 3 | 0 | 0 |
| D11 | M | 148670 | 6 | 0 | 0 |
| - | - | 482320 | 16 | 1 | 1 |

The HAVING statement can be used to refer to the two GROUPING fields. For example, in the following query, we eliminate all rows except the grand total:

*Use HAVING to get only grand-total row*

```
SELECT SUM(salary) AS salary
     , SMALLINT(COUNT(*)) AS #rows
FROM employee_view
GROUP BY ROLLUP(sex
              , dept)
HAVING GROUPING(dept) = 1  AND
       GROUPING(sex) = 1
ORDER BY salary;
```

*ANSWER*

| SALARY | #ROWS |
|---|---|
| 482320 | 16 |

Below is a logically equivalent SQL statement:

*Use GROUPING SETS to get grand-total row*

```
SELECT SUM(salary)        AS salary
     , SMALLINT(COUNT(*)) AS #rows
FROM employee_view
GROUP BY GROUPING SETS(());
```

*ANSWER*

| SALARY | #ROWS |
|---|---|
| 482320 | 16 |

Here is another:

*Use GROUP BY to get grand-total row*

```
SELECT SUM(salary)        AS salary
     , SMALLINT(COUNT(*)) AS #rows
FROM employee_view
GROUP BY ();
```

*ANSWER*

| SALARY | #ROWS |
|---|---|
| 482320 | 16 |

And another:

*Get grand-total row directly*

```
SELECT SUM(salary)        AS salary
     , SMALLINT(COUNT(*)) AS #rows
FROM employee_view;
```

*ANSWER*

| SALARY | #ROWS |
|---|---|
| 482320 | 16 |

### 10.1.11. CUBE Statement

A CUBE expression displays a cross-tabulation of the sub-totals for any specified fields. As such, it generates many more totals than the similar ROLLUP.

*CUBE vs. GROUPING SETS*

```
GROUP BY CUBE(A,B,C)          ===> GROUP BY GROUPING SETS((A,B,C)
                                                        , (A,B)
                                                        , (A,C)
                                                        , (B,C)
                                                        , (A)
                                                        , (B)
                                                        , (C)
                                                        , ())
GROUP BY CUBE(C,B)            ===> GROUP BY GROUPING SETS((C,B)
                                                        , (C)
                                                        , (B)
                                                        , ())
GROUP BY CUBE(A)              ===> GROUP BY GROUPING SETS((A)
                                                        , ())
```

As with the ROLLLUP statement, any set of fields in nested parenthesis is treated by the CUBE as a single field:

*CUBE vs. GROUPING SETS*

```
GROUP BY CUBE(A,(B,C))          ===>  GROUP BY GROUPING SETS((A,B,C)
                                                           , (B,C)
                                                           , (A)
                                                           , ())
```

Having multiple CUBE statements is allowed, but very, very silly:

*CUBE vs. GROUPING SETS*

```
GROUP BY CUBE(A,B)          ==> GROUPING SETS((A,B,C),(A,B),(A,B,C),(A,B)
       , CUBE(B,C)                          , (A,B,C),(A,B),(A,C),(A)
                                            , (B,C),(B),(B,C),(B)
                                            , (B,C),(B),(C),())
```

Obviously, the above is a lot of GROUPING SETS, and even more underlying GROUP BY statements. Think of the query as the Cartesian Product of the two CUBE statements, which are first resolved down into the following two GROUPING SETS:

```
((A,B),(A),(B),())
((B,C),(B),(C),())
```

### 10.1.12. SQL Examples

Below is a standard CUBE statement:

*CUBE example*

```
SELECT d1
     , dept
     , sex
     , INT(SUM(salary))   AS sal
     , SMALLINT(COUNT(*)) AS #r
     , GROUPING(d1)       AS f1
     , GROUPING(dept)     AS fd
     , GROUPING(sex)      AS fs
FROM employee_view
GROUP BY CUBE(d1, dept, sex)
ORDER BY d1
       , dept
       , sex;
```

*ANSWER*

| D1 | DEPT | SEX | SAL | #R | F1 | FD | FS |
|---|---|---|---|---|---|---|---|
| A | A00 | F | 52750 | 1 | 0 | 0 | 0 |
| A | A00 | M | 75750 | 2 | 0 | 0 | 0 |
| A | A00 | - | 128500 | 3 | 0 | 0 | 1 |
| A |  | F | 52750 | 1 | 0 | 1 | 0 |
| A |  | M | 75750 | 2 | 0 | 1 | 0 |
| A |  |  | 128500 | 3 | 0 | 1 | 1 |
| B | B01 | M | 41250 | 1 | 0 | 0 | 0 |
| B | B01 |  | 41250 | 1 | 0 | 0 | 1 |
| B |  | M | 41250 | 1 | 0 | 1 | 0 |
| B |  |  | 41250 | 1 | 0 | 1 | 1 |
| C | C01 | F | 90470 | 3 | 0 | 0 | 0 |
| C | C01 |  | 90470 | 3 | 0 | 0 | 1 |
| C |  | F | 90470 | 3 | 0 | 1 | 0 |
| C |  |  | 90470 | 3 | 0 | 1 | 1 |
| D | D11 | F | 73430 | 3 | 0 | 0 | 0 |
| D | D11 | M | 148670 | 6 | 0 | 0 | 0 |
| D | D11 |  | 222100 | 9 | 0 | 0 | 1 |
| D |  | F | 73430 | 3 | 0 | 1 | 0 |
| D |  | M | 148670 | 6 | 0 | 1 | 0 |
| D |  |  | 222100 | 9 | 0 | 1 | 1 |
| - | A00 | F | 2750 | 1 | 1 | 0 | 0 |

| D1 | DEPT | SEX | SAL | #R | F1 | FD | FS |
|---|---|---|---|---|---|---|---|
| - | A00 | M | 75750 | 2 | 1 | 0 | 0 |
| - | A00 |  | 128500 | 3 | 1 | 0 | 1 |
| - | B01 | M | 41250 | 1 | 1 | 0 | 0 |
| - | B01 |  | 41250 | 1 | 1 | 0 | 1 |
| - | C01 | F | 90470 | 3 | 1 | 0 | 0 |
| - | C01 |  | 90470 | 3 | 1 | 0 | 1 |
| - | D11 | F | 73430 | 3 | 1 | 0 | 0 |
| - | D11 | M | 148670 | 6 | 1 | 0 | 0 |
| - | D11 |  | 222100 | 9 | 1 | 0 | 1 |
| - |  | F | 216650 | 7 | 1 | 1 | 0 |
| - |  | M | 265670 | 9 | 1 | 1 | 0 |
| - |  |  | 482320 | 16 | 1 | 1 | 1 |

Here is the same query expressed as GROUPING SETS;

*CUBE expressed using multiple GROUPING SETS*

```
SELECT d1
     , dept
     , sex
     , INT(SUM(salary)) AS sal
     , SMALLINT(COUNT(*)) AS #r
     , GROUPING(d1) AS f1
     , GROUPING(dept) AS fd
     , GROUPING(sex) AS fs
FROM employee_view
GROUP BY GROUPING SETS ((d1, dept, sex)
                      , (d1,dept)
                      , (d1,sex)
                      , (dept,sex)
                      , (d1)
                      , (dept)
                      , (sex)
                      , ())
ORDER BY d1
       , dept
       , sex;
```

*ANSWER*

| D1 | DEPT | SEX | SAL | #R | F1 | FD | FS |
|---|---|---|---|---|---|---|---|
| A | A00 | F | 52750 | 1 | 0 | 0 | 0 |

| D1 | DEPT | SEX | SAL | #R | F1 | FD | FS |
|---|---|---|---|---|---|---|---|
| A | A00 | M | 75750 | 2 | 0 | 0 | 0 |

```
etc... (same as prior query)
```

A CUBE on a list of columns in nested parenthesis acts as if the set of columns was only one field. The result is that one gets a standard GROUP BY (on the listed columns), plus a row with the grand-totals:

*CUBE on compound fields*

```
SELECT d1
     , dept
     , sex
     , INT(SUM(salary))   AS sal
     , SMALLINT(COUNT(*)) AS #r
     , GROUPING(d1)       AS f1
     , GROUPING(dept)     AS fd
     , GROUPING(sex)      AS fs
FROM employee_VIEW
GROUP BY CUBE((d1, dept, sex))
ORDER BY d1
       , dept
       , sex;
```

*ANSWER*

| D1 | DEPT | SEX | SAL | #R | F1 | FD | FS |
|---|---|---|---|---|---|---|---|
| A | A00 | F | 52750 | 1 | 0 | 0 | 0 |
| A | A00 | M | 75750 | 2 | 0 | 0 | 0 |
| B | B01 | M | 41250 | 1 | 0 | 0 | 0 |
| C | C01 | F | 90470 | 3 | 0 | 0 | 0 |
| D | D11 | F | 73430 | 3 | 0 | 0 | 0 |
| D | D11 | M | 148670 | 6 | 0 | 0 | 0 |
| - | | | 482320 | 16 | 1 | 1 | 1 |

The above query is resolved thus:

*CUBE on compound field, explanation*

```
GROUP BY CUBE((A,B,C))      => GROUP BY GROUPING SETS((A,B,C)      => GROUP BY A
                                                    , ())                 , B
                                                                          , C
                                                                UNION ALL
                                                                GROUP BY()
```

### Complex Grouping Sets - Done Easy

Many of the more complicated SQL statements illustrated above are essentially unreadable because it is very hard to tell what combinations of fields are being rolled up, and what are not. There ought to be a more user-friendly way and, fortunately, there is. The CUBE command can be used to roll up everything. Then one can use ordinary SQL predicates to select only those totals and sub-totals that one wants to display.

Queries with multiple complicated ROLLUP and/or GROUPING SET statements sometimes fail to compile. In which case, this method can be used to get the answer.

To illustrate this technique, consider the following query. It summarizes the data in the sample view by three fields:

*Basic GROUP BY example*

```
SELECT d1                 AS d1
     , dept               AS dpt
     , sex                AS sx
     , INT(SUM(salary))   AS sal
     , SMALLINT(COUNT(*)) AS r
FROM employee_VIEW
GROUP BY d1
       , dept
       , sex
ORDER BY 1,2,3;
```

*ANSWER*

| D1 | DPT | SX | SAL | R |
|---|---|---|---|---|
| A | A00 | F | 52750 | 1 |
| A | A00 | M | 75750 | 2 |
| B | B01 | M | 41250 | 1 |
| C | C01 | F | 90470 | 3 |
| D | D11 | F | 73430 | 3 |
| D | D11 | M | 148670 | 6 |

Now imagine that we want to extend the above query to get the following sub-total rows:

*Sub-totals that we want to get*

```
DESIRED SUB-TOTALS                  EQUIVILENT TO
D1, DEPT, and SEX.                  GROUP BY GROUPING SETS ((d1,dept,sex)
D1 and DEPT.                                             , (d1,dept)
D1 and SEX.                                              , (d1,sex)
D1.                                                      , (d1)
SEX.                                                     , (sex)
Grand total.                        EQUIVILENT TO        , ())
                                    GROUP BY ROLLUP(d1,dept)
                                           , ROLLUP(sex)
```

Rather than use either of the syntaxes shown on the right above, below we use the CUBE expression to get all sub-totals, and then select those that we want:

*Get lots of sub-totals, using CUBE*

```
SELECT *
FROM (SELECT d1 AS d1
           , dept AS dpt
           , sex AS sx
           , INT(SUM(salary)) AS sal
           , SMALLINT(COUNT(*)) AS #r
           , SMALLINT(GROUPING(d1)) AS g1
           , SMALLINT(GROUPING(dept)) AS gd
           , SMALLINT(GROUPING(sex)) AS gs
      FROM EMPLOYEE_VIEW
      GROUP BY CUBE(d1,dept,sex)
      ) AS xxx
WHERE (g1,gd,gs) = (0,0,0)
   OR (g1,gd,gs) = (0,0,1)
   OR (g1,gd,gs) = (0,1,0)
   OR (g1,gd,gs) = (0,1,1)
   OR (g1,gd,gs) = (1,1,0)
   OR (g1,gd,gs) = (1,1,1)
ORDER BY 1,2,3;
```

*ANSWER*

| D1 | DPT | SX | SAL | #R | G1 | GD | GS |
|---|---|---|---|---|---|---|---|
| A | A00 | F | 52750 | 1 | 0 | 0 | 0 |
| A | A00 | M | 75750 | 2 | 0 | 0 | 0 |
| A | A00 | - | 128500 | 3 | 0 | 0 | 1 |
| A | | F | 52750 | 1 | 0 | 1 | 0 |
| A | | M | 75750 | 2 | 0 | 1 | 0 |

| D1 | DPT | SX | SAL | #R | G1 | GD | GS |
|---|---|---|---|---|---|---|---|
| A | | - | 128500 | 3 | 0 | 1 | 1 |
| B | B01 | M | 41250 | 1 | 0 | 0 | 0 |
| B | B01 | | 41250 | 1 | 0 | 0 | 1 |
| B | | M | 41250 | 1 | 0 | 1 | 0 |
| B | | | 41250 | 1 | 0 | 1 | 1 |
| C | C01 | F | 90470 | 3 | 0 | 0 | 0 |
| C | C01 | | 90470 | 3 | 0 | 0 | 1 |
| C | | F | 90470 | 3 | 0 | 1 | 0 |
| C | | | 90470 | 3 | 0 | 1 | 1 |
| D | D11 | F | 73430 | 3 | 0 | 0 | 0 |
| D | D11 | M | 148670 | 6 | 0 | 0 | 0 |
| D | D11 | - | 222100 | 9 | 0 | 0 | 1 |
| D | | F | 73430 | 3 | 0 | 1 | 0 |
| D | | M | 148670 | 6 | 0 | 1 | 0 |
| D | | - | 222100 | 9 | 0 | 1 | 1 |
| - | | F | 216650 | 7 | 1 | 1 | 0 |
| - | | M | 265670 | 9 | 1 | 1 | 0 |
| - | - | | 482320 | 16 | 1 | 1 | 1 |

In the above query, the GROUPING function (see GROUPING) is used to identify what fields are being summarized on each row. A value of one indicates that the field is being summarized; while a value of zero means that it is not. Only the following combinations are kept:

*Predicates used - explanation*

```
(G1,GD,GS) = (0,0,0) <== D1, DEPT, SEX
(G1,GD,GS) = (0,0,1) <== D1, DEPT
(G1,GD,GS) = (0,1,0) <== D1, SEX
(G1,GD,GS) = (0,1,1) <== D1,
(G1,GD,GS) = (1,1,0) <== SEX,
(G1,GD,GS) = (1,1,1) <== grand total
```

Here is the same query written using two ROLLUP expressions. You can be the judge as to which is the easier to understand:

*Get lots of sub-totals, using ROLLUP*

```
SELECT d1
     , dept
     , sex
     , INT(SUM(salary)) AS sal
     , SMALLINT(COUNT(*)) AS #r
FROM employee_view
GROUP BY ROLLUP(d1, dept)
       , ROLLUP(sex)
ORDER BY 1,2,3;
```

*ANSWER*

| D1 | DEPT | SEX | SAL | #R |
|---|---|---|---|---|
| A | A00 | F | 52750 | 1 |
| A | A00 | M | 75750 | 2 |
| A | A00 | | 128500 | 3 |
| A | | F | 52750 | 1 |
| A | | M | 75750 | 2 |
| A | | | 128500 | 3 |
| B | B01 | M | 41250 | 1 |
| B | B01 | | 41250 | 1 |
| B | | M | 41250 | 1 |
| B | | | 41250 | 1 |
| C | C01 | F | 90470 | 3 |
| C | C01 | | 90470 | 3 |
| C | | F | 90470 | 3 |
| C | | | 90470 | 3 |
| D | D11 | F | 73430 | 3 |
| D | D11 | M | 148670 | 6 |
| D | D11 | | 222100 | 9 |
| D | | F | 73430 | 3 |
| D | | M | 148670 | 6 |
| D | | | 222100 | 9 |
| - | | F | 216650 | 7 |
| - | | M | 265670 | 9 |
| - | | | 482320 | 16 |

### 10.1.13. Group By and Order By

One should never assume that the result of a GROUP BY will be a set of appropriately ordered rows because Db2 may choose to use a "strange" index for the grouping so as to avoid doing a row sort. For example, if one says "GROUP BY C1, C2" and the only suitable index is on C2 descending and then C1, the data will probably come back in index-key order.

*GROUP BY with ORDER BY*

```
SELECT dept
     , job
     , COUNT(*)
FROM staff
GROUP BY dept, job
ORDER BY dept, job;
```

Always code an ORDER BY if there is a need for the rows returned from the query to be specifically ordered - which there usually is._

### 10.1.14. Group By in Join

We want to select those rows in the STAFF table where the average SALARY for the employee's DEPT is greater than $18,000. Answering this question requires using a JOIN and GROUP BY in the same statement. The GROUP BY will have to be done first, then its' result will be joined to the STAFF table. There are two syntactically different, but technically similar, ways to write this query. Both techniques use a temporary table, but the way by which this is expressed differs. In the first example, we shall use a common table expression:

*GROUP BY on one side of join - using common table expression*

```
WITH staff2 (dept, avgsal) AS
 (SELECT dept
       , AVG(salary)
  FROM staff
  GROUP BY dept
  HAVING AVG(salary) > 18000
 )
SELECT a.id
     , a.name
     , a.dept
FROM staff a
   , staff2 b
WHERE a.dept = b.dept
ORDER BY a.id;
```

*ANSWER*

| ID | NAME | DEPT |
|---|---|---|
| 160 | Molinare | 10 |
| 210 | Lu | 10 |
| 240 | Daniels | 10 |
| 260 | Jones | 10 |

In the next example, we shall use a fullselect:

*GROUP BY on one side of join - using fullselect*

```
SELECT a.id
     , a.name
     , a.dept
FROM staff a
  , (SELECT dept AS dept
          , AVG(salary) AS avgsal
     FROM staff
     GROUP BY dept
     HAVING AVG(salary) > 18000
    ) AS b
WHERE a.dept = b.dept
ORDER BY a.id;
```

*ANSWER*

| ID | NAME | DEPT |
|---|---|---|
| 160 | Molinare | 10 |
| 210 | Lu | 10 |
| 240 | Daniels | 10 |
| 260 | Jones | 10 |

### 10.1.15. COUNT and No Rows

When there are no matching rows, the value returned by the COUNT depends upon whether this is a GROUP BY in the SQL statement or not:

*COUNT and No Rows*

```
SELECT COUNT(*) AS c1
FROM staff
WHERE id < 1;

ANSWER ==> 0

SELECT COUNT(*) AS c1
FROM staff
WHERE id < 1
GROUP BY id;

ANSWER ==> no row
```

see No Rows Match for a comprehensive discussion of what happens when no rows match.

# Chapter 11. Joins

A join is used to relate sets of rows in two or more logical tables. The tables are always joined on a row-by-row basis using whatever join criteria are provided in the query. The result of a join is always a new, albeit possibly empty, set of rows. In a join, the matching rows are joined side-by-side to make the result table. By contrast, in a union (see Union, Intersect, and Except) the matching rows are joined (in a sense) one-above-the-other to make the result table.

## 11.1. Why Joins Matter

The most important data in a relational database is not that stored in the individual rows. Rather, it is the implied relationships between sets of related rows. For example, individual rows in an EMPLOYEE table may contain the employee ID and salary - both of which are very important data items. However, it is the set of all rows in the same table that gives the gross wages for the whole company, and it is the (implied) relationship between the EMPLOYEE and DEPARTMENT tables that enables one to get a breakdown of employees by department and/or division. Joins are important because one uses them to tease the relationships out of the database. They are also important because they are very easy to get wrong.

## 11.2. Sample Views

*Sample Views used in Join Examples*

```
CREATE VIEW staff_v1 AS
SELECT id
     , name
FROM staff
WHERE id BETWEEN 10 AND 30;

CREATE VIEW staff_v2 AS
SELECT id
     , job
FROM staff
WHERE id BETWEEN 20 AND 50
  UNION ALL
SELECT id
    , 'Clerk' AS job
FROM staff
WHERE id = 30;
```

*STAFF_V1*

| ID | NAME |
|---|---|
| 10 | Sanders |
| 20 | Pernal |
| 30 | Marenghi |

*STAFF_V2*

| ID | JOB |
|---|---|
| 20 | Sales |
| 30 | Clerk |
| 30 | Mgr |
| 40 | Sales |
| 50 | Mgr |

Observe that the above two views have the following characteristics:

- Both views contain rows that have no corresponding ID in the other view.
- In the V2 view, there are two rows for ID of 30.

## 11.3. Join Syntax

Db2 SQL comes with two quite different ways to represent a join. Both syntax styles will be shown throughout this section though, in truth, ne of the styles is usually the better, depending upon the situation.
The first style, which is only really suitable for inner joins, involves listing the tables to be joined in a FROM statement. A comma separates each table name. A subsequent WHERE statement constrains the join.

Here are some sample joins:

*Sample two-table join*

```
SELECT v1.id
     , v1.name
     , v2.job
FROM staff_v1 v1
   , staff_v2 v2
WHERE v1.id = v2.id
ORDER BY v1.id
       , v2.job;
```

*JOIN ANSWER*

| ID | NAME | JOB |
|---|---|---|
| 20 | Pernal | Sales |
| 30 | Marenghi | Clerk |
| 30 | Marenghi | Mgr |

*Sample three-table join*

```
SELECT v1.id
     , v2.job
     , v3.name
FROM staff_v1 v1
   , staff_v2 v2
   , staff_v1 v3
WHERE v1.id = v2.id AND
      v2.id = v3.id AND
      v3.name LIKE 'M%'
ORDER BY v1.name
       , v2.job;
```

*JOIN ANSWER*

| ID | JOB | NAME |
|---|---|---|
| 30 | Clerk | Marenghi |
| 30 | Mgr | Marenghi |

The second join style, which is suitable for both inner and outer joins, involves joining the tables two at a time, listing the type of join as one goes. ON conditions constrain the join (note: there must be at least one), while WHERE conditions are applied after the join and constrain the result.

The following sample joins are logically equivalent to the two given above:

*Sample two-table inner join*

```
SELECT v1.id
     , v1.name
     , v2.job
FROM staff_v1 v1
INNER JOIN staff_v2 v2
ON v1.id = v2.id
ORDER BY v1.id
       , v2.job;
```

*JOIN ANSWER*

| ID | NAME | JOB |
|---|---|---|
| 20 | Pernal | Sales |
| 30 | Marenghi | Clerk |
| 30 | Marenghi | Mgr |

*Sample three-table inner join*

```
SELECT v1.id
     , v2.job
     , v3.name
FROM staff_v1 v1
JOIN staff_v2 v2
ON v1.id = v2.id
JOIN staff_v1 v3
ON v2.id = v3.id
WHERE v3.name LIKE 'M%'
ORDER BY v1.name
       , v2.job;
```

*STAFF_V1*

| ID | NAME |
|---|---|
| 10 | Sanders |
| 20 | Pernal |
| 30 | Marenghi |

*JOIN ANSWER*

| ID | JOB | NAME |
|---|---|---|
| 30 | Clerk | Marenghi |
| 30 | Mgr | Marenghi |

*STAFF_V2*

| ID | JOB |
|---|---|
| 20 | Sales |
| 30 | Clerk |
| 30 | Mgr |
| 40 | Sales |
| 50 | Mgr |

### 11.3.1. Query Processing Sequence

The following table lists the sequence with which various parts of a query are executed:

*Query Processing Sequence*

```
FROM clause
JOIN ON clause
WHERE clause
GROUP BY and aggregate
HAVING clause
SELECT list
ORDER BY clause
FETCH FIRST
```

Observe that ON predicates (e.g. in an outer join) are always processed before any WHERE predicates (in the same join) are applied. Ignoring this processing sequence can cause what looks like an outer join to run as an inner join - see ON and WHERE Usage.

### 11.3.2. ON vs. WHERE

A join written using the second syntax style shown above can have either, or both, ON and WHERE checks. These two types of check work quite differently:

- **WHERE** checks are used to filter rows, and to define the nature of the join. Only those rows that match all WHERE checks are returned.
- **ON** checks define the nature of the join. They are used to categorize rows as either joined or not-joined, rather than to exclude rows from the answer-set, though they may do this in some situations.

Let illustrate this difference with a simple, if slightly silly, left outer join:

*Sample Views used in Join Examples*

```
SELECT *
FROM staff_v1 v1
LEFT OUTER JOIN staff_v2 v2
ON 1 = 1
AND v1.id = v2.id
ORDER BY v1.id
       , v2.job;
```

*ANSWER*

| ID | NAME | ID | JOB |
|---|---|---|---|
| 10 | Sanders | - | - |
| 20 | Pernal | 20 | Sales |
| 30 | Marenghi | 30 | Clerk |
| 30 | Marenghi | 30 | Mgr |

Now lets replace the second ON check with a WHERE check:

*Sample Views used in Join Examples*

```
SELECT *
FROM staff_v1 v1
LEFT OUTER JOIN staff_v2 v2
ON 1 = 1
WHERE v1.id = v2.id
ORDER BY v1.id
       , v2.job;
```

*ANSWER*

| ID | NAME | ID | JOB |
|---|---|---|---|
| 20 | Pernal | 20 | Sales |
| 30 | Marenghi | 30 | Clerk |
| 30 | Marenghi | 30 | Mgr |

In the first example above, all rows were retrieved from the V1 view. Then, for each row, the two ON checks were used to find matching rows in the V2 view. In the second query, all rows were again retrieved from the V1 view. Then each V1 row was joined to every row in the V2 view using the (silly) ON check. Finally, the WHERE check (which is always done after the join) was applied to filter out all pairs that do not match on ID. Can an ON check ever exclude rows? The answer is complicated:

- In an inner join, an ON check can exclude rows because it is used to define the nature of the join and, by definition, in an inner join only matching rows are returned.
- In a partial outer join, an ON check on the originating table does not exclude rows. It simply categorizes each row as participating in the join or not.
- In a partial outer join, an ON check on the table to be joined to can exclude rows because if the row fails the test, it does not match the join.
- In a full outer join, an ON check never excludes rows. It simply categorizes them as matching the join or not.
- Each of the above principles will be demonstrated as we look at the different types of join.

## 11.3.3. Join Types

A generic join matches one row with another to create a new compound row. Joins can be categorized by the nature of the match between the joined rows. In this section we shall discuss each join type and how to code it in SQL.

#### Inner Join

An inner-join is another name for a standard join in which two sets of columns are joined by matching those rows that have equal data values. Most of the joins that one writes will probably be of this kind and, assuming that suitable indexes have been created, they will almost always be very efficient.

*STAFF_V1*

| ID | NAME |
|---|---|
| 10 | Sanders |
| 20 | Pernal |
| 30 | Marenghi |

*STAFF_V2*

| ID | JOB |
|---|---|
| 20 | Sales |
| 30 | Clerk |
| 30 | Mgr |
| 40 | Sales |
| 50 | Mgr |

**Join on ID**

*INNER-JOIN ANSWER*

| ID | NAME | ID | JOB |
|---|---|---|---|
| 20 | Pernal | 20 | Sales |
| 30 | Marenghi | 30 | Clerk |
| 30 | Marenghi | 30 | Mgr |

*Inner Join SQL (1 of 2)*

```
SELECT *
FROM staff_v1 v1
   , staff_v2 v2
WHERE v1.id = v2.id
ORDER BY v1.id
       , v2.job;
```

*ANSWER*

| ID | NAME | ID | JOB |
|---|---|---|---|
| 20 | Pernal | 20 | Sales |
| 30 | Marenghi | 30 | Clerk |
| 30 | Marenghi | 30 | Mgr |

*Inner Join SQL (2 of 2)*

```
SELECT *
FROM staff_v1 v1
INNER JOIN staff_v2 v2
ON v1.id = v2.id
ORDER BY v1.id
       , v2.job;
```

*ANSWER*

| ID | NAME | ID | JOB |
|---|---|---|---|
| 20 | Pernal | 20 | Sales |
| 30 | Marenghi | 30 | Clerk |
| 30 | Marenghi | 30 | Mgr |

### 11.3.4. ON and WHERE Usage

In an inner join only, an ON and a WHERE check work much the same way. Both define the nature of the join, and because in an inner join, only matching rows are returned, both act to exclude all rows that do not match the join. Below is an inner join that uses an ON check to exclude managers:

*Inner join, using ON check*

```
SELECT *
FROM staff_v1 v1
INNER JOIN staff_v2 v2
ON  v1.id  = v2.id
AND v2.job <> 'Mgr'
ORDER BY v1.id
       , v2.job;
```

*ANSWER*

| ID | NAME | ID | JOB |
|---|---|---|---|
| 20 | Pernal | 20 | Sales |
| 30 | Marenghi | 30 | Clerk |

Here is the same query written using a WHERE check

*Inner join, using WHERE check*

```
SELECT *
FROM staff_v1 v1
INNER JOIN staff_v2 v2
ON v1.id = v2.id
WHERE v2.job <> 'Mgr'
ORDER BY v1.id
       , v2.job;
```

*ANSWER*

| ID | NAME | ID | JOB |
|---|---|---|---|
| 20 | Pernal | 20 | Sales |
| 30 | Marenghi | 30 | Clerk |

### 11.3.5. Left Outer Join

A left outer join is the same as saying that I want all of the rows in the first table listed, plus any matching rows in the second table:

_STAFF_V1_

| ID | NAME |
|---|---|
| 10 | Sanders |
| 20 | Pernal |
| 30 | Marenghi |

_STAFF_V2_

| ID | JOB |
|---|---|
| 20 | Sales |
| 30 | Clerk |
| 30 | Mgr |
| 40 | Sales |
| 50 | Mgr |

*LEFT-OUTER-JOIN ANSWER*

*Table 9. Example of Left Outer Join*

| ID | NAME | ID | JOB |
|---|---|---|---|
| 10 | Sanders | - | |
| 20 | Pernal | 20 | Sales |

| ID | NAME | ID | JOB |
|---|---|---|---|
| 30 | Marenghi | 30 | Clerk |
| 30 | Marenghi | 30 | Mgr |

*Left Outer Join SQL (1 of 2)*

```
SELECT *
FROM staff_v1 v1
LEFT OUTER JOIN staff_v2 v2
ON v1.id = v2.id
ORDER BY 1,4;
```

It is possible to code a left outer join using the standard inner join syntax (with commas between tables), but it is a lot of work:

*Left Outer Join SQL (2 of 2)*

```
SELECT v1.*                                    -- (1)
     , v2.*
FROM staff_v1 v1
   , staff_v2 v2
WHERE v1.id = v2.id
UNION
SELECT v1.*                                    -- (2)
     , CAST(NULL AS SMALLINT) AS id
     , CAST(NULL AS CHAR(5)) AS job
FROM staff_v1 v1
WHERE v1.id NOT IN
      (SELECT id FROM staff_v2)
ORDER BY 1,4;
```

**(1)** This join gets all rows in STAFF_V1 that match rows in STAFF_V2. **(2)** This query gets all the rows in STAFF_V1 with no matching rows in STAFF_V2.

### 11.3.6. ON and WHERE Usage

In any type of join, a WHERE check works as if the join is an inner join. If no row matches, then no row is returned, regardless of what table the predicate refers to. By contrast, in a left or right outer join, an ON check works differently, depending on what table field it refers to:

- If it refers to a field in the table being joined to, it determines whether the related row matches the join or not.
- If it refers to a field in the table being joined from, it determines whether the related row finds a match or not. Regardless, the row will be returned.

In the next example, those rows in the table being joined to (i.e. the V2 view) that match on ID, and that are not for a manager are joined to:

*ON check on table being joined to*

```
SELECT *
FROM staff_v1 v1
LEFT OUTER JOIN staff_v2 v2
ON  v1.id  =  v2.id
AND v2.job <> 'Mgr'
ORDER BY v1.id
       , v2.job;
```

*ANSWER*

| ID | NAME | ID | JOB |
|---|---|---|---|
| 10 | Sanders | | |
| 20 | Pernal | 20 | Sales |
| 30 | Marenghi | 30 | Clerk |

If we rewrite the above query using a WHERE check we will lose a row (of output) because the check is applied after the join is done, and a null JOB does not match:

*WHERE check on table being joined to (1 of 2)*

```
SELECT *
FROM staff_v1 v1
LEFT OUTER JOIN staff_v2 v2
ON v1.id = v2.id
WHERE v2.job <> 'Mgr'
ORDER BY v1.id
       , v2.job;
```

*ANSWER*

| ID | NAME | ID | JOB |
|---|---|---|---|
| 20 | Pernal | 20 | Sales |
| 30 | Marenghi | 30 | Clerk |

We could make the WHERE equivalent to the ON, if we also checked for nulls:

*WHERE check on table being joined to (2 of 2)*

```
SELECT *
FROM staff_v1 v1
LEFT OUTER JOIN staff_v2 v2
ON v1.id = v2.id
WHERE (v2.job <> 'Mgr' OR
       v2.job IS NULL)
ORDER BY v1.id
       , v2.job;
```

*ANSWER*

| ID | NAME | ID | JOB |
|---|---|---|---|
| 10 | Sanders | | |
| 20 | Pernal | 20 | Sales |
| 30 | Marenghi | 30 | Clerk |

In the next example, those rows in the table being joined from (i.e. the V1 view) that match on ID and have a NAME > 'N' participate in the join. Note however that V1 rows that do not participate in the join (i.e. ID = 30) are still returned:

*ON check on table being joined from*

```
SELECT *
FROM staff_v1 v1
LEFT OUTER JOIN staff_v2 v2
ON v1.id = v2.id
AND v1.name > 'N'
ORDER BY v1.id
       , v2.job;
```

*ANSWER*

| ID | NAME | ID | JOB |
|---|---|---|---|
| 10 | Sanders | - | - |
| 20 | Pernal | 20 | Sales |
| 30 | Marenghi | - | - |

If we rewrite the above query using a WHERE check (on NAME) we will lose a row because now the check excludes rows from the answer-set, rather than from participating in the join:

*WHERE check on table being joined from*

```
SELECT *
FROM staff_v1 v1
LEFT OUTER JOIN staff_v2 v2
ON v1.id = v2.id
WHERE v1.name > 'N'
ORDER BY v1.id
       , v2.job;
```

*ANSWER*

| ID | NAME | ID | JOB |
|---|---|---|---|
| 10 | Sanders | | |
| 20 | Pernal | 20 | Sales |

Unlike in the previous example, there is no way to alter the above WHERE check to make it logically equivalent to the prior ON check. The ON and the WHERE are applied at different times and for different purposes, and thus do completely different things.

### 11.3.7. Right Outer Join

A right outer join is the inverse of a left outer join. One gets every row in the second table listed, plus any matching rows in the first table:

_STAFF_V1_

| ID | NAME |
|---|---|
| 10 | Sanders |
| 20 | Pernal |
| 30 | Marenghi |

_STAFF_V2_

| ID | JOB |
|---|---|
| 20 | Sales |
| 30 | Clerk |
| 30 | Mgr |
| 40 | Sales |
| 50 | Mgr |

*RIGHT-OUTER-JOIN ANSWER* .Example of Right Outer Join

| ID | NAME | ID | JOB |
|---|---|---|---|
| 20 | Pernal | 20 | Sales |
| 30 | Marenghi | 30 | Clerk |
| 30 | Marenghi | 30 | Mgr |
| | | 40 | Sales |
| | | 50 | Mgr |

*Right Outer Join SQL (1 of 2)*

```
SELECT *
FROM staff_v1 v1
RIGHT OUTER JOIN staff_v2 v2
ON v1.id = v2.id
ORDER BY v2.id
       , v2.job;
```

*ANSWER*

| ID | NAME | ID | JOB |
|---|---|---|---|
| 20 | Pernal | 20 | Sales |
| 30 | Marenghi | 30 | Clerk |
| 30 | Marenghi | 30 | Mgr |
| - | | 40 | Sales |
| - | | 50 | Mgr |

It is also possible to code a right outer join using the standard inner join syntax:

*Right Outer Join SQL (2 of 2)*

```
SELECT v1.*
     , v2.*
FROM staff_v1 v1
   , staff_v2 v2
WHERE v1.id = v2.id
UNION
SELECT CAST(NULL AS SMALLINT) AS id
     , CAST(NULL AS VARCHAR(9)) AS name
     , v2.*
FROM staff_v2 v2
WHERE v2.id NOT IN
      (SELECT id FROM staff_v1)
ORDER BY 3, 4;
```

*ANSWER*

| ID | NAME | ID | JOB |
|---|---|---|---|
| 20 | Pernal | 20 | Sales |
| 30 | Marenghi | 30 | Clerk |
| 30 | Marenghi | 30 | Mgr |
| | | 40 | Sales |
| | | 50 | Mgr |

**ON and WHERE Usage**

The rules for ON and WHERE usage are the same in a right outer join as they are for a left outer join (see Left Outer Join), except that the relevant tables are reversed.

### 11.3.8. Full Outer Joins

A full outer join occurs when all of the matching rows in two tables are joined, and there is also returned one copy of each non-matching row in both tables.

_STAFF_V1_

| ID | NAME |
|---|---|
| 10 | Sanders |
| 20 | Pernal |
| 30 | Marenghi |

_STAFF_V2_

| ID | JOB |
|---|---|
| 20 | Sales |
| 30 | Clerk |
| 30 | Mgr |
| 40 | Sales |
| 50 | Mgr |

*FULL-OUTER-JOIN ANSWER* .Example of Full Outer Join

| ID | NAME | ID | JOB |
|---|---|---|---|
| 10 | Sanders | | |
| 20 | Pernal | 20 | Sales |
| 30 | Marenghi | 30 | Clerk |
| 30 | Marenghi | 30 | Mgr |
| | | 40 | Sales |

| ID | NAME | ID | JOB |
|---|---|---|---|
| | | 50 | Mgr |

*Full Outer Join SQL*

```
SELECT *
FROM staff_v1 v1
FULL OUTER JOIN staff_v2 v2
ON v1.id = v2.id
ORDER BY v1.id
       , v2.id
       , v2.job;
```

*ANSWER*

| ID | NAME | ID | JOB |
|---|---|---|---|
| 10 | Sanders | | |
| 20 | Pernal | 20 | Sales |
| 30 | Marenghi | 30 | Clerk |
| 30 | Marenghi | 30 | Mgr |
| ´ | | 40 | Sales |
| | | 50 | Mgr |

Here is the same done using the standard inner join syntax:

*Full Outer Join SQL*

```
SELECT v1.*
     , v2.*
FROM staff_v1 v1
   , staff_v2 v2
WHERE v1.id = v2.id
UNION
SELECT v1.*
     , CAST(NULL AS SMALLINT) AS id
     , CAST(NULL AS CHAR(5)) AS job
FROM staff_v1 v1
WHERE v1.id NOT IN
      (SELECT id FROM staff_v2)
UNION
SELECT CAST(NULL AS SMALLINT) AS id
     , CAST(NULL AS VARCHAR(9)) AS name
     , v2.*
FROM staff_v2 v2
WHERE v2.id NOT IN
      (SELECT id FROM staff_v1)
ORDER BY 1,3,4;
```

*ANSWER*

| ID | NAME | ID | JOB |
|---|---|---|---|
| 10 | Sanders | | |
| 20 | Pernal | 20 | Sales |
| 30 | Marenghi | 30 | Clerk |
| 30 | Marenghi | 30 | Mgr |
| | | 40 | Sales |
| | | 50 | Mgr |

The above is reasonably hard to understand when two tables are involved, and it goes down hill fast as more tables are joined. Avoid.

#### ON and WHERE Usage

In a full outer join, an ON check is quite unlike a WHERE check in that it never results in a row being excluded from the answer set. All it does is categorize the input row as being either matching or non-matching. For example, in the following full outer join, the ON check joins those rows with equal key values:

*Full Outer Join, match on keys*

```
SELECT *
FROM staff_v1 v1
FULL OUTER JOIN staff_v2 v2
ON v1.id = v2.id
ORDER BY v1.id
       , v2.id
       , v2.job;
```

*ANSWER*

| ID | NAME | ID | JOB |
|---|---|---|---|
| 10 | Sanders | | |
| 20 | Pernal | 20 | Sales |
| 30 | Marenghi | 30 | Clerk |
| 30 | Marenghi | 30 | Mgr |
| | | 40 | Sales |
| | | 50 | Mgr |

In the next example, we have deemed that only those IDs that match, and that also have a value greater than 20, are a true match:

*Full Outer Join, match on keys > 20*

```
SELECT *
FROM staff_v1 v1
FULL OUTER JOIN staff_v2 v2
ON v1.id = v2.id
AND v1.id > 20
ORDER BY v1.id
       , v2.id
       , v2.job;
```

*ANSWER*

| ID | NAME | ID | JOB |
|---|---|---|---|
| 10 | Sanders | | |
| 20 | Pernal | | |
| 30 | Marenghi | 30 | Clerk |
| 30 | Marenghi | 30 | Mgr |
| | | 20 | Sales |
| | | 40 | Sales |

| ID | NAME | ID | JOB |
|---|---|---|---|
| | | 50 | Mgr |

Observe how in the above statement we added a predicate, and we got more rows! This is because in an outer join an ON predicate never removes rows. It simply categorizes them as being either matching or non-matching. If they match, it joins them. If they don't, it passes them through.

In the next example, nothing matches. Consequently, every row is returned individually. This query is logically similar to doing a UNION ALL on the two views:

*Full Outer Join, match on keys (no rows match)*

```
SELECT *
FROM staff_v1 v1
FULL OUTER JOIN staff_v2 v2
ON v1.id = v2.id
AND +1 = -1
ORDER BY v1.id
       , v2.id
       , v2.job;
```

*ANSWER*

| ID | NAME | ID | JOB |
|---|---|---|---|
| 10 | Sanders | | |
| 20 | Pernal | | |
| 30 | Marenghi | | |
| | | 20 | Sales |
| | | 30 | Clerk |
| | | 30 | Mgr |
| | | 40 | Sales |
| | | 50 | Mgr |

**ON** checks are somewhat like **WHERE** checks in that they have two purposes. Within a table, they are used to categorize rows as being either matching or non-matching. Between tables, they are used to define the fields that are to be joined on. In the prior example, the first **ON** check defined the fields to join on, while the second join identified those fields that matched the join. Because nothing matched (due to the second predicate), everything fell into the "outer join" category. This means that we can remove the first ON check without altering the answer set:

*Full Outer Join, don't match on keys (no rows match)*

```
SELECT *
FROM staff_v1 v1
FULL OUTER JOIN staff_v2 v2
ON +1 = -1
ORDER BY v1.id
       , v2.id
       , v2.job;
```

*ANSWER*

| ID | NAME | ID | JOB |
|---|---|---|---|
| 10 | Sanders | | |
| 20 | Pernal | | |
| 30 | Marenghi | | |
| | | 20 | Sales |
| | | 30 | Clerk |
| | | 30 | Mgr |
| | | 40 | Sales |
| | | 50 | Mgr |

What happens if everything matches and we don't identify the join fields? The result in a Cartesian Product:

```
SELECT *
FROM staff_v1 v1
FULL OUTER JOIN staff_v2 v2
ON +1 <> -1
ORDER BY v1.id
       , v2.id
       , v2.job;
```

_STAFF_V1_

| ID | NAME |
|---|---|
| 10 | Sanders |
| 20 | Pernal |
| 30 | Marenghi |

_STAFF_V2_

| ID | JOB |
|---|---|
| 20 | Sales |
| 30 | Clerk |
| 30 | Mgr |
| 40 | Sales |
| 50 | Mgr |

*ANSWER* .Full Outer Join, don't match on keys (all rows match)

| ID | NAME | ID | JOB |
|---|---|---|---|
| 10 | Sanders | 20 | Sales |
| 10 | Sanders | 30 | Clerk |
| 10 | Sanders | 30 | Mgr |
| 10 | Sanders | 40 | Sales |
| 10 | Sanders | 50 | Mgr |
| 20 | Pernal | 20 | Sales |
| 20 | Pernal | 30 | Clerk |
| 20 | Pernal | 30 | Mgr |
| 20 | Pernal | 40 | Sales |
| 20 | Pernal | 50 | Mgr |
| 30 | Marenghi | 20 | Sales |
| 30 | Marenghi | 30 | Clerk |
| 30 | Marenghi | 30 | Mgr |
| 30 | Marenghi | 40 | Sales |
| 30 | Marenghi | 50 | Mgr |

In an outer join, WHERE predicates behave as if they were written for an inner join. In particular, they always do the following:

- **WHERE** predicates defining join fields enforce an inner join on those fields.
- **WHERE** predicates on non-join fields are applied after the join, which means that when they are used on not-null fields, they negate the outer join.

Here is an example of a WHERE join predicate turning an outer join into an inner join:

*Full Outer Join, turned into an inner join by WHERE*

```
SELECT *
FROM staff_v1 v1
FULL JOIN staff_v2 v2
ON v1.id = v2.id
WHERE v1.id = v2.id
ORDER BY 1,3,4;
```

*ANSWER*

| ID | NAME | ID | JOB |
|---|---|---|---|
| 20 | Pernal | 20 | Sales |
| 30 | Marenghi | 30 | Clerk |
| 30 | Marenghi | 30 | Mgr |

To illustrate some of the complications that WHERE checks can cause, imagine that we want to do a FULL OUTER JOIN on our two test views (see below), limiting the answer to those rows where the "V1 ID" field is less than 30. There are several ways to express this query, each giving a different answer:

_STAFF_V1_

| ID | NAME |
|---|---|
| 10 | Sanders |
| 20 | Pernal |
| 30 | Marenghi |

_STAFF_V2_

| ID | JOB |
|---|---|
| 20 | Sales |
| 30 | Clerk |
| 30 | Mgr |
| 40 | Sales |
| 50 | Mgr |

*OUTER-JOIN CRITERIA* .Outer join V1.ID < 30, sample data

```
V1.ID = V2.ID V1.ID < 30

ANSWER ???, *_DEPENDS_*
```

In our first example, the "V1.ID < 30" predicate is applied after the join, which effectively eliminates all "V2" rows that don't match (because their "V1.ID" value is null):

*Outer join V1.ID < 30, check applied in WHERE (after join)*

```
SELECT *
FROM staff_v1 v1
FULL JOIN staff_v2 v2
ON v1.id = v2.id
WHERE v1.id < 30
ORDER BY 1,3,4;
```

*ANSWER*

| ID | NAME | ID | JOB |
|---|---|---|---|
| 10 | Sanders | | |
| 20 | Pernal | 20 | Sales |

In the next example the "V1.ID < 30" check is done during the outer join where it does not any eliminate rows, but rather limits those that match in the two views:

*Outer join V1.ID < 30, check applied in ON (during join)*

```
SELECT *
FROM staff_v1 v1
FULL JOIN staff_v2 v2
ON v1.id = v2.id
AND v1.id < 30
ORDER BY 1,3,4;
```

*ANSWER*

| ID | NAME | ID | JOB |
|---|---|---|---|
| 10 | Sanders | | |
| 20 | Pernal | 20 | Sales |
| 30 | Marenghi | | |
| | | 30 | Clerk |
| | | 30 | Mgr |
| | | 40 | Sales |
| | | 50 | Mgr |

Imagine that what really wanted to have the "V1.ID < 30" check to only apply to those rows in the "V1" table. Then one has to apply the check before the join, which requires the use of a nested-table expression:

*Outer join V1.ID < 30, check applied in WHERE (before join)*

```
SELECT *
FROM (SELECT *
      FROM staff_v1
      WHERE id < 30) AS v1
FULL OUTER JOIN staff_v2 v2
ON v1.id = v2.id
ORDER BY 1,3,4;
```

*ANSWER*

| ID | NAME | ID | JOB |
|---|---|---|---|
| 10 | Sanders | | |
| 20 | Pernal | 20 | Sales |
| | | 30 | Clerk |
| | | 30 | Mgr |
| | | 40 | Sales |
| | | 50 | Mgr |

Observe how in the above query we still got a row back with an ID of 30, but it came from the "V2" table. This makes sense, because the WHERE condition had been applied before we got to this table. There are several incorrect ways to answer the above question. In the first example, we shall keep all non-matching V2 rows by allowing to pass any null V1.ID values:

*Outer join V1.ID < 30, (gives wrong answer - see text)*

```
SELECT *
FROM staff_v1 v1
FULL OUTER JOIN staff_v2 v2
ON v1.id = v2.id
WHERE v1.id < 30
OR v1.id IS NULL
ORDER BY 1,3,4;
```

*ANSWER*

| ID | NAME | ID | JOB |
|---|---|---|---|
| 10 | Sanders | | |
| 20 | Pernal | 20 | Sales |
| | | 40 | Sales |
| | | 50 | Mgr |

There are two problems with the above query: First, it is only appropriate to use when the V1.ID

field is defined as not null, which it is in this case. Second, we lost the row in the V2 table where the ID equaled 30. We can fix this latter problem, by adding another check, but the answer is still wrong:

*Outer join V1.ID < 30, (gives wrong answer - see text)*

```
SELECT  *
FROM staff_v1 v1
FULL OUTER JOIN staff_v2 v2
ON v1.id = v2.id
WHERE v1.id < 30
OR v1.id = v2.id
OR v1.id IS NULL
ORDER BY 1,3,4;
```

*ANSWER*

| ID | NAME | ID | JOB |
|---|---|---|---|
| 10 | Sanders | | |
| 20 | Pernal | 20 | Sales |
| 30 | Marenghi | 30 | Clerk |
| 30 | Marenghi | 30 | Mgr |
| | | 40 | Sales |
| | | 50 | Mgr |

The last two checks in the above query ensure that every V2 row is returned. But they also have the affect of returning the NAME field from the V1 table whenever there is a match. Given our intentions, this should not happen.
**SUMMARY**: Query WHERE conditions are applied after the join. When used in an outer join, this means that they applied to all rows from all tables. In effect, this means that any WHERE conditions in a full outer join will, in most cases, turn it into a form of inner join.

### 11.3.9. Cartesian Product

A Cartesian Product is a form of inner join, where the join predicates either do not exist, or where they do a poor job of matching the keys in the joined tables.

_STAFF_V1_

| ID | NAME |
|---|---|
| 10 | Sanders |
| 20 | Pernal |
| 30 | Marenghi |

_STAFF_V2_

| ID | JOB |
|---|---|
| 20 | Sales |
| 30 | Clerk |
| 30 | Mgr |
| 40 | Sales |
| 50 | Mgr |

*Table 10. Example of Cartesian Product*

| ID | NAME | ID | JOB |
|---|---|---|---|
| 10 | Sanders | 20 | Sales |
| 10 | Sanders | 30 | Clerk |
| 10 | Sanders | 30 | Mgr |
| 10 | Sanders | 40 | Sales |
| 10 | Sanders | 50 | Mgr |
| 20 | Pernal | 20 | Sales |
| 20 | Pernal | 30 | Clerk |
| 20 | Pernal | 30 | Mgr |
| 20 | Pernal | 40 | Sales |
| 20 | Pernal | 50 | Mgr |
| 30 | Marenghi | 20 | Sales |
| 30 | Marenghi | 30 | Clerk |
| 30 | Marenghi | 30 | Mgr |
| 30 | Marenghi | 40 | Sales |
| 30 | Marenghi | 50 | Mgr |

Writing a Cartesian Product is simplicity itself. One simply omits the WHERE conditions:

*Cartesian Product SQL (1 of 2)*

```
SELECT *
FROM staff_v1 v1
   , staff_v2 v2
ORDER BY v1.id
       , v2.id
       , v2.job;
```

One way to reduce the likelihood of writing a full Cartesian Product is to always use the inner/outer join style. With this syntax, an ON predicate is always required. There is however no guarantee that the ON will do any good. Witness the following example:

*Cartesian Product SQL (2 of 2)*

```
SELECT *
FROM staff_v1 v1
INNER JOIN staff_v2 v2
ON 'A' <> 'B'
ORDER BY v1.id
       , v2.id
       , v2.job;
```

A Cartesian Product is almost always the wrong result. There are very few business situations where it makes sense to use the kind of SQL shown above. The good news is that few people ever make the mistake of writing the above. But partial Cartesian Products are very common, and they are also almost always incorrect. Here is an example:

*Partial Cartesian Product SQL*

```
SELECT v2a.id
     , v2a.job
     , v2b.id
FROM staff_v2 v2a
   , staff_v2 v2b
WHERE v2a.job = v2b.job
AND   v2a.id < 40
ORDER BY v2a.id
       , v2b.id;
```

*ANSWER*

| ID | JOB | ID |
|---|---|---|
| 20 | Sales | 20 |
| 20 | Sales | 40 |
| 30 | Clerk | 30 |
| 30 | Mgr | 30 |
| 30 | Mgr | 50 |

In the above example we joined the two views by JOB, which is not a unique key. The result was that for each JOB value, we got a mini Cartesian Product.
Cartesian Products are at their most insidious when the result of the (invalid) join is feed into a GROUP BY or DISTINCT statement that removes all of the duplicate rows. Below is an example where the only clue that things are wrong is that the count is incorrect:

*Partial Cartesian Product SQL, with GROUP BY*

```
SELECT v2.job
     , COUNT(*) AS #rows
FROM staff_v1 v1
   , staff_v2 v2
GROUP BY v2.job
ORDER BY #rows
       , v2.job;
```

*ANSWER*

| JOB | #ROWS |
|---|---|
| Clerk | 3 |
| Mgr | 6 |
| Sales | 6 |

To really mess up with a Cartesian Product you may have to join more than one table. Note however that big tables are not required. For example, a Cartesian Product of five 100-row tables will result in 10,000,000,000 rows being returned.

A good rule of thumb to use when writing a join is that for all of the tables (except one) there should be equal conditions on all of the fields that make up the various unique keys. If this is not true then it is probable that some kind Cartesian Product is being done and the answer may be wrong.

## 11.3.10. Join Notes

### Using the COALESCE Function

If you don't like working with nulls, but you need to do outer joins, then life is tough. In an outer join, fields in non-matching rows are given null values as placeholders. Fortunately, these nulls can be eliminated using the COALESCE function. The COALESCE function can be used to combine multiple fields into one, and/or to eliminate null values where they occur. The result of the COALESCE is always the first non-null value encountered. In the following example, the two ID fields are combined, and any null NAME values are replaced with a question mark.

*Use of COALESCE function in outer join*

```
SELECT COALESCE(v1.id, v2.id) AS id
     , COALESCE(v1.name, '?') AS name
     , v2.job
FROM staff_v1 v1
FULL OUTER JOIN staff_v2 v2
ON v1.id = v2.id
ORDER BY v1.id
       , v2.job;
```

*ANSWER*

| ID | NAME | JOB |
|---|---|---|
| 10 | Sanders | |
| 20 | Pernal | Sales |
| 30 | Marenghi | Clerk |
| 30 | Marenghi | Mgr |
| 40 | ? | Sales |
| 50 | ? | Mgr |

**Listing non-matching rows only**

Imagine that we wanted to do an outer join on our two test views, only getting those rows that do not match. This is a surprisingly hard query to write.

_STAFF_V1_

| ID | NAME |
|---|---|
| 10 | Sanders |
| 20 | Pernal |
| 30 | Marenghi |

_STAFF_V2_

| ID | JOB |
|---|---|
| 20 | Sales |
| 30 | Clerk |
| 30 | Mgr |
| 40 | Sales |
| 50 | Mgr |

*NON-MATCHING OUTER-JOIN ANSWER*

*Table 11. Example of outer join, only getting the non-matching rows*

| ID | NAME | ID | JOB |
|---|---|---|---|
| 10 | Sanders | | |
| | | 40 | Sales |
| | | 50 | Mgr |

One way to express the above is to use the standard inner-join syntax:

*Outer Join SQL, getting only non-matching rows*

```
SELECT v1.*                                    -- (1)
     , CAST(NULL AS SMALLINT) AS id
     , CAST(NULL AS CHAR(5)) AS job
FROM staff_v1 v1
WHERE v1.id NOT IN
      (SELECT id FROM staff_v2)
UNION
SELECT CAST(NULL AS SMALLINT) AS id            -- (2)
     , CAST(NULL AS VARCHAR(9)) AS name
     , v2.*
FROM staff_v2 v2
WHERE v2.id NOT IN
      (SELECT id FROM staff_v1)
ORDER BY 1,3,4;
```

**(1)** Get all the rows in STAFF_V1 that have no matching row in STAFF_V2.

**(2)** Get all the rows in STAFF_V2 that have no matching row in STAFF_V1.

The above question can also be expressed using the outer-join syntax, but it requires the use of two nested-table expressions. These are used to assign a label field to each table. Only those rows where either of the two labels are null are returned:

*Outer Join SQL, getting only non-matching rows*

```
SELECT *
FROM (SELECT v1.*
           , 'V1' AS flag
      FROM staff_v1 v1) AS v1
FULL OUTER JOIN
     (SELECT v2.*
           , 'V2' AS flag
      FROM staff_v2 v2) AS v2
ON v1.id = v2.id
WHERE v1.flag IS NULL
OR    v2.flag IS NULL
ORDER BY v1.id
       , v2.id
       , v2.job;
```

*ANSWER*

| ID | NAME | FLAG | ID | JOB | FLAG |
|---|---|---|---|---|---|
| 10 | Sanders | V1 | | | |
| | | | 40 | Sales | V2 |
| | | | 50 | Mgr | V2 |

Alternatively, one can use two common table expressions to do the same job:

*Outer Join SQL, getting only non-matching rows*

```
WITH v1 AS
  (SELECT v1.*
        , 'V1' AS flag
   FROM staff_v1 v1)
, v2 AS
  (SELECT v2.*
        , 'V2' AS flag
   FROM staff_v2 v2)
SELECT *
FROM v1 v1
FULL OUTER JOIN v2 v2
ON v1.id = v2.id
WHERE v1.flag IS NULL
OR    v2.flag IS NULL
ORDER BY v1.id
       , v2.id
       , v2.job;
```

*ANSWER*

| ID | NAME | FLAG | ID | JOB | FLAG |
|---|---|---|---|---|---|
| 10 | Sanders | V1 | | | |
| | | | 40 | Sales | V2 |
| | | | 50 | Mgr | V2 |

If either or both of the input tables have a field that is defined as not null, then label fields can be discarded. For example, in our test tables, the two ID fields will suffice:

*Outer Join SQL, getting only non-matching rows*

```
SELECT *
FROM staff_v1 v1
FULL OUTER JOIN staff_v2 v2
ON v1.id = v2.id
WHERE v1.id IS NULL
OR v2.id IS NULL
ORDER BY v1.id
       , v2.id
       , v2.job;
```

Join in SELECT Phrase

Imagine that we want to get selected rows from the V1 view, and for each matching row, get the corresponding JOB from the V2 view - if there is one:

_STAFF_V1_

| ID | NAME |
|---|---|
| 10 | Sanders |
| 20 | Pernal |
| 30 | Marenghi |

_STAFF_V2_

| ID | JOB |
|---|---|
| 20 | Sales |
| 30 | Clerk |
| 30 | Mgr |
| 40 | Sales |
| 50 | Mgr |

*Left outer join example*

```
V1.ID = V2.ID V1.ID <> 30
```

*ANSWER*

| ID | NAME | ID | JOB |
|---|---|---|---|
| 10 | Sanders | | |
| 20 | Pernal | 20 | Sales |

Here is one way to express the above as a query:

*Outer Join done in FROM phrase of SQL*

```
SELECT v1.id
     , v1.name
     , v2.job
FROM staff_v1 v1
LEFT OUTER JOIN staff_v2 v2
ON v1.id = v2.id
WHERE v1.id <> 30
ORDER BY v1.id ;
```

*ANSWER*

| ID | NAME | JOB |
|---|---|---|
| 10 | Sanders | |

| ID | NAME | JOB |
|---|---|---|
| 20 | Pernal | Sales |

Below is a logically equivalent left outer join with the join placed in the SELECT phrase of the SQL statement. In this query, for each matching row in STAFF_V1, the join (i.e. the nested table expression) will be done:

*Outer Join done in SELECT phrase of SQL*

```
SELECT v1.id
     , v1.name
     , (SELECT v2.job
        FROM staff_v2 v2
        WHERE v1.id = v2.id) AS jb
FROM staff_v1 v1
WHERE v1.id <> 30
ORDER BY v1.id;
```

*ANSWER*

| ID | NAME | JB |
|---|---|---|
| 10 | Sanders | |
| 20 | Pernal | Sales |

Certain rules apply when using the above syntax:

- The nested table expression in the SELECT is applied after all other joins and sub-queries (i.e. in the FROM section of the query) are done.
- The nested table expression acts as a left outer join.
- Only one column and row (at most) can be returned by the expression.
- If no row is returned, the result is null.

Given the above restrictions, the following query will fail because more than one V2 row is returned for every V1 row (for ID = 30):

*Outer Join done in SELECT phrase of SQL - gets error*

```
SELECT v1.id
     , v1.name
     , (SELECT v2.job
        FROM staff_v2 v2
        WHERE v1.id = v2.id) AS jb
FROM staff_v1 v1
ORDER BY v1.id;
```

*ANSWER*

| ID | NAME | JB |
|---|---|---|
| 10 | Sanders | |
| 20 | Pernal | Sales |

To make the above query work for all IDs, we have to decide which of the two matching JOB values for ID 30 we want. Let us assume that we want the maximum:

*Outer Join done in SELECT phrase of SQL - fixed*

```
SELECT v1.id
     , v1.name
     , (SELECT MAX(v2.job)
        FROM staff_v2 v2
        WHERE v1.id = v2.id) AS jb
FROM staff_v1 v1
ORDER BY v1.id;
```

*ANSWER*

| ID | NAME | JB |
|---|---|---|
| 10 | Sanders | |
| 20 | Pernal | Sales |
| 30 | Marenghi | Mgr |

The above is equivalent to the following query:

*Same as prior query - using join and GROUP BY*

```
SELECT v1.id
     , v1.name
     , MAX(v2.job) AS jb
FROM staff_v1 v1
LEFT OUTER JOIN staff_v2 v2
ON v1.id = v2.id
GROUP BY v1.id
       , v1.name
ORDER BY v1.id ;
```

*ANSWER*

| ID | NAME | JB |
|---|---|---|
| 10 | Sanders | |
| 20 | Pernal | Sales |
| 30 | Marenghi | Mgr |

The above query is rather misleading because someone unfamiliar with the data may not understand why the NAME field is in the GROUP BY. Obviously, it is not there to remove any rows, it simply needs to be there because of the presence of the MAX function. Therefore, the preceding query is better because it is much easier to understand. It is also probably more efficient.

#### CASE Usage

The SELECT expression can be placed in a CASE statement if needed. To illustrate, in the following query we get the JOB from the V2 view, except when the person is a manager, in which case we get the NAME from the corresponding row in the V1 view:

*Sample Views used in Join Examples*

```
SELECT v2.id
     , CASE
         WHEN v2.job <> 'Mgr' THEN v2.job
         ELSE (SELECT v1.name
               FROM staff_v1 v1
               WHERE v1.id = v2.id)
       END AS j2
FROM staff_v2 v2
ORDER BY v2.id
       , j2;
```

*ANSWER*

| ID | J2 |
|---|---|
| 20 | Sales |
| 30 | Clerk |
| 30 | Marenghi |
| 40 | Sales |
| 50 | - |

#### Multiple Columns

If you want to retrieve two columns using this type of join, you need to have two independent nested table expressions:

*Outer Join done in SELECT, 2 columns*

```
SELECT v2.id
     , v2.job
     , (SELECT v1.name
        FROM staff_v1 v1
        WHERE v2.id = v1.id)
     , (SELECT LENGTH(v1.name) AS n2
        FROM staff_v1 v1
        WHERE v2.id = v1.id)
FROM staff_v2 v2
ORDER BY v2.id
       , v2.job;
```

*ANSWER*

| ID | JOB | NAME | N2 |
|---|---|---|---|
| 20 | Sales | Pernal | 6 |
| 30 | Clerk | Marenghi | 8 |
| 30 | Mgr | Marenghi | 8 |
| 40 | Sales | | |
| 50 | Mgr | | |

An easier way to do the above is to write an ordinary left outer join with the joined columns in the SELECT list. To illustrate this, the next query is logically equivalent to the prior:

*Outer Join done in FROM, 2 columns*

```
SELECT v2.id
     , v2.job
     , v1.name
     , LENGTH(v1.name) AS n2
FROM staff_v2 v2
LEFT OUTER JOIN staff_v1 v1
ON v2.id = v1.id
ORDER BY v2.id
       , v2.job;
```

*ANSWER*

| ID | JOB | NAME | N2 |
|---|---|---|---|
| 20 | Sales | Pernal | 6 |
| 30 | Clerk | Marenghi | 8 |
| 30 | Mgr | Marenghi | 8 |
| 40 | Sales | | |

| ID | JOB | NAME | N2 |
|---|---|---|---|
| 50 | Mgr | | |

### Column Functions

This join style lets one easily mix and match individual rows with the results of column functions. For example, the following query returns a running SUM of the ID column:

*Running total, using JOIN in SELECT*

```
SELECT v1.id
     , v1.name
     , (SELECT SUM(x1.id)
        FROM staff_v1 x1
        WHERE x1.id <= v1.id
       )AS sum_id
FROM staff_v1 v1
ORDER BY v1.id
       , v2.job;
```

| ID | NAME | SUM_ID |
|---|---|---|
| 10 | Sanders | 10 |
| 20 | Pernal | 30 |
| 30 | Marenghi | 60 |

An easier way to do the same as the above is to use an OLAP function:

*Running total, using OLAP function*

```
SELECT v1.id
     , v1.name
     , SUM(id) OVER(ORDER BY id) AS sum_id
FROM staff_v1 v1
ORDER BY v1.id;
```

*ANSWER*

| ID | NAME | SUM_ID |
|---|---|---|
| 10 | Sanders | 10 |
| 20 | Pernal | 30 |
| 30 | Marenghi | 60 |

### Predicates and Joins, a Lesson

Imagine that one wants to get all of the rows in STAFF_V1, and to also join those matching rows in STAFF_V2 where the JOB begins with an 'S':

_STAFF_V1_

| ID | NAME |
|---|---|
| 10 | Sanders |
| 20 | Pernal |
| 30 | Marenghi |

_STAFF_V2_

| ID | JOB |
|---|---|
| 20 | Sales |
| 30 | Clerk |
| 30 | Mgr |
| 40 | Sales |
| 50 | Mgr |

*OUTER-JOIN CRITERIA*

*Outer join, with WHERE filter*

```
V1.ID = V2.ID
V2.JOB LIKE 'S%'
```

*ANSWER*

| ID | NAME | JOB |
|---|---|---|
| 10 | Sanders | |
| 20 | Pernal | Sales |
| 30 | Marenghi | |

The first query below gives the wrong answer. It is wrong because the WHERE is applied after the join, so eliminating some of the rows in the STAFF_V1 table:

*Outer Join, WHERE done after - wrong*

```
SELECT v1.id
     , v1.name
     , v2.job
FROM staff_v1 v1
LEFT OUTER JOIN staff_v2 v2
ON v1.id = v2.id
WHERE v2.job LIKE 'S%'
ORDER BY v1.id
       , v2.job;
```

*ANSWER (WRONG)*

| ID | NAME | JOB |
|---|---|---|
| 20 | Pernal | Sales |

In the next query, the WHERE is moved into a nested table expression - so it is done before the join (and against STAFF_V2 only), thus giving the correct answer:

*Outer Join, WHERE done before - correct*

```
SELECT v1.id
     , v1.name
     , v2.job
FROM staff_v1 v1
LEFT OUTER JOIN
    (SELECT *
     FROM staff_v2
     WHERE job LIKE 'S%'
    ) AS v2
ON v1.id = v2.id
ORDER BY v1.id
       , v2.job;
```

*ANSWER*

| ID | NAME | JOB |
|---|---|---|
| 10 | Sanders | |
| 20 | Pernal | Sales |
| 30 | Marenghi | |

The next query does the join in the SELECT phrase. In this case, whatever predicates are in the nested table expression apply to STAFF_V2 only, so we get the correct answer:

*Outer Join, WHERE done independently - correct*

```
SELECT v1.id
     , v1.name
     , (SELECT v2.job
        FROM staff_v2 v2
        WHERE v1.id = v2.id
        AND v2.job LIKE 'S%')
FROM staff_v1 v1
ORDER BY v1.id
       , job;
```

*ANSWER*

| ID | NAME | JOB |
|---|---|---|
| 10 | Sanders | |
| 20 | Pernal | Sales |
| 30 | Marenghi | |

**Joins - Things to Remember**

- You get nulls in an outer join, whether you want them or not, because the fields in nonmatching rows are set to null. If they bug you, use the COALESCE function to remove them. See Using the COALESCE Function for an example.
- From a logical perspective, all WHERE conditions are applied after the join. For performance reasons, Db2 may apply some checks before the join, especially in an inner join, where doing this cannot affect the result set.
- All WHERE conditions that join tables act as if they are doing an inner join, even when they are written in an outer join.
- The ON checks in a full outer join never remove rows. They simply determine what rows are matching versus not (see ON and WHERE Usage). To eliminate rows in an outer join, one must use a WHERE condition.
- The ON checks in a partial outer join work differently, depending on whether they are against fields in the table being joined to, or joined from (ON and WHERE Usage).
- A Cartesian Product is not an outer join. It is a poorly matching inner join. By contrast, a true outer join gets both matching rows, and non-matching rows.
- The NODENUMBER and PARTITION functions cannot be used in an outer join. These functions only work on rows in real tables.
- When the join is defined in the SELECT part of the query (see ON and WHERE Usage), it is done after any other joins and/or sub-queries specified in the FROM phrase. And it acts as if it is a left outer join.

### 11.3.11. Complex Joins

When one joins multiple tables using an outer join, one must consider carefully what exactly what one wants to do, because the answer that one gets will depend upon how one writes the query. To illustrate, the following query first gets a set of rows from the employee table, and then joins (from the employee table) to both the activity and photo tables:

*Join from Employee to Activity and Photo*

```
SELECT eee.empno
     , aaa.projno
     , aaa.actno
     , ppp.photo_format AS format
FROM employee eee
LEFT OUTER JOIN emp_act aaa
ON eee.empno = aaa.empno
AND aaa.emptime = 1
AND aaa.projno LIKE 'M%1%'
LEFT OUTER JOIN emp_photo ppp
ON eee.empno = ppp.empno
AND ppp.photo_format LIKE 'b%'
WHERE eee.lastname LIKE '%A%'
AND eee.empno <  '000170'
AND eee.empno <> '000030'
ORDER BY eee.empno;
```

*ANSWER*

| EMPN O | PROJNO | ACTNO | FORMAT |
|---|---|---|---|
| 000010 | MA2110 | 10 | - |
| 000070 | - | - | - |
| 000130 | - | - | bitmap |
| 000150 | MA2112 | 60 | bitmap |
| 000150 | MA2112 | 180 | bitmap |
| 000160 | MA2113 | 60 | - |

Observe that we got photo data, even when there was no activity data. This is because both tables were joined directly from the employee table. In the next query, we will again start at the employee table, then join to the activity table, and then from the activity table join to the photo table. We will not get any photo data, if the employee has no activity:

*Join from Employee to Activity, then from Activity to Photo*

```
SELECT eee.empno
     , aaa.projno
     , aaa.actno
     , ppp.photo_format AS format
FROM employee eee
LEFT OUTER JOIN emp_act aaa
ON eee.empno = aaa.empno
AND aaa.emptime = 1
AND aaa.projno LIKE 'M%1%'
LEFT OUTER JOIN emp_photo ppp
ON aaa.empno = ppp.empno
AND ppp.photo_format LIKE 'b%'
WHERE eee.lastname LIKE '%A%'
AND eee.empno < '000170'
AND eee.empno <> '000030'
ORDER BY eee.empno;
```

*ANSWER*

| EMPNO | PROJNO | ACTNO | FORMAT |
|---|---|---|---|
| 000010 | MA2110 | 10 | |
| 000070 | - | | |
| 000130 | | | |
| 000150 | MA2112 | 60 | bitmap |
| 000150 | MA2112 | 180 | bitmap |
| 000160 | MA2113 | 60 | - |

The only difference between the above two queries is the first line of the second ON.

### Outer Join followed by Inner Join

Mixing and matching inner and outer joins in the same query can cause one to get the wrong answer. To illustrate, the next query has an outer join, followed by an inner join. We are trying to do the following:

- Get a list of matching employees - based on some local predicates.
- For each employee found, list their matching activities, if any (i.e. left outer join).
- For each activity found, only list it if its project-name contains the letter "Q" (i.e. inner join between activity and project).

Below is the wrong way to write this query. It is wrong because the final inner join (between activity and project) turns the preceding outer join into an inner join. This causes an employee to not show when there are no matching projects:

*Complex join - wrong*

```
SELECT eee.workdept AS dp#
     , eee.empno
     , aaa.projno
     , ppp.prstaff AS staff
FROM
   (SELECT *
    FROM employee
    WHERE lastname LIKE '%A%'
    AND job <> 'DESIGNER'
    AND workdept BETWEEN 'B' AND 'E'
   ) AS eee
LEFT OUTER JOIN emp_act aaa
ON aaa.empno = eee.empno
AND aaa.emptime <= 0.5
INNER JOIN project ppp
ON aaa.projno = ppp.projno
AND ppp.projname LIKE '%Q%'
ORDER BY eee.workdept
       , eee.empno
       , aaa.projno;
```

| DP# | EMPNO | PROJNO | STAFF |
|---|---|---|---|
| C01 | 000030 | IF1000 | 2.00 |
| C01 | 000130 | IF1000 | 2.00 |

As was stated above, we really want to get all matching employees, and their related activities (projects). If an employee has no matching activates, we still want to see the employee.

The next query gets the correct answer by putting the inner join between the activity and project tables in parenthesis, and then doing an outer join to the combined result:

*Complex join - right*

```
SELECT eee.workdept AS dp#
     , eee.empno
     , xxx.projno
     , xxx.prstaff AS staff
FROM
    (SELECT *
     FROM employee
     WHERE lastname LIKE '%A%'
     AND job <> 'DESIGNER'
     AND workdept BETWEEN 'B' AND 'E'
    ) AS eee
LEFT OUTER JOIN
   (SELECT aaa.empno
         , aaa.emptime
         , aaa.projno
         , ppp.prstaff
    FROM emp_act aaa
    INNER JOIN project ppp
    ON aaa.projno = ppp.projno
    AND ppp.projname LIKE '%Q%'
   ) AS xxx
ON xxx.empno = eee.empno
AND xxx.emptime <= 0.5
ORDER BY eee.workdept
       , eee.empno
       , xxx.projno;
```

| DP# | EMPNO | PROJNO | STAFF |
|---|---|---|---|
| C01 | 000030 | IF1000 | 2.00 |
| C01 | 000130 | IF1000 | 2.00 |
| D21 | 000070 | | |
| D21 | 000240 | | |

The lesson to be learnt here is that if a subsequent inner join acts upon data in a preceding outer join, then it, in effect, turns the former into an inner join.

### 11.3.12. Simplified Nested Table Expression

The next query is the same as the prior, except that the nested-table expression has no select list, nor correlation name. In this example, any columns in tables that are inside of the nested table expression are referenced directly in the rest of the query:

*Complex join - right*

```
SELECT eee.workdept AS dp#
     , eee.empno
     , aaa.projno
     , ppp.prstaff AS staff
FROM
    (SELECT *
     FROM employee
     WHERE lastname LIKE '%A%'
     AND job <> 'DESIGNER'
     AND workdept BETWEEN 'B' AND 'E'
    ) AS eee
LEFT OUTER JOIN
    (SELECT aaa.empno
          , aaa.emptime
          , aaa.projno
          , ppp.prstaff
     FROM emp_act aaa
     INNER JOIN project ppp
     ON aaa.projno = ppp.projno
     AND ppp.projname LIKE '%Q%'
    ) AS xxx
ON xxx.empno = eee.empno
AND xxx.emptime <= 0.5
ORDER BY eee.workdept
       , eee.empno
       , xxx.projno;
```

| DP# | EMPNO | PROJNO | STAFF |
|---|---|---|---|
| C01 | 000030 | IF1000 | 2.00 |
| C01 | 000130 | IF1000 | 2.00 |
| D21 | 000070 | | |
| D21 | 000240 | | |

# Chapter 12. Sub-Query

Sub-queries are hard to use, tricky to tune, and often do some strange things. Consequently, a lot of people try to avoid them, but this is stupid because sub-queries are really, really, useful. Using a relational database and not writing sub-queries is almost as bad as not doing joins. A sub-query is a special type of fullselect that is used to relate one table to another without actually doing a join. For example, it lets one select all of the rows in one table where some related value exists, or does not exist, in another table.

## 12.1. Sample Tables

Two tables will be used in this section. Please note that the second sample table has a mixture of null and not-null values:

*Sample tables used in sub-query examples*

```
CREATE TABLE table1
( t1a CHAR(1) NOT NULL
, t1b CHAR(2) NOT NULL
, PRIMARY KEY(t1a));
COMMIT;

CREATE TABLE table2
( t2a CHAR(1) NOT NULL
, t2b CHAR(1) NOT NULL
, t2c CHAR(1));

INSERT INTO table1 VALUES ('A', 'AA')
                        , ('B','BB')
                        , ('C','CC');
INSERT INTO table2 VALUES ('A', 'A', 'A')
                        , ('B', 'A', NULL);
```

*TABLE1*

| T1A | T1B |
|---|---|
| A | AA |
| B | BB |
| C | CC |

*TABLE2*

| T2A | T2B | T2C |
|---|---|---|
| A | A | A |
| B | A | - |

```
"-" = null
```

A sub-query compares an expression against a fullselect. The type of comparison done is a function of which, if any, keyword is used.

The result of doing a sub-query check can be any one of the following:

- True, in which case the current row being processed is returned.
- False, in which case the current row being processed is rejected.
- Unknown, which is functionally equivalent to false.
- A SQL error, due to an invalid comparison.

## 12.2. No Keyword Sub-Query

One does not have to provide a SOME, or ANY, or IN, or any other keyword, when writing a sub-query. But if one does not, there are three possible results:

- If no row in the sub-query result matches, the answer is false.
- If one row in the sub-query result matches, the answer is true.
- If more than one row in the sub-query result matches, you get a SQL error.

In the example below, the T1A field in TABLE1 is checked to see if it equals the result of the sub-query (against T2A in TABLE2). For the value "A" there is a match, while for the values "B" and "C" there is no match:

*No keyword sub-query, works*

```
SELECT *
FROM table1
WHERE t1a =
   (SELECT t2a
    FROM table2
    WHERE t2a = 'A');
```

*ANSWER*

| T1A | T1B |
|---|---|
| A | AA |

*SUB-Q RESLT*

| T2A |
|---|
| A |

*TABLE1*

| T1A | T1B |
|---|---|
| A | AA |
| B | BB |
| C | CC |

*TABLE2*

| T2A | T2B | T2C |
|---|---|---|
| A | A | A |
| B | A | - |

```
"-" = null
```

The next example gets a SQL error. The sub-query returns two rows, which the "=l" check cannot process. Had an "= ANY" or an "= SOME" check been used instead, the query would have worked fine:

*No keyword sub-query, fails*

```
SELECT *
FROM table1
WHERE t1a =
    (SELECT t2a
     FROM table2);
```

*ANSWER* =⇒ **error**

*SUB-Q RESLT*

| T2A |
|---|
| A |
| B |

*TABLE1*

| T1A | T1B |
|---|---|
| A | AA |
| B | BB |
| C | CC |

*TABLE2*

| T2A | T2B | T2C |
|---|---|---|
| A | A | A |
| B | A | - |

```
"-" = null
```

There is almost never a valid reason for coding a sub-query that does not use an appropriate sub-query keyword. Do not do the above.

## 12.3. SOME/ANY Keyword Sub-Query

When a SOME or ANY sub-query check is used, there are two possible results:

- If any row in the sub-query result matches, the answer is true.
- If the sub-query result is empty, or all nulls, the answer is false.
- If no value found in the sub-query result matches, the answer is also false.
- The query below compares the current T1A value against the sub-query result three times.

The first row (i.e. T1A = "A") fails the test, while the next two rows pass:

*ANY sub-query*

```
SELECT *
FROM table1
WHERE t1a > ANY
 (SELECT t2a
  FROM table2);
```

*ANSWER*

| T1A | T1B |
|---|---|
| B | BB |
| C | CC |

*SUB-Q RESLT*

| T2A |
|---|
| A |
| B |

*TABLE1*

| T1A | T1B |
|---|---|
| A | AA |
| B | BB |
| C | CC |

*TABLE2*

| T2A | T2B | T2C |
|---|---|---|
| A | A | A |
| B | A | - |

```
"-" = null
```

When an ANY or ALL sub-query check is used with a "greater than" or similar expression (as opposed to an "equal" or a "not equal" expression) then the check can be considered similar to evaluating the MIN or the MAX of the sub-query result set. The following table shows what type of sub-query check equates to what type of column function: .ANY and ALL vs. column functions

| SUB-QUERY CHECK | EQUIVALENT COLUMN FUNCTION |
|---|---|
| > ANY(sub-query) | > MINIMUM(sub-query results) |
| < ANY(sub-query) | < MAXIMUM(sub-query results) |
| > ALL(sub-query) | > MAXIMUM(sub-query results) |
| < ALL(sub-query) | < MINIMUM(sub-query results) |

### 12.3.1. All Keyword Sub-Query

When an ALL sub-query check is used, there are two possible results:

- If all rows in the sub-query result match, the answer is true.
- If there are no rows in the sub-query result, the answer is also true.
- If any row in the sub-query result does not match, or is null, the answer is false.

Below is a typical example of the ALL check usage. Observe that a TABLE1 row is returned only if the current T1A value equals all of the rows in the sub-query result:

*ALL sub-query, with non-empty sub-query result*

```
SELECT *
FROM table1
WHERE t1a = ALL
    (SELECT t2b
     FROM table2
     WHERE t2b >= 'A');
```

*ANSWER*

| T1A | T1B |
|---|---|
| A | AA |

*SUB-Q RESLT*

| T2B |
|---|
| A |
| A |

When the sub-query result consists of zero rows (i.e. an empty set) then all rows processed in TABLE1 are deemed to match:

*ALL sub-query, with empty sub-query result*

```
SELECT *
FROM table1
WHERE t1a = ALL
    (SELECT t2b
     FROM table2
     WHERE t2b >= 'X');
```

*ANSWER*

| T1A | T1B |
|---|---|
| A | AA |
| B | BB |
| C | CC |

*SUB-Q RESLT*

| T2B |
|---|

The above may seem a little unintuitive, but it actually makes sense, and is in accordance with how the NOT EXISTS sub-query (see [sub.query]) handles a similar situation.

Imagine that one wanted to get a row from TABLE1 where the T1A value matched all of the sub-query result rows, but if the latter was an empty set (i.e. no rows), one wanted to get a non-match. Try this:

*ALL sub-query, with extra check for empty set*

```
SELECT *
FROM table1
WHERE t1a = ALL
    (SELECT t2b
     FROM table2
     WHERE t2b >= 'X')
AND 0 <>
    (SELECT COUNT(*)
     FROM table2
     WHERE t2b >= 'X');
```

*ANSWER* =⇒ 0 rows

*SQ-#1 RESLT*

| T2B |
|---|

*SQ-#2 RESLT*

| (*) |
|---|
| 0 |

*TABLE1*

| T1A | T1B |
|---|---|
| A | AA |
| B | BB |
| C | CC |

*TABLE2*

| T2A | T2B | T2C |
|---|---|---|
| A | A | A |
| B | A | - |

```
"-" = null
```

Two sub-queries are done above: The first looks to see if all matching values in the sub-query equal the current T1A value. The second confirms that the number of matching values in the sub-query is not zero.

Observe that the ANY sub-query check returns false when used against an empty set, while a similar ALL check returns true.

## 12.4. EXISTS Keyword Sub-Query

So far, we have been taking a value from the TABLE1 table and comparing it against one or more rows in the TABLE2 table. The EXISTS phrase does not compare values against rows, rather it simply looks for the existence or non-existence of rows in the sub-query result set:

- If the sub-query matches on one or more rows, the result is true.
- If the sub-query matches on no rows, the result is false.

Below is an EXISTS check that, given our sample data, always returns true:

*EXISTS sub-query, always returns a match*

```
SELECT *
FROM table1
WHERE EXISTS
    (SELECT *
     FROM table2);
```

*ANSWER*

| T1A | T1B |
|---|---|
| A | AA |
| B | BB |
| C | CC |

Below is an EXISTS check that, given our sample data, always returns false:

*EXISTS sub-query, always returns a non-match*

```
SELECT *
FROM table1
WHERE EXISTS
    (SELECT *
     FROM table2
     WHERE t2b >= 'X');
```

*ANSWER* =⇒ 0 rows

When using an EXISTS check, it doesn't matter what field, if any, is selected in the sub-query SELECT phrase. What is important is whether the sub-query returns a row or not. If it does, the sub-query returns true. Having said this, the next query is an example of an EXISTS subquery that will always return true, because even when no matching rows are found in the subquery, the SELECT COUNT(*) statement will return something (i.e. a zero). Arguably, this query is logically flawed:

*EXISTS sub-query, always returns a match*

```
SELECT *
FROM table1
WHERE EXISTS
    (SELECT COUNT(*)
     FROM table2
     WHERE t2b = 'X');
```

*ANSWER*

| T1A | T1B |
|---|---|
| A | AA |
| B | BB |
| C | CC |

## 12.5. NOT EXISTS Keyword Sub-query

The NOT EXISTS phrases looks for the non-existence of rows in the sub-query result set:

- If the sub-query matches on no rows, the result is true.
- If the sub-query has rows, the result is false.

We can use a NOT EXISTS check to create something similar to an ALL check, but with one very important difference. The two checks will handle nulls differently. To illustrate, consider the following two queries, both of which will return a row from TABLE1 only when it equals all of the matching rows in TABLE2:

```
SELECT *
FROM table1
WHERE NOT EXISTS
    (SELECT *
     FROM table2
     WHERE t2c >= 'A'
     AND t2c <> t1a);
```

*ANSWERS*

| T1A | T1B |
|---|---|
| A | AA |

*NOT EXISTS vs. ALL, ignore nulls, find match*

```
SELECT *
FROM table1
WHERE t1a = ALL
    (SELECT t2c
     FROM table2
     WHERE t2c >= 'A');
```

The above two queries are very similar. Both define a set of rows in TABLE2 where the T2C value is greater than or equal to "A", and then both look for matching TABLE2 rows that are not equal to the current T1A value. If a row is found, the sub-query is false. What happens when no TABLE2 rows match the ">=" predicate? As is shown below, both of our test queries treat an empty set as a match:

*NOT EXISTS vs. ALL, ignore nulls, no match*

```
-- NOT EXISTS
SELECT *
FROM table1
WHERE NOT EXISTS
    (SELECT *
     FROM table2
     WHERE t2c >= 'X'
     AND t2c <> t1a);

-- ALL
SELECT *
FROM table1
WHERE t1a = ALL
    (SELECT t2c
     FROM table2
     WHERE t2c >= 'X');
```

*ANSWERS*

| T1A | T1B |
|---|---|
| A | AA |
| B | BB |
| C | CC |

One might think that the above two queries are logically equivalent, but they are not. As is shown below, they return different results when the sub-query answer set can include nulls:

*NOT EXISTS vs. ALL, process nulls*

```
-- NOT EXISTS
SELECT *
FROM table1
WHERE NOT EXISTS
    (SELECT *
     FROM table2
     WHERE t2c <> t1a);

-- ALL
SELECT *
FROM table1
WHERE t1a = ALL
    (SELECT t2c
     FROM table2);
```

*ANSWER NOT EXISTS*

| T1A | T1B |
|---|---|
| A | AA |

*ANSWER_ALL* =⇒ no rows

A sub-query can only return true or false, but a Db2 field value can either match (i.e. be true), or not match (i.e. be false), or be unknown. It is the differing treatment of unknown values that is causing the above two queries to differ: * In the ALL sub-query, each value in T1A is checked against all of the values in T2C. The null value is checked, deemed to differ, and so the sub-query always returns false. * In the NOT EXISTS sub-query, each value in T1A is used to find those T2C values that are not equal. For the T1A values "B" and "C", the T2C value "A" does not equal, so the NOT EXISTS check will fail. But for the T1A value "A", there are no "not equal" values in T2C, because a null value does not "not equal" a literal. So the NOT EXISTS check will pass.

The following three queries list those T2C values that do "not equal" a given T1A value:

*List of values in T2C <> T1A value*

```
SELECT *              -- (a)
FROM table2
WHERE t2c <> 'A';

SELECT *              -- (b)
FROM table2
WHERE t2c <> 'B';

SELECT *              -- (c)
FROM table2
WHERE t2c <> 'C';
```

*ANSWER (a)* (no rows)

| T2A | T2B | T2C |
|---|---|---|

*ANSWER (b)*

| T2A | T2B | T2C |
|---|---|---|
| A | A | A |

*ANSWER (c)*

| T2A | T2B | T2C |
|---|---|---|
| A | A | A |

To make a NOT EXISTS sub-query that is logically equivalent to the ALL sub-query that we have used above, one can add an additional check for null T2C values:

*NOT EXISTS - same as ALL*

```
SELECT *
FROM table1
WHERE NOT EXISTS
    (SELECT *
     FROM table2
     WHERE t2c <> t1a
     OR t2c IS NULL);
```

*ANSWER* =⇒ no rows

One problem with the above query is that it is not exactly obvious. Another is that the two T2C predicates will have to be fenced in with parenthesis if other predicates (on TABLE2) exist. For these reasons, use an ALL sub-query when that is what you mean to do.

# 12.6. IN Keyword Sub-Query

The IN sub-query check is similar to the ANY and SOME checks:

- If any row in the sub-query result matches, the answer is true.
- If the sub-query result is empty, the answer is false.
- If no row in the sub-query result matches, the answer is also false.
- If all of the values in the sub-query result are null, the answer is false.

Below is an example that compares the T1A and T2A columns. Two rows match:

*IN sub-query example, two matches*

```
SELECT *
FROM table1
WHERE t1a IN
    (SELECT t2a
     FROM table2);
```

*ANSWER*

| T1A | T1B |
|---|---|
| A | AA |
| B | BB |

In the next example, no rows match because the sub-query result is an empty set:

*IN sub-query example, no matches*

```
SELECT *
FROM table1
WHERE t1a IN
    (SELECT t2a
     FROM table2
     WHERE t2a >= 'X');
```

*ANSWER* =⇒ 0 rows

The IN, ANY, SOME, and ALL checks all look for a match. Because one null value does not equal another null value, having a null expression in the "top" table causes the sub-query to always returns false:

*IN and = ANY sub-query examples, with nulls*

```
-- IN
SELECT *
FROM table2
WHERE t2c IN
    (SELECT t2c
     FROM table2);

-- = ANY
SELECT *
FROM table2
WHERE t2c = ANY
    (SELECT t2c
     FROM table2);
```

*ANSWERS*

| T2A | T2B | T2C |
|---|---|---|
| A | A | A |

## 12.7. NOT IN Keyword Sub-Queries

Sub-queries that look for the non-existence of a row work largely as one would expect, except when a null value in involved. To illustrate, consider the following query, where we want to see if the current T1A value is not in the set of T2C values:

*NOT IN sub-query example, no matches*

```
SELECT *
FROM table1
WHERE t1a NOT IN
    (SELECT t2c
     FROM table2);
```

*ANSWER* =⇒ 0 rows

Observe that the T1A values "B" and "C" are obviously not in T2C, yet they are not returned. The sub-query result set contains the value null, which causes the NOT IN check to return unknown, which equates to false. The next example removes the null values from the sub-query result, which then enables the NOT IN check to find the non-matching values:

*NOT IN sub-query example, matches*

```
SELECT *
FROM table1
WHERE t1a NOT IN
    (SELECT t2c
     FROM table2
     WHERE t2c IS NOT NULL);
```

*ANSWER*

| T1A | T1B |
|---|---|
| B | BB |
| C | CC |

Another way to find the non-matching values while ignoring any null rows in the sub-query, is to use an EXISTS check in a correlated sub-query:

*NOT EXISTS sub-query example, matches*

```
SELECT *
FROM table1
WHERE NOT EXISTS
    (SELECT *
     FROM table2
     WHERE t1a = t2c);
```

*ANSWER*

| T1A | T1B |
|---|---|
| B | BB |
| C | CC |

# 12.8. Correlated vs. Uncorrelated Sub-Queries

An uncorrelated sub-query is one where the predicates in the sub-query part of SQL statement have no direct relationship to the current row being processed in the "top" table (hence uncorrelated). The following sub-query is uncorrelated:

*Uncorrelated sub-query*

```
SELECT *
FROM table1
WHERE t1a IN
    (SELECT t2a
     FROM table2);
```

*ANSWER*

| T1A | T1B |
|---|---|
| A | AA |
| B | BB |

A correlated sub-query is one where the predicates in the sub-query part of the SQL statement cannot be resolved without reference to the row currently being processed in the "top" table (hence correlated). The following query is correlated:

*Correlated sub-query*

```
SELECT *
FROM table1
WHERE t1a IN
    (SELECT t2a
     FROM table2
     WHERE t1a = t2a);
```

*ANSWER*

| T1A | T1B |
|---|---|
| A | AA |
| B | BB |

Below is another correlated sub-query. Because the same table is being referred to twice, correlation names have to be used to delineate which column belongs to which table:

*Correlated sub-query, with correlation names*

```
SELECT *
FROM table2 aa
WHERE EXISTS
    (SELECT *
     FROM table2 bb
     WHERE aa.t2a = bb.t2b);
```

*ANSWER*

| T2A | T2B | T2C |
|---|---|---|
| A | A | A |

# 12.9. Which is Faster

In general, if there is a suitable index on the sub-query table, use a correlated sub-query. Else, use an uncorrelated sub-query. However, there are several very important exceptions to this rule, and some queries can only be written one way. NOTE: The Db2 optimizer is not as good at choosing the best access path for sub-queries as it is with joins. Be prepared to spend some time doing tuning.

# 12.10. Multi-Field Sub-Queries

Imagine that you want to compare multiple items in your sub-query. The following examples use an IN expression and a correlated EXISTS sub-query to do two equality checks:

*Multi-field sub-queries, equal checks*

```
SELECT *
FROM table1
WHERE (t1a,t1b) IN
    (SELECT t2a, t2b
     FROM table2);

SELECT *
FROM table1
WHERE EXISTS
    (SELECT *
    FROM table2
    WHERE t1a = t2a
    AND t1b = t2b);
```

*ANSWER* =⇒ 0 rows

Observe that to do a multiple-value IN check, you put the list of expressions to be compared in parenthesis, and then select the same number of items in the sub-query. An IN phrase is limited because it can only do an equality check. By contrast, use whatever predicates you want in an EXISTS correlated sub-query to do other types of comparison:

*Multi-field sub-query, with non-equal check*

```
SELECT *
FROM table1
WHERE EXISTS
    (SELECT *
     FROM table2
     WHERE t1a = t2a
     AND t1b >= t2b);
```

*ANSWER*

| T1A | T1B |
|---|---|
| A | AA |
| B | BB |

# 12.11. Nested Sub-Queries

Some business questions may require that the related SQL statement be written as a series of nested sub-queries. In the following example, we are after all employees in the EMPLOYEE table who have a salary that is greater than the maximum salary of all those other employees that do not work on a project with a name beginning 'MA'.

*Nested Sub-Queries*

```
SELECT empno
     , lastname
     , salary
FROM employee
WHERE salary >
    (SELECT MAX(salary)
     FROM employee
     WHERE empno NOT IN
         (SELECT empno
          FROM emp_act
          WHERE projno LIKE 'MA%'
         )
    )
ORDER BY 1;
```

*ANSWER*

| EMPNO | LASTNAME | SALARY |
|---|---|---|
| 000010 | HAAS | 52750.00 |
| 000110 | LUCCHESSI | 46500.00 |

# 12.12. Usage Examples

In this section we will use various sub-queries to compare our two test tables - looking for those rows where none, any, ten, or all values match.

## 12.12.1. Beware of Nulls

The presence of null values greatly complicates sub-query usage. Not allowing for them when they are present can cause one to get what is arguably a wrong answer. And do not assume that just because you don't have any nullable fields that you will never therefore encounter a null value. The DEPTNO table in the Department table is defined as not null, but in the following query, the maximum DEPTNO that is returned will be null:

*Getting a null value from a not null field*

```
SELECT COUNT(*)    AS #rows
     , MAX(deptno) AS maxdpt
FROM department
WHERE deptname LIKE 'Z%'
ORDER BY 1;
```

*ANSWER*

| #ROWS | MAXDEPT |
|---|---|
| 0 | null |

### 12.12.2. True if NONE Match

Find all rows in TABLE1 where there are no rows in TABLE2 that have a T2C value equal to the current T1A value in the TABLE1 table:

*Sub-queries, true if none match*

```
SELECT *
FROM table1 t1
WHERE 0 =
    (SELECT COUNT(*)
     FROM table2 t2
     WHERE t1.t1a = t2.t2c);

SELECT *
FROM table1 t1
WHERE NOT EXISTS
    (SELECT *
     FROM table2 t2
     WHERE t1.t1a = t2.t2c);

SELECT *
FROM table1
WHERE t1a NOT IN
    (SELECT t2c
     FROM table2
     WHERE t2c IS NOT NULL);
```

*ANSWER*

| T1A | T1B |
|---|---|
| B | BB |
| C | CC |

Observe that in the last statement above we eliminated the null rows from the sub-query. Had this not been done, the NOT IN check would have found them and then returned a result of "unknown" (i.e. false) for all of rows in the TABLE1A table.

### 12.12.3. Using a Join

Another way to answer the same problem is to use a left outer join, going from TABLE1 to TABLE2 while matching on the T1A and T2C fields. Get only those rows (from TABLE1) where the corresponding T2C value is null:

*Outer join, true if none match*

```
SELECT t1.*
FROM table1 t1
LEFT OUTER JOIN table2 t2
ON t1.t1a = t2.t2c
WHERE t2.t2c IS NULL;
```

*ANSWER*

| T1A | T1B |
|---|---|
| B | BB |
| C | CC |

### 12.12.4. True if ANY Match

Find all rows in TABLE1 where there are one, or more, rows in TABLE2 that have a T2C value equal to the current T1A value:

*Sub-queries, true if any match*

```
SELECT *
FROM table1 t1
WHERE EXISTS
    (SELECT *
     FROM table2 t2
     WHERE t1.t1a = t2.t2c);

SELECT *
FROM table1 t1
WHERE 1 <=
    (SELECT COUNT(*)
    FROM table2 t2
    WHERE t1.t1a = t2.t2c);

SELECT *
FROM table1
WHERE t1a = ANY
    (SELECT t2c
     FROM table2);
```

*ANSWER*

| T1A | T1B |
|---|---|
| A | AA |

```
SELECT *
FROM table1
WHERE t1a = SOME
    (SELECT t2c
     FROM table2);

SELECT *
FROM table1
WHERE t1a IN
    (SELECT t2c
     FROM table2);
```

Of all of the above queries, the second query is almost certainly the worst performer. All of the others can, and probably will, stop processing the sub-query as soon as it encounters a single matching value. But the sub-query in the second statement has to count all of the matching rows before it return either a true or false indicator.

### 12.12.5. Using a Join

This question can also be answered using an inner join. The trick is to make a list of distinct T2C values, and then join that list to TABLE1 using the T1A column. Several variations on this theme are given below:

*Joins, true if any match*

```
WITH t2 AS
(SELECT DISTINCT t2c
 FROM table2
)
SELECT t1.*
FROM table1 t1
          , t2
WHERE t1.t1a = t2.t2c;

SELECT t1.*
FROM table1 t1
          , (SELECT DISTINCT t2c
             FROM table2
            ) AS t2
WHERE t1.t1a = t2.t2c;
```

*ANSWER*

| T1A | T1B |
|---|---|
| A | AA |

```
SELECT t1.*
FROM table1 t1
INNER JOIN
    (SELECT DISTINCT t2c
     FROM table2
    )AS t2
ON t1.t1a = t2.t2c;
```

## 12.13. True if TEN Match

Find all rows in TABLE1 where there are exactly ten rows in TABLE2 that have a T2B value equal to the current T1A value in the TABLE1 table:

*Sub-queries, true if ten match (1 of 2)*

```
SELECT *
FROM table1 t1
WHERE 10 =
    (SELECT COUNT(*)
     FROM table2 t2
     WHERE t1.t1a = t2.t2b);

SELECT *
FROM table1
WHERE EXISTS
    (SELECT t2b
     FROM table2
     WHERE t1a = t2b
     GROUP BY t2b
     HAVING COUNT(*) = 10);

SELECT *
FROM table1
WHERE t1a IN
    (SELECT t2b
     FROM table2
     GROUP BY t2b
     HAVING COUNT(*) = 10);
```

*ANSWER* =⇒ 0 rows

The first two queries above use a correlated sub-query. The third is uncorrelated. The next query, which is also uncorrelated, is guaranteed to befuddle your coworkers. It uses a multifield IN (see Multi-Field Sub-Queries for more notes) to both check T2B and the count at the same time:

*Sub-queries, true if ten match (2 of 2)*

```
SELECT *
FROM table1
WHERE (t1a,10) IN
    (SELECT t2b
          , COUNT(*)
     FROM table2
     GROUP BY t2b);
```

*ANSWER* =⇒ 0 rows

### 12.13.1. Using a Join

To answer this generic question using a join, one simply builds a distinct list of T2B values that have ten rows, and then joins the result to TABLE1:

*Joins, true if ten match*

```
WITH t2 AS
(SELECT t2b
 FROM table2
 GROUP BY t2b
 HAVING COUNT(*) = 10
)
SELECT t1.*
FROM table1 t1
          , t2
WHERE t1.t1a = t2.t2b;
```

*ANSWER* =⇒ 0 rows

*Joins, true if ten match*

```
SELECT t1.*
FROM table1 t1
  , (SELECT t2b
     FROM table2
     GROUP BY t2b
     HAVING COUNT(*) = 10) AS t2
WHERE t1.t1a = t2.t2b;

SELECT t1.*
FROM table1 t1
INNER JOIN
    (SELECT t2b
     FROM table2
     GROUP BY t2b
     HAVING COUNT(*) = 10 )AS t2
ON t1.t1a = t2.t2b;
```

# 12.14. True if ALL match

Find all rows in TABLE1 where all matching rows in TABLE2 have a T2B value equal to the current T1A value in the TABLE1 table. Before we show some SQL, we need to decide what to do about nulls and empty sets:

- When nulls are found in the sub-query, we can either deem that their presence makes the relationship false, which is what Db2 does, or we can exclude nulls from our analysis.
- When there are no rows found in the sub-query, we can either say that the relationship is false, or we can do as Db2 does, and say that the relationship is true.

See All Keyword Sub-Query for a detailed discussion of the above issues.

The next two queries use the basic Db2 logic for dealing with empty sets; In other words, if no rows are found by the sub-query, then the relationship is deemed to be true. Likewise, the relationship is also true if all rows found by the sub-query equal the current T1A value:

*Sub-queries, true if all match, find rows*

```
SELECT *
FROM table1
WHERE t1a = ALL
    (SELECT t2b
     FROM table2);

SELECT *
FROM table1
WHERE NOT EXISTS
    (SELECT *
     FROM table2
     WHERE t1a <> t2b);
```

*ANSWER*

| T1A | T1B |
|---|---|
| A | AA |

The next two queries are the same as the prior, but an extra predicate has been included in the sub-query to make it return an empty set. Observe that now all TABLE1 rows match:

*Sub-queries, true if all match, empty set*

```
SELECT *
FROM table1
WHERE t1a = ALL
    (SELECT t2b
     FROM table2
     WHERE t2b >= 'X');
```

*ANSWER*

| T1A | T1B |
|---|---|
| A | AA |
| B | BB |
| C | CC |

*Sub-queries, true if all match, empty set*

```
SELECT *
FROM table1
WHERE NOT EXISTS
    (SELECT *
     FROM table2
     WHERE t1a <> t2b
     AND t2b >= 'X');
```

## 12.15. False if no Matching Rows

The next two queries differ from the above in how they address empty sets. The queries will return a row from TABLE1 if the current T1A value matches all of the T2B values found in the sub-query, but they will not return a row if no matching values are found:

*Sub-queries, true if all match, and at least one value found*

```
SELECT *
FROM table1
WHERE t1a = ALL
    (SELECT t2b
     FROM table2
     WHERE t2b >= 'X')
AND 0 <>
    (SELECT COUNT(*)
     FROM table2
     WHERE t2b >= 'X');

SELECT *
FROM table1
WHERE t1a IN
    (SELECT MAX(t2b)
     FROM table2
     WHERE t2b >= 'X'
     HAVING COUNT(DISTINCT t2b) = 1);
```

*ANSWER* =⇒ 0 rows

Both of the above statements have flaws: The first processes the TABLE2 table twice, which not only involves double work, but also requires that the sub-query predicates be duplicated. The second statement is just plain strange.

# Chapter 13. Union, Intersect, and Except

A UNION, EXCEPT, or INTERCEPT expression combines sets of columns into new sets of columns. An illustration of what each operation does with a given set of data is shown below:

*Examples of Union, Except, and Intersect*

```
        R1      R1      R1          R1          R1      R1      R1
        UNION   UNION   INTERSECT   INTERSECT   EXCEPT  EXCEPT  MINUS
        R2      ALL     R2          ALL         R2      ALL     R2
R1 R2           R2                  R2                  R2
-- --   -----   -----   ---------   -----       ------  ------  -----
A  A    A       A       A           A           E       A       E
A  A    B       A       B           A                   C
A  B    C       A       C           B                   C
B  B    D       A                   B                   E
B  B    E       A                   C
C  C            B
C  D            B
C               B
E               B
                B
                C
                C
                C
                C
                D
                E
```

Unlike the UNION and INTERSECT operations, the EXCEPT statement is not commutative. This means that "A EXCEPT B" is not the same as "B EXCEPT A".

*Sample Views*

*Query sample views*

```
CREATE VIEW R1 (R1)
AS VALUES ('A'), ('A'), ('A'), ('B'), ('B'), ('C'), ('C'), ('C'), ('E');

CREATE VIEW R2 (R2)
AS VALUES ('A'), ('A'), ('B'), ('B'), ('B'), ('C'), ('D');


SELECT R1
FROM R1
ORDER BY R1;

SELECT R2
FROM R2
ORDER BY R2;
```

*ANSWER*

| R1 | R2 |
|---|---|
| A | A |
| A | A |
| A | B |
| B | B |
| B | B |
| C | C |
| C | D |
| C | |
| E | |

**Usage Notes**

# 13.1. Union & Union All

A UNION operation combines two sets of columns and removes duplicates. The UNION ALL expression does the same but does not remove the duplicates. -Union and Union All SQL

```
SELECT R1
FROM R1
  UNION
SELECT R2
FROM R2
ORDER BY 1;

SELECT R1
FROM R1
  UNION ALL
SELECT R2
FROM R2
ORDER BY 1;
```

| R1 | R2 | UNION | UNION ALL |
|---|---|---|---|
| A | A | A | A |
| A | A | B | A |
| A | B | C | A |
| B | B | D | A |
| B | B | E | A |
| C | C | | B |
| C | D | | B |
| C | | | B |
| E | | | B |
| | | | B |
| | | | C |
| | | | C |
| | | | C |
| | | | C |
| | | | D |
| | | | E |

Recursive SQL requires that there be a UNION ALL phrase between the two main parts of the statement. The UNION ALL, unlike the UNION, allows for duplicate output rows which is what often comes out of recursive processing.

===Intersect & Intersect All

An INTERSECT operation retrieves the matching set of distinct values (not rows) from two columns. The INTERSECT ALL returns the set of matching individual rows.

*Intersect and Intersect All SQL*

```
SELECT R1
FROM R1
 INTERSECT
SELECT R2
FROM R2
ORDER BY 1;

SELECT R1
FROM R1
 INTERSECT ALL
SELECT R2
FROM R2
ORDER BY 1;
```

| R1 | R2 | INTERSECT | INTERSECT ALL |
|---|---|---|---|
| A | A | A | A |
| A | A | B | A |
| A | B | C | B |
| B | B | | B |
| B | B | | C |
| C | C | | |
| C | D | | |
| C | | | |
| E | | | |

An INTERSECT and/or EXCEPT operation is done by matching ALL of the columns in the top and bottom result-sets. In other words, these are row, not column, operations. It is not possible to only match on the keys, yet at the same time, also fetch non-key columns. To do this, one needs to use a sub-query.

## 13.2. Except, Except All & Minus

An EXCEPT operation retrieves the set of distinct data values (not rows) that exist in the first the table but not in the second. The EXCEPT ALL returns the set of individual rows that exist only in the first table. The word MINUS is a synonym for EXCEPT.

*Except and Except All SQL (R1 on top)*

```
SELECT R1
FROM R1
 EXCEPT
SELECT R2
FROM R2
ORDER BY 1;

SELECT R1
FROM R1
 EXCEPT ALL
SELECT R2
FROM R2
ORDER BY 1;
```

| R1 | R2 | R1 EXCEPT R2 | R1 EXCEPT ALL R2 |
|---|---|---|---|
| A | A | E | A |
| A | A | | C |
| A | B | | C |
| B | B | | E |
| B | B | | |
| C | C | | |
| C | D | | |
| C | | | |
| E | | | |

Because the EXCEPT/MINUS operation is not commutative, using it in the reverse direction (i.e. R2 to R1 instead of R1 to R2) will give a different result:

*Except and Except All SQL (R2 on top)*

```
SELECT R2
FROM R2
 EXCEPT
SELECT R1
FROM R1
ORDER BY 1;

SELECT R2
FROM R2
 EXCEPT ALL
SELECT R1
FROM R1
ORDER BY 1;
```

| R1 | R2 | R2 EXCEPT R1 | R2 EXCEPT ALL R1 |
|---|---|---|---|
| A | A | D | B |
| A | A | | D |
| A | B | | |
| B | B | | |
| B | B | | |
| C | C | | |
| C | D | | |
| C | | | |
| E | | | |

Only the EXCEPT/MINUS operation is not commutative. Both the UNION and the INTERSECT operations work the same regardless of which table is on top or on bottom.

### 13.2.1. Precedence Rules

When multiple operations are done in the same SQL statement, there are precedence rules:

- Operations in parenthesis are done first.
- INTERSECT operations are done before either UNION or EXCEPT.
- Operations of equal worth are done from top to bottom.

The next example illustrates how parenthesis can be used change the processing order:

```
SELECT R1
FROM R1
 UNION
SELECT R2
FROM R2
 EXCEPT
SELECT R2
FROM R2
ORDER BY 1;
```

*ANSWER*

| E |
|---|

*Use of parenthesis in Union*

```
(SELECT R1
 FROM R1
  UNION
 SELECT R2
 FROM R2
 )
  EXCEPT
 SELECT R2
 FROM R2
 ORDER BY 1;
```

*ANSWER*

| **E** |
|---|

*Use of parenthesis in Union*

```
SELECT R1
FROM R1
 UNION
(SELECT R2
 FROM R2
  EXCEPT
 SELECT R2
 FROM R2
)
ORDER BY 1;
```

*ANSWER*

| **A** |
|---|
| B |
| C |
| E |

### 13.2.2. Unions and Views

Imagine that one has a series of tables that track sales data, with one table for each year. One can define a view that is the UNION ALL of these tables, so that a user would see them as a single object. Such a view can support inserts, updates, and deletes, as long as each table in the view has a constraint that distinguishes it from all the others. Below is an example:

*Define view to combine yearly tables*

```
CREATE TABLE sales_data_2002
( sales_date   DATE       NOT NULL
, daily_seq#   INTEGER    NOT NULL
, cust_id      INTEGER    NOT NULL
, amount       DEC(10, 2) NOT NULL
, invoice#     INTEGER    NOT NULL
, sales_rep    CHAR(10)   NOT NULL
, CONSTRAINT C CHECK (YEAR(sales_date) = 2002)
, PRIMARY KEY (sales_date, daily_seq#)
);

CREATE TABLE sales_data_2003
( sales_date   DATE      NOT NULL
, daily_seq#   INTEGER   NOT NULL
, cust_id      INTEGER   NOT NULL
, amount       DEC(10,2) NOT NULL
, invoice#     INTEGER   NOT NULL
, sales_rep    CHAR(10)  NOT NULL
, CONSTRAINT C CHECK (YEAR(sales_date) = 2003)
, PRIMARY KEY (sales_date, daily_seq#));

CREATE VIEW sales_data AS
SELECT *
FROM sales_data_2002
 UNION ALL
SELECT *
FROM sales_data_2003;
```

Below is some SQL that changes the contents of the above view:

*Insert, update, and delete using view*

```
INSERT INTO sales_data VALUES
  ('2002-11-22',1,123,100.10,996,'SUE')
, ('2002-11-22',2,123,100.10,997,'JOHN')
, ('2003-01-01',1,123,100.10,998,'FRED')
, ('2003-01-01',2,123,100.10,999,'FRED');

UPDATE sales_data
SET amount = amount / 2
WHERE sales_rep = 'JOHN';

DELETE
FROM sales_data
WHERE sales_date = '2003-01-01'
AND daily_seq# = 2;
```

Below is the view contents, after the above is run: .View contents after insert, update, delete

| SALES_DATE | DAILY_SEQ# | CUST_ID | AMOUNT | INVOICE# | SALES_REP |
|---|---|---|---|---|---|
| 01/01/2003 | 1 | 123 | 100.10 | 998 | FRED |
| 11/22/2002 | 1 | 123 | 100.10 | 996 | SUE |
| 11/22/2002 | 2 | 123 | 50.05 | 997 | JOHN |

# Chapter 14. Materialized Query Tables

## 14.1. Introduction

A materialized query table contains the results of a query. The Db2 optimizer knows this and can, if appropriate, redirect a query that is against the source table(s) to use the materialized query table instead. This can make the query run much faster. The following statement defines a materialized query table:

*Sample materialized query table DDL*

```
CREATE TABLE staff_summary AS
(SELECT dept
      , COUNT(*) AS count_rows
      , SUM(id) AS sum_id
 FROM staff
 GROUP BY dept)
 DATA INITIALLY DEFERRED REFRESH IMMEDIATE;
```

Below on the left is a query that is very similar to the one used in the above CREATE. The Db2 optimizer can convert this query into the optimized equivalent on the right, which uses the materialized query table. Because (in this case) the data in the materialized query table is maintained in sync with the source table, both statements will return the same answer.

ORIGINAL QUERY:

*Original and optimized queries*

```
SELECT dept
     , AVG(id)
FROM staff
GROUP BY dept

-- OPTIMIZED QUERY:

SELECT Q1.dept AS "dept"
     , Q1.sum_id / Q1.count_rows
FROM staff_summary AS Q1
```

When used appropriately, materialized query tables can cause dramatic improvements in query performance. For example, if in the above STAFF table there was, on average, about 5,000 rows per individual department, referencing the STAFF_SUMMARY table instead of the STAFF table in the sample query might be about 1,000 times faster.

## 14.2. Db2 Optimizer Issues

In order for a materialized query table to be considered for use by the Db2 optimizer, the following

has to be true:

- The table has to be refreshed at least once.
- The table MAINTAINED BY parameter and the related Db2 special registers must correspond. For example, if the table is USER maintained, then the CURRENT REFRESH AGE special register must be set to ANY, and the CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION special register must be set to USER or ALL. See Optimizer Options for more details on these registers.

### 14.2.1. Usage Notes

A materialized query table is defined using a variation of the standard CREATE TABLE statement. Instead of providing an element list, one supplies a SELECT statement, and defines the refresh option. ===Syntax Options ====Refresh

**REFRESH DEFERRED**: The data is refreshed whenever one does a REFRESH TABLE. At this point, Db2 will first delete all of the existing rows in the table, then run the select statement defined in the CREATE to (you guessed it) repopulate.

**REFRESH IMMEDIATE**: Once created, this type of table has to be refreshed once using the REFRESH statement. From then on, Db2 will maintain the materialized query table in sync with the source table as changes are made to the latter.

Materialized query tables that are defined REFRESH IMMEDIATE are obviously more useful in that the data in them is always current. But they may cost quite a bit to maintain, and not all queries can be defined thus.

#### Query Optimization

**ENABLE**: The table is used for query optimization when appropriate. This is the default. The table can also be queried directly.

**DISABLE**: The table will not be used for query optimization. It can be queried directly.

#### Maintained By

**SYSTEM**: The data in the materialized query table is maintained by the system. This is the default.

**USER**: The user is allowed to perform insert, update, and delete operations against the materialized query table. The table cannot be refreshed. This type of table can be used when you want to maintain your own materialized query table (e.g. using triggers) to support features not provided by Db2. The table can also be defined to enable query optimization, but the optimizer will probably never use it as a substitute for a real table.

**FEDERATED_TOOL**: The data in the materialized query table is maintained by the replication tool. Only a REFRESH DEFERRED table can be maintained using this option.

### 14.2.2. Options vs. Actions

The following table compares materialized query table options to subsequent actions:

*Materialized query table options vs. allowable actions*

```
MATERIALIZED QUERY TABLE                ALLOWABLE ACTIONS ON TABLE
REFRESH     MAINTAINED BY               REFRESH TABLE   INSERT/UPDATE/DELETE
---------   -------------               -------------   --------------------
DEFERRED    SYSTEM                      yes             no
            USER                        no              yes
IMMEDIATE   SYSTEM                      yes             no
```

# 14.3. Select Statement

Various restrictions apply to the select statement that is used to define the materialized query table. In general, materialized query tables defined refresh-immediate need simpler queries than those defined refresh-deferred.

**Refresh Deferred Tables**

- The query must be a valid SELECT statement.
- Every column selected must have a name.
- An ORDER BY is not allowed.
- Reference to a typed table or typed view is not allowed.
- Reference to declared temporary table is not allowed.
- Reference to a nickname or materialized query table is not allowed.
- Reference to a system catalogue table is not allowed. Reference to an explain table is allowed, but is impudent.
- Reference to NODENUMBER, PARTITION, or any other function that depends on physical characteristics, is not allowed.
- Reference to a datalink type is not allowed.
- Functions that have an external action are not allowed.
- Scalar functions, or functions written in SQL, are not allowed. So SUM(SALARY) is fine, but SUM(INT(SALARY)) is not allowed.

===Refresh Immediate Tables

All of the above restrictions apply, plus the following:

- If the query references more than one table or view, it must define as inner join, yet not use the INNER JOIN syntax (i.e. must use old style).
- If there is a GROUP BY, the SELECT list must have a COUNT*() or COUNT_BIG()* column.
- Besides the COUNT and COUNT_BIG, the only other column functions supported are SUM and GROUPING - all with the DISTINCT phrase. Any field that allows nulls, and that is summed, but also have a COUNT(column name) function defined.
- Any field in the GROUP BY list must be in the SELECT list.

- The table must have at least one unique index defined, and the SELECT list must include (amongst other things) all the columns of this index.
- Grouping sets, CUBE an ROLLUP are allowed. The GROUP BY items and associated GROUPING column functions in the select list must for a unique key of the result set.
- The HAVING clause is not allowed.
- The DISTINCT clause is not allowed.
- Non-deterministic functions are not allowed.
- Special registers are not allowed.
- If REPLICATED is specified, the table must have a unique key.

## 14.4. Optimizer Options

A materialized query table that has been defined ENABLE QUERY OPTIMIZATION, and has been refreshed, is a candidate for use by the Db2 optimizer if, and only if, three Db2 special registers are set to match the table status:

- CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION.
- CURRENT QUERY OPTIMIZATION.
- CURRENT REFRESH AGE.

Each of the above are discussed below.

**CURRENT REFRESH AGE**

The refresh age special register tells the Db2 optimizer how up-to-date the data in an materialized query table has to be in order to be considered. There are only two possible values:

- 0: Only use those materialized query tables that are defined as refresh-immediate are eligible. This is the default.
- 99,999,999,999,999: Consider all valid materialized query tables. This is the same as ANY.

The above number is a 26-digit decimal value that is a timestamp duration, but without the microsecond component. The value ANY is logically equivalent.

The database default value can be changed using the following command:

*Changing default refresh age for database*

```
UPDATE DATABASE CONFIGURATION USING dft_refresh_age ANY;
```

The database default value can be overridden within a thread using the SET REFRESH AGE statement.

Below are some examples of the SET command:

*Set refresh age command, examples*

```
SET CURRENT REFRESH AGE 0;
SET CURRENT REFRESH AGE = ANY;
SET CURRENT REFRESH AGE = 99999999999999;
```

**CURRENT MAINTAINED TYPES**

The current maintained types special register tells the Db2 optimizer what types of materialized query table that are defined refresh deferred are to be considered - assuming that the refresh-age parameter is not set to zero:

- **ALL**: All refresh-deferred materialized query tables are to be considered. If this option is chosen, no other option can be used.
- **NONE**: No refresh-deferred materialized query tables are to be considered. If this option is chosen, no other option can be used.
- **SYSTEM**: System-maintained refresh-deferred materialized query tables are to be considered. This is the default.
- **USER**: User-maintained refresh-deferred materialized query tables are to be considered.
- **FEDERATED TOOL**: Federated-tool-maintained refresh-deferred materialized query tables are to be considered, but only if the CURRENT QUERY OPTIMIZATION special register is 2 or greater than 5.
- **CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION**: The existing values for this special register are used.

The database default value can be changed using the following command:

*Changing default maintained type for database*

```
UPDATE DATABASE CONFIGURATION USING dft_refresh_age ANY;
```

The database default value can be overridden within a thread using the SET REFRESH AGE statement. Below are some examples of the SET command:

*Set maintained type command, examples*

```
SET CURRENT MAINTAINED TYPES = ALL;
SET CURRENT MAINTAINED TABLE TYPES = SYSTEM;
SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION = USER, SYSTEM;
```

**CURRENT QUERY OPTIMIZATION**

The current query optimization special register tells the Db2 optimizer what set of optimization techniques to use. The value can range from zero to nine - except for four or eight. A value of five or above will cause the optimizer to consider using materialized query tables.

The database default value can be changed using the following command:

*Changing default maintained type for database*

```
UPDATE DATABASE CONFIGURATION USING DFT_QUERYOPT 5;
```

The database default value can be overridden within a thread using the SET CURRENT QUERY OPTIMIZATION statement.

Below are an example of the SET command:

*Set query optimization, example*

```
SET CURRENT QUERY OPTIMIZATION = 9;
```

### 14.4.1. What Matches What

Assuming that the current query optimization special register is set to five or above, the Db2 optimizer will consider using a materialized query table (instead of the base table) when any of the following conditions are true:

*When Db2 will consider using a materialized query table*

```
       MQT DEFINITION         DATABASE/APPLICATION STATUS        DB2
========================== ===================================   USE
REFRESH    MAINTAINED-BY   REFRESH-AGE  MAINTAINED-TYPE          MQT
========= ==============   =========== =====================    ===
IMMEDIATE  SYSTEM             -               -                 Yes
DEFERRED   SYSETM           ANY           ALL or SYSTEM         Yes
DEFERRED   USER             ANY           ALL or USER           Yes
DEFERRED   FEDERATED-TOOL    ANY           ALL or FEDERATED-TOOL Yes
```

### 14.4.2. Selecting Special Registers

One can select the relevant special register to see what the values are:

*Selecting special registers*

```
SELECT CURRENT REFRESH AGE        AS age_ts
     , CURRENT TIMESTAMP          AS current_ts
     , CURRENT QUERY OPTIMIZATION AS q_opt
FROM sysibm.sysdummy1;
```

### 14.4.3. Refresh Deferred Tables

A materialized query table defined REFRESH DEFERRED can be periodically updated using the REFRESH TABLE command. Below is an example of a such a table that has one row per qualifying department in the STAFF table:

*Refresh deferred materialized query table DD*

```
CREATE TABLE staff_names AS
(SELECT dept
      , COUNT(*)          AS count_rows
      , SUM(salary)       AS sum_salary
      , AVG(salary)       AS avg_salary
      , MAX(salary)       AS max_salary
      , MIN(salary)       AS min_salary
      , STDDEV(salary)    AS std_salary
      , VARIANCE(salary)  AS var_salary
      , CURRENT TIMESTAMP AS last_change
FROM staff
WHERE TRANSLATE(name) LIKE '%A%'
AND salary > 10000
GROUP BY dept
HAVING COUNT(*) = 1
) DATA INITIALLY DEFERRED REFRESH DEFERRED;
```

### 14.4.4. Refresh Immediate Tables

A materialized query table defined REFRESH IMMEDIATE is automatically maintained in sync with the source table by Db2. As with any materialized query table, it is defined by referring to a query. Below is a table that refers to a single source table:

*Refresh immediate materialized query table DDL*

```
CREATE TABLE emp_summary AS
(SELECT emp.workdept
      , COUNT(*) AS num_rows
      , COUNT(emp.salary) AS num_salary
      , SUM(emp.salary) AS sum_salary
      , COUNT(emp.comm) AS num_comm
      , SUM(emp.comm) AS sum_comm
 FROM employee emp
 GROUP BY emp.workdept
) DATA INITIALLY DEFERRED REFRESH IMMEDIATE;
```

Below is a query that can use the above materialized query table in place of the base table:

*Query that uses materialized query table (1 of 3)*

```
SELECT emp.workdept
     , DEC(SUM(emp.salary),8,2)  AS sum_sal
     , DEC(AVG(emp.salary),7,2)  AS avg_sal
     , SMALLINT(COUNT(emp.comm)) AS #comms
     , SMALLINT(COUNT(*))        AS #emps
FROM employee emp
WHERE emp.workdept > 'C'
GROUP BY emp.workdept
HAVING COUNT(*) <> 5
AND SUM(emp.salary) > 50000
ORDER BY sum_sal DESC;
```

The next query can also use the materialized query table. This time, the data returned from the materialized query table is qualified by checking against a sub-query:

*Query that uses materialized query table (2 of 3)*

```
SELECT emp.workdept
     , COUNT(*) AS #rows
FROM employee emp
WHERE emp.workdept IN
    (SELECT deptno
     FROM department
     WHERE deptname LIKE '%S%'
    )
GROUP BY emp.workdept
HAVING SUM(salary) > 50000;
```

This last example uses the materialized query table in a nested table expression:

*Query that uses materialized query table (3 of 3)*

```
SELECT #emps
     , DEC(SUM(sum_sal), 9, 2) AS sal_sal
     , SMALLINT(COUNT(*))      AS #depts
FROM
    (SELECT emp.workdept
          , DEC(SUM(emp.salary), 8, 2) AS sum_sal
          , MAX(emp.salary)            AS max_sal
          , SMALLINT(COUNT(*))         AS #emps
     FROM employee emp
     GROUP BY emp.workdept
    ) AS XXX
GROUP BY #emps
HAVING COUNT(*) > 1
ORDER BY #emps
FETCH FIRST 3 ROWS ONLY
OPTIMIZE FOR 3 ROWS;
```

### 14.4.5. Using Materialized Query Tables to Duplicate Data

All of the above materialized query tables have contained a GROUP BY in their definition. But this is not necessary. To illustrate, we will first create a simple table:

*Create source table*

```
CREATE TABLE staff_all
( id      SMALLINT      NOT NULL
, name    VARCHAR(9)    NOT NULL
, job     CHAR(5)
, salary  DECIMAL(7, 2)
, PRIMARY KEY(id));
```

As long as the above table has a primary key, which it does, we can define a duplicate of the above using the following code:

*Create duplicate data table*

```
CREATE TABLE staff_all_dup AS
(SELECT *
 FROM staff_all)
DATA INITIALLY DEFERRED REFRESH IMMEDIATE;
```

We can also decide to duplicate only certain rows:

*Create table - duplicate certain rows only*

```
CREATE TABLE staff_all_dup_some AS
(SELECT *
 FROM staff_all
 WHERE id < 30)
 DATA INITIALLY DEFERRED REFRESH IMMEDIATE;
```

Imagine that we had another table that listed all those staff that we are about to fire:

*Create source table*

```
CREATE TABLE staff_to_fire
( id SMALLINT     NOT NULL
, name VARCHAR(9) NOT NULL
, dept SMALLINT
, PRIMARY KEY(id));
```

We can create materialized query table that joins the above two staff tables as long as the following is true:

- Both tables have identical primary keys (i.e. same number of columns).
- The join is an inner join on the common primary key fields.
- All primary key columns are listed in the SELECT.

Now for an example:

*Materialized query table on join*

```
CREATE TABLE staff_combo AS
(SELECT aaa.id    AS id1
      , aaa.job   AS job
      , fff.id    AS id2
      , fff.dept  AS dept
 FROM staff_all     aaa
    , staff_to_fire fff
 WHERE aaa.id = fff.id)
DATA INITIALLY DEFERRED REFRESH IMMEDIATE;
```

See Multi-table Materialized Query Tables for more examples of join usage.

### 14.4.6. Queries that don't use Materialized Query Table

Below is a query that can not use the EMP_SUMMARY table because of the reference to the MAX function. Ironically, this query is exactly the same as the nested table expression above, but in the prior example the MAX is ignored because it is never actually selected:

*Query that doesn't use materialized query table (1 of 2)*

```
SELECT emp.workdept
     , DEC(SUM(emp.salary), 8, 2) AS sum_sal
     , MAX(emp.salary)            AS max_sal
FROM employee emp
GROUP BY emp.workdept;
```

The following query can't use the materialized query table because of the DISTINCT clause:

*Query that doesn't use materialized query table (2 of 2)*

```
SELECT emp.workdept
     , DEC(SUM(emp.salary), 8, 2) AS sum_sal
     , COUNT(DISTINCT salary) AS #salaries
FROM employee emp
GROUP BY emp.workdept;
```

### 14.4.7. Usage Notes and Restrictions

- A materialized query table must be refreshed before it can be queried. If the table is defined refresh immediate, then the table will be maintained automatically after the initial refresh.
- Make sure to commit after doing a refresh. The refresh does not have an implied commit.
- Run RUNSTATS after refreshing a materialized query table.
- One can not load data into materialized query tables.
- One can not directly update materialized query tables.

To refresh a materialized query table, use either of the following commands:

*Materialized query table refresh commands*

```
REFRESH TABLE emp_summary;
COMMIT;
SET INTEGRITY FOR emp_summary iMMEDIATE CHECKED;
COMMIT;
```

### 14.4.8. Multi-table Materialized Query Tables

Single-table materialized query tables save having to look at individual rows to resolve a GROUP BY. Multi-table materialized query tables do this, and also avoid having to resolve a join.

*Multi-table materialized query table DDL*

```
CREATE TABLE dept_emp_summary AS
(SELECT emp.workdept
      , dpt.deptname
      , COUNT(*) AS num_rows
      , COUNT(emp.salary) AS num_salary
      , SUM(emp.salary) AS sum_salary
      , COUNT(emp.comm) AS num_comm
      , SUM(emp.comm) AS sum_comm
 FROM employee   emp
    , department dpt
 WHERE dpt.deptno = emp.workdept
 GROUP BY emp.workdept
        , dpt.deptname
) DATA INITIALLY DEFERRED REFRESH IMMEDIATE;
```

The following query is resolved using the above materialized query table:

*Query that uses materialized query table*

```
SELECT d.deptname
, d.deptno
, DEC(AVG(e.salary), 7, 2) AS avg_sal
, SMALLINT(COUNT(*)) AS #emps
FROM department d
   , employee e
WHERE e.workdept = d.deptno
AND d.deptname LIKE '%S%'
GROUP BY d.deptname
       , d.deptno
HAVING SUM(e.comm) > 4000
ORDER BY avg_sal DESC;
```

Here is the SQL that Db2 generated internally to get the answer:

*Db2 generated query to use materialized query table*

```
SELECT Q2.$C0 AS "deptname"
     , Q2.$C1 AS "deptno"
     , Q2.$C2 AS "avg_sal"
     , Q2.$C3 AS "#emps"
FROM
    (SELECT Q1.deptname                                AS $C0
          , Q1.workdept                                AS $C1
          , DEC((Q1.sum_salary / Q1.num_salary),7,2) AS $C2
          , SMALLINT(Q1.num_rows)                      AS $C3
     FROM dept_emp_summary                             AS Q1
     WHERE (Q1.deptname LIKE '%S%')
     AND (4000 < Q1.sum_comm)
    ) AS Q2
ORDER BY Q2.$C2 DESC;
```

### 14.4.9. Rules and Restrictions

- The join must be an inner join, and it must be written in the old style syntax.
- Every table accessed in the join (except one?) must have a unique index.
- The join must not be a Cartesian product.
- The GROUP BY must include all of the fields that define the unique key for every table (except one?) in the join.

### 14.4.10. Three-table Example

*Three-table materialized query table DDL*

```
CREATE TABLE dpt_emp_act_sumry AS
(SELECT emp.workdept
      , dpt.deptname
      , emp.empno
      , emp.firstnme
      , SUM(act.emptime) AS sum_time
      , COUNT(act.emptime) AS num_time
      , COUNT(*) AS num_rows
 FROM department dpt
    , employee   emp
    , emp_act    act
 WHERE dpt.deptno = emp.workdept
 AND emp.empno = act.empno
 GROUP BY emp.workdept
        , dpt.deptname
        , emp.empno
        , emp.firstnme
) DATA INITIALLY DEFERRED REFRESH IMMEDIATE;
```

Now for a query that will use the above:

*Query that uses materialized query table*

```
SELECT d.deptno
     , d.deptname
     , DEC(AVG(a.emptime),5,2) AS avg_time
FROM department d
   , employee   e
   , emp_act    a
WHERE d.deptno = e.workdept
AND   e.empno  = a.empno
AND d.deptname LIKE '%S%'
AND e.firstnme LIKE '%S%'
GROUP BY d.deptno
       , d.deptname
ORDER BY 3 DESC;
```

And here is the Db2 generated SQL:

*Db2 generated query to use materialized query table*

```
SELECT Q4.$C0 AS "deptno"
     , Q4.$C1 AS "deptname"
     , Q4.$C2 AS "avg_time"
FROM
    (SELECT Q3.$C3 AS $C0
          , Q3.$C2 AS $C1
          , DEC((Q3.$C1 / Q3.$C0),5,2) AS $C2
     FROM (SELECT SUM(Q2.$C2) AS $C0
                , SUM(Q2.$C3) AS $C1
                , Q2.$C0 AS $C2
                , Q2.$C1 AS $C3
           FROM (SELECT Q1.deptname AS $C0
                      , Q1.workdept AS $C1
                      , Q1.num_time AS $C2
                      , Q1.sum_time AS $C3
                 FROM dpt_emp_act_sumry AS Q1
                 WHERE (Q1.firstnme LIKE '%S%')
                 AND (Q1.DEPTNAME LIKE '%S%')
                ) AS Q2
           GROUP BY Q2.$C1
                  , Q2.$C0
          ) AS Q3
    ) AS Q4
ORDER BY Q4.$C2 DESC;
```

### 14.4.11. Indexes on Materialized Query Tables

To really make things fly, one can add indexes to the materialized query table columns. Db2 will

then use these indexes to locate the required data. Certain restrictions apply:

- Unique indexes are not allowed.
- The materialized query table must not be in a "check pending" status when the index is defined. Run a refresh to address this problem.

Below are some indexes for the DPT_EMP_ACT_SUMRY table that was defined above:

*Indexes for DPT_EMP_ACT_SUMRY materialized query table*

```
CREATE INDEX dpt_emp_act_sumx1
ON dpt_emp_act_sumry ( workdept
                     , deptname
                     , empno
                     , firstnme);

CREATE INDEX dpt_emp_act_sumx2
ON dpt_emp_act_sumry (num_rows);
```

The next query will use the first index (i.e. on WORKDEPT):

*Sample query that use WORKDEPT index*

```
SELECT d.deptno
     , d.deptname
     , e.empno
     , e.firstnme
     , INT(AVG(a.emptime)) AS avg_time
FROM department d
   , employee   e
   , emp_act    a
WHERE d.deptno =    e.workdept
AND e.empno    =    a.empno
AND d.deptno   LIKE 'D%'
GROUP BY d.deptno
       , d.deptname
       , e.empno
       , e.firstnme
ORDER BY 1, 2, 3, 4;
```

The next query will use the second index (i.e. on NUM_ROWS):

*Sample query that uses NUM_ROWS index*

```
SELECT d.deptno
     , d.deptname
     , e.empno
     , e.firstnme
     , COUNT(*) AS #acts
FROM department d
   , employee   e
   , emp_act    a
WHERE d.deptno = e.workdept
AND e.empno = a.empno
GROUP BY d.deptno
       , d.deptname
       , e.empno
       , e.firstnme
HAVING COUNT(*) > 4
ORDER BY 1, 2, 3, 4;
```

## 14.5. Organizing by Dimensions

The following materialized query table is organized (clustered) by the two columns that are referred to in the GROUP BY. Under the covers, Db2 will also create a dimension index on each column, and a block index on both columns combined:

*Materialized query table organized by dimensions*

```
CREATE TABLE emp_sum AS
(SELECT workdept
      , job
      , SUM(salary)        AS sum_sal
      , COUNT(*)           AS #emps
      , GROUPING(workdept) AS grp_dpt
      , GROUPING(job)      AS grp_job
 FROM employee
 GROUP BY CUBE( workdept
              , job)
)
DATA INITIALLY DEFERRED REFRESH DEFERRED
ORGANIZE BY DIMENSIONS (workdept, job)
IN tsempsum;
```

Multi-dimensional tables may perform very poorly when created in the default tablespace, or in a system-maintained tablespace. Use a database-maintained tablespace with the right extent size, and/or run the Db2EMPFA command.

Don’t forget to run RUNSTATS!

### 14.5.1. Using Staging Tables

A staging table can be used to incrementally maintain a materialized query table that has been defined refresh deferred. Using a staging table can result in a significant performance saving (during the refresh) if the source table is very large, and is not changed very often.

To use a staging table, the SQL statement used to define the target materialized query table must follow the rules that apply for a table that is defined refresh immediate even though it is defined refresh deferred.

The staging table CREATE statement has the following components:

- The name of the staging table.
- A list of columns (with no attributes) in the target materialized query table. The column names do not have to match those in the target table.
- Either two or three additional columns with specific names- as provided by Db2.
- The name of the target materialized query table.

To illustrate, below is a typical materialized query table:

```
CREATE TABLE emp_sumry AS
(SELECT workdept      AS dept
      , COUNT(*)      AS #rows
      , COUNT(salary) AS #sal
      , SUM(salary)   AS sum_sal
 FROM employee emp
 GROUP BY emp.workdept
) DATA INITIALLY DEFERRED REFRESH DEFERRED;
```

Here is a staging table for the above:

*Staging table for the above materialized query table*

```
CREATE TABLE emp_sumry_s
( dept
, num_rows
, num_sal
, sum_sal
, GLOBALTRANSID
, GLOBALTRANSTIME
) FOR emp_sumry PROPAGATE IMMEDIATE;
```

#### Additional Columns

The two, or three, additional columns that every staging table must have are as follows:

- **GLOBALTRANSID**: The global transaction ID for each propagated row.

- **GLOBALTRANSTIME**: The transaction timestamp
- **OPERATIONTYPE**: The operation type (i.e. insert, update, or delete). This column is needed if the target materialized query table does not contain a GROUP BY statement.

#### Using a Staging Table

To activate the staging table one must first use the SET INTEGRITY command to remove the check pending flag, and then do a full refresh of the target materialized query table. After this is done, the staging table will record all changes to the source table. Use the refresh incremental command to apply the changes recorded in the staging table to the target materialized query table.

*Enabling and the using a staging table*

```
SET INTEGRITY FOR emp_sumry_s STAGING IMMEDIATE UNCHECKED;
REFRESH TABLE emp_sumry;
<< make changes to the source table (i.e. employee) >>
REFRESH TABLE emp_sumry INCREMENTAL;
```

A multi-row update (or insert, or delete) uses the same CURRENT TIMESTAMP for all rows changed, and for all invoked triggers. Therefore, the #CHANGING_SQL field is only incremented when a new timestamp value is detected.

# Chapter 15. Identity Columns and Sequences

Imagine that one has an INVOICE table that records invoices generated. Also imagine that one wants every new invoice that goes into this table to get an invoice number value that is part of a unique and unbroken sequence of ascending values - assigned in the order that the invoices are generated. So if the highest invoice number is currently 12345, then the next invoice will get 12346, and then 12347, and so on. There are three ways to do this, up to a point:

- Use an identity column, which generates a unique value per row in a table.
- Use a sequence, which generates a unique value per one or more tables.
- Do it yourself, using an insert trigger to generate the unique values.

You may need to know what values were generated during each insert. There are several ways to do this:

- For all of the above techniques, embed the insert inside a select statement (see Selecting identity column values inserted). This is probably the best solution.
- For identity columns, use the IDENTITY_VAL_LOCAL function (see IDENTITY_VAL_LOCAL).
- For sequences, make a NEXTVAL or PREVVAL call (see NEXTVAL and PREVVAL - Usage Notes).

**Living With Gaps**

The only way that one can be absolutely certain not to have a gap in the sequence of values generated is to create your own using an insert trigger. However, this solution is probably the least efficient of those listed here, and it certainly has the least concurrency. There is almost never a valid business reason for requiring an unbroken sequence of values. So the best thing to do, if your users ask for such a feature, is to beat them up.

**Living With Sequence Errors**

For efficiency reasons, identity column and sequence values are usually handed out (to users doing inserts) in block of values, where the block size is defined using the CACHE option. If a user inserts a row, and then dithers for a bit before inserting another, it is possible that some other user (with a higher value) will insert first. In this case, the identity column or sequence value will be a good approximation of the insert sequence, but not right on. If the users need to know the precise order with which rows were inserted, then either set the cache size to one, which will cost, or include a current timestamp value.

## 15.1. Identity Columns

One can define a column in a Db2 table as an "identity column". This column, which must be numeric (note: fractional fields not allowed), will be incremented by a fixed constant each time a new row is inserted.

Below is an example of a typical invoice table that uses an identity column that starts at one, and then goes ever upwards:

*Identity column, sample table*

```
CREATE TABLE invoice_data
 ( invoice# INTEGER           NOT NULL
                   GENERATED ALWAYS AS IDENTITY
                  ( START WITH 1
                  , INCREMENT BY 1
                  , NO MAXVALUE
                  , NO CYCLE,
                  ORDER)
 , sale_date   DATE           NOT NULL
 , customer_id CHAR(20)       NOT NULL
 , product_id  INTEGER        NOT NULL
 , quantity    INTEGER        NOT NULL
 , price       DECIMAL(18, 2) NOT NULL
 , PRIMARY KEY (invoice#));
```

## 15.2. Rules and Restrictions

Identity columns come in one of two general flavors:

- The value is always generated by Db2.
- The value is generated by Db2 only if the user does not provide a value (i.e. by default). This configuration is typically used when the input is coming from an external source (e.g. data propagation).

## 15.3. Rules

- There can only be one identity column per table.
- The field cannot be updated if it is defined "generated always".
- The column type must be numeric and must not allow fractional values. Any integer type is OK. Decimal is also fine, as long as the scale is zero. Floating point is a no-no.
- The identity column value is generated before any BEFORE triggers are applied. Use a trigger transition variable to see the value.
- A unique index is not required on the identity column, but it is a good idea. Certainly, if the value is being created by Db2, then a non-unique index is a fairly stupid idea.
- Unlike triggers, identity column logic is invoked and used during a LOAD. However, a load-replace will not reset the identity column value. Use the RESTART command (see below) to do this. An identity column is not affected by a REORG.

## 15.4. Syntax Notes

**START WITH** defines the start value, which can be any valid integer value. If no start value is provided, then the default is the MINVALUE for ascending sequences, and the MAXVALUE for descending sequences. If this value is also not provided, then the default is 1.

**INCREMENT BY** defines the interval between consecutive values. This can be any valid integer value, though using zero is pretty silly. The default is 1.

**MINVALUE** defines (for ascending sequences) the value that the sequence will start at if no start value is provided. It is also the value that an ascending sequence will begin again at after it reaches the maximum and loops around. If no minimum value is provided, then after reaching the maximum the sequence will begin again at the start value. If that is also not defined, then the sequence will begin again at 1, which is the default start value. For descending sequences, it is the minimum value that will be used before the sequence loops around, and starts again at the maximum value.

**MAXVALUE** defines (for ascending sequences) the value that a sequence will stop at, and then go back to the minimum value. For descending sequences, it is the start value (if no start value is provided), and also the restart value - if the sequence reaches the minimum and loops around.

**CYCLE** defines whether the sequence should cycle about when it reaches the maximum value (for an ascending sequences), or whether it should stop. The default is no cycle.

**CACHE** defines whether or not to allocate sequences values in chunks, and thus to save on log writes. The default is no cache, which means that every row inserted causes a log write (to save the current value). If a cache value (from 2 to 20) is provided, then the new values are assigned to a common pool in blocks. Each insert user takes from the pool, and only when all of the values are used is a new block (of values) allocated and a log write done. If the table is deactivated, either normally or otherwise, then the values in the current block are discarded, resulting in gaps in the sequence. Gaps in the sequence of values also occur when an insert is subsequently rolled back, so they cannot be avoided. But don't use the cache if you want to try and avoid them.

**ORDER** defines whether all new rows inserted are assigned a sequence number in the order that they were inserted. The default is no, which means that occasionally a row that is inserted after another may get a slightly lower sequence number. This is the default.

# 15.5. Identity Column Examples

The following example uses all of the defaults to start an identity column at one, and then to go up in increments of one. The inserts will eventually die when they reach the maximum allowed value for the field type (i.e. for small integer = 32K).

*Identity column, ascending sequence*

```
CREATE TABLE test_data
( key# SMALLINT NOT NULL
                GENERATED ALWAYS AS IDENTITY
, dat1 SMALLINT NOT NULL
, ts1 TIMESTAMP NOT NULL
, PRIMARY KEY(key#));

--  KEY# FIELD - VALUES ASSIGNED
--  1 2 3 4 5 6 7 8 9 10 11 etc.
```

The next example defines an identity column that goes down in increments of -3:

*Identity column, descending sequence*

```
CREATE TABLE test_data
( key# SMALLINT NOT NULL
                GENERATED ALWAYS AS IDENTITY
                ( START WITH 6
                , INCREMENT BY -3
                , NO CYCLE
                , NO CACHE
                , ORDER)
, dat1 SMALLINT NOT NULL
, ts1 TIMESTAMP NOT NULL
, PRIMARY KEY(key#));

-- KEY# FIELD - VALUES ASSIGNED
--  6 3 0 -3 -6 -9 -12 -15 etc.
```

The next example, which is amazingly stupid, goes nowhere fast. A primary key cannot be defined on this table:

*Identity column, dumb sequence*

```
CREATE TABLE test_data
(key# SMALLINT NOT NULL
               GENERATED ALWAYS AS IDENTITY
               ( START WITH 123
               , MAXVALUE 124
               , INCREMENT BY 0
               , NO CYCLE
               , NO ORDER)
, dat1 SMALLINT NOT NULL
, ts1 TIMESTAMP NOT NULL);

-- KEY# VALUES ASSIGNED
-- 123 123 123 123 123 123 etc.
```

The next example uses every odd number up to the maximum (i.e. 6), then loops back to the minimum value, and goes through the even numbers, ad-infinitum:

*Identity column, odd values, then even, then stuck*

```
CREATE TABLE test_data
( key# SMALLINT NOT NULL
                GENERATED ALWAYS AS IDENTITY
                ( START WITH 1
                , INCREMENT BY 2
                , MAXVALUE 6
                , MINVALUE 2
                , CYCLE
                , NO CACHE
                , ORDER)
, dat1 SMALLINT NOT NULL
, ts1 TIMESTAMP NOT NULL);

-- KEY# VALUES ASSIGNED0
-- 1 3 5 2 4 6 2 4 6 2 4 6 etc.
```

## 15.6. Usage Examples

Below is the DDL for a simplified invoice table where the primary key is an identity column. Observe that the invoice# is always generated by Db2:

*Identity column, definition*

```
CREATE TABLE invoice_data
( invoice# INTEGER     NOT NULL
                   GENERATED ALWAYS AS IDENTITY
                   ( START WITH 100
                   , INCREMENT BY 1
                   , NO CYCLE
                   , ORDER)
, sale_date DATE       NOT NULL
, customer_id CHAR(20) NOT NULL
, product_id INTEGER   NOT NULL
, quantity INTEGER     NOT NULL
, price DECIMAL(18, 2) NOT NULL
, PRIMARY KEY (invoice#));
```

One cannot provide a value for the invoice# when inserting into the above table. Therefore, one must either use a default placeholder, or leave the column out of the insert. An example of both techniques is given below. The second insert also selects the generated values:

*Invoice table, sample inserts*

```
INSERT INTO invoice_data
VALUES (DEFAULT, '2001-11-22', 'ABC', 123, 100, 10);

SELECT invoice#
FROM FINAL TABLE
    (INSERT INTO invoice_data
    (sale_date, customer_id, product_id, quantity,price)
    VALUES ('2002-11-22', 'DEF', 123, 100, 10)
         , ('2003-11-22', 'GHI', 123, 100, 10));
```

*ANSWER*

| INVOICE# |
|---|
| 101 |
| 102 |

Below is the state of the table after the above two inserts: .Invoice table, after inserts

| INVOICE# | SALE_DATE | CUSTOMER_ID | PRODUCT_ID | QUANTITY | PRICE |
|---|---|---|---|---|---|
| 100 | 2001-11-22 | ABC | 123 | 100 | 10.00 |
| 101 | 2002-11-22 | DEF | 123 | 100 | 10.00 |
| 102 | 2003-11-22 | GHI | 123 | 100 | 10.00 |

## 15.7. Altering Identity Column Options

Imagine that the application is happily collecting invoices in the above table, but your silly boss is unhappy because not enough invoices, as measured by the ever-ascending invoice# value, are being generated per unit of time. We can improve things without actually fixing any difficult business problems by simply altering the invoice# current value and the increment using the ALTER TABLE … RESTART command:

*Invoice table, restart identity column value*

```
ALTER TABLE invoice_data
    ALTER COLUMN invoice#
        RESTART WITH 1000
        SET INCREMENT BY 2;
```

Now imagine that we insert two more rows thus:

*Invoice table, more sample inserts*

```
INSERT INTO invoice_data
    VALUES (DEFAULT, '2004-11-24', 'XXX', 123, 100, 10)
         , (DEFAULT, '2004-11-25', 'YYY', 123, 100, 10);
```

Our mindless management will now see this data:

*Table 12. Invoice table, after second inserts*

| INVOICE# | SALE_DATE | CUSTOMER_ID | PRODUCT_ID | QUANTITY | PRICE |
|---|---|---|---|---|---|
| 100 | 2001-11-22 | ABC | 123 | 100 | 10.00 |
| 101 | 2002-11-22 | DEF | 123 | 100 | 10.00 |
| 102 | 2003-11-22 | GHI | 123 | 100 | 10.00 |
| 1000 | 2004-11-24 | XXX | 123 | 100 | 10.00 |
| 1002 | 2004-11-25 | YYY | 123 | 100 | 10.00 |

Restarting the identity column start number to a lower number, or to a higher number if the increment is a negative value, can result in the column getting duplicate values. This can also occur if the increment value is changed from positive to negative, or vice-versa. If no value is provided for the restart option, the sequence restarts at the previously defined start value.

## 15.8. Gaps in Identity Column Values

If an identity column is generated always, and no cache is used, and the increment value is 1, then there will usually be no gaps in the sequence of assigned values. But gaps can occur if an insert is subsequently rolled out instead of committed. In the following example, there will be no row in the table with customer number "1" after the rollback:

*Gaps in Values, example*

```
CREATE TABLE customers
( cust# INTEGER NOT NULL
                GENERATED ALWAYS AS IDENTITY (NO CACHE)
, cname CHAR(10) NOT NULL
, ctype CHAR(03) NOT NULL
, PRIMARY KEY (cust#));

COMMIT;

SELECT cust#
FROM FINAL TABLE
    (INSERT INTO customers
     VALUES (DEFAULT, 'FRED', 'XXX'));

ROLLBACK;
```

*ANSWER*

| CUST# |
| --- |
| 1 |

*Gaps in Values, example*

```
SELECT cust#
FROM FINAL TABLE
    (INSERT INTO customers
     VALUES (DEFAULT, 'FRED', 'XXX'));

COMMIT;
```

*ANSWER*

| CUST# |
| --- |
| 2 |

## 15.9. Find Gaps in Values

The following query can be used to list the missing values in a table. It starts by getting the minimum and maximum values. It next generates every value in between. Finally, it checks the generated values against the source tables. Non-matches are selected.

*Find gaps in values*

```
WITH generate_values (min_val, max_val, num_val, cur_val) AS
(SELECT MIN(dat1)
       , MAX(dat1)
       , COUNT(*)
       , MIN(dat1)
 FROM test_data td1
   UNION ALL
 SELECT min_val
      , max_val
      , num_val
      , cur_val + 1
 FROM generate_values gv1
 WHERE cur_val < max_val
)
SELECT *
FROM generate_values gv2
WHERE NOT EXISTS
    (SELECT *
     FROM test_data td2
     WHERE td2.dat1 = cur_val)
ORDER BY cur_val;
```

*INPUT*

| DAT1 |
|---|
| 1 |
| 2 |
| 3 |
| 4 |
| 6 |
| 7 |
| 9 |
| 10 |

*ANSWER*

| MIN_VAL | MAX_VAL | NUM_VAL | CUR_VAL |
|---|---|---|---|
| 1 | 10 | 8 | 5 |
| 1 | 10 | 8 | 8 |

The above query may be inefficient if there is no suitable index on the DAT1 column. The next query gets around this problem by using an EXCEPT instead of a sub-query:

*Find gaps in values*

```
WITH generate_values (min_val, max_val, num_val, cur_val) AS
(SELECT MIN(dat1)
      , MAX(dat1)
      , COUNT(*)
      , MIN(dat1)
 FROM test_data td1
   UNION ALL
 SELECT min_val
      , max_val
      , num_val
      , cur_val + 1
 FROM generate_values gv1
 WHERE cur_val < max_val)
SELECT cur_val
FROM generate_values gv2
  EXCEPT ALL
SELECT dat1
FROM test_data td2
ORDER BY 1;
```

*INPUT*

| DAT1 |
|---|
| 1 |
| 2 |
| 3 |
| 4 |
| 6 |
| 7 |
| 9 |
| 10 |

*ANSWER*

| CUR_VAL |
|---|
| 5 |
| 8 |

The next query uses a totally different methodology. It assigns a rank to every value, and then looks for places where the rank and value get out of step:

*Find gaps in values*

```
WITH assign_ranks AS
(SELECT dat1
      , DENSE_RANK() OVER(ORDER BY dat1) AS rank#
 FROM test_data)
, locate_gaps AS
(SELECT dat1 - rank# AS diff
      , min(dat1) AS min_val
      , max(dat1) AS max_val
      , ROW_NUMBER() OVER(ORDER BY dat1 - rank#) AS gap#
 FROM assign_ranks ar1
 GROUP BY dat1 - rank#)
SELECT lg1.gap# AS gap#
     , lg1.max_val AS prev_val
     , lg2.min_val AS next_val
     , lg2.min_val - lg1.max_val AS diff
FROM locate_gaps lg1
   , locate_gaps lg2
WHERE lg2.gap# = lg1.gap# + 1
ORDER BY lg1.gap#;
```

*ANSWER*

| GAP# | PREV_VAL | NEXT_VAL | DIFF |
|---|---|---|---|
| 1 | 4 | 6 | 2 |
| 2 | 7 | 9 | 2 |

### 15.9.1. IDENTITY_VAL_LOCAL Function

There are two ways to find out what values were generated when one inserted a row into a table with an identity column:

- Embed the insert within a select statement (see Selecting identity column values inserted).
- Call the IDENTITY_VAL_LOCAL function.

Certain rules apply to IDENTITY_VAL_LOCAL function usage:

- The value returned from is a decimal (31.0) field.
- The function returns null if the user has not done a single-row insert in the current unit of work. Therefore, the function has to be invoked before one does a commit. Having said this, in some versions of Db2 it seems to work fine after a commit.
- If the user inserts multiple rows into table(s) having identity columns in the same unit of work, the result will be the value obtained from the last single-row insert. The result will be null if there was none.
- Multiple-row inserts are ignored by the function. So if the user first inserts one row, and then separately inserts two rows (in a single SQL statement), the function will return the identity column value generated during the first insert.
- The function cannot be called in a trigger or SQL function. To get the current identity column value in an insert trigger, use the trigger transition variable for the column. The value, and thus the transition variable, is defined before the trigger is begun.
- If invoked inside an insert statement (i.e. as an input value), the value will be taken from the most recent (previous) single-row insert done in the same unit of work. The result will be null if there was none.
- The value returned by the function is unpredictable if the prior single-row insert failed. It may be the value from the insert before, or it may be the value given to the failed insert.
- The function is non-deterministic, which means that the result is determined at fetch time (i.e. not at open) when used in a cursor. So if one fetches a row from a cursor, and then does an insert, the next fetch may get a different value from the prior.
- The value returned by the function may not equal the value in the table - if either a trigger or an update has changed the field since the value was generated. This can only occur if the identity column is defined as being "generated by default". An identity column that is "generated always" cannot be updated.
- When multiple users are inserting into the same table concurrently, each will see their own most recent identity column value. They cannot see each other’s.

If the above sounds unduly complex, it is because it is. It is often much easier to simply get the values by embedding the insert inside a select:

*Selecting identity column values inserted*

```
SELECT MIN(cust#) AS minc
     , MAX(cust#) AS maxc
     , COUNT(*) AS rows
FROM FINAL TABLE
    (INSERT INTO customers
     VALUES (DEFAULT, 'FRED', 'xxx')
          , (DEFAULT, 'DAVE', 'yyy')
          , (DEFAULT, 'JOHN', 'zzz'));
```

*ANSWER*

| MINC | MAXC | ROWS |
|---|---|---|
| 3 | 5 | 3 |

Below are two examples of the function in use. Observe that the second invocation (done after the commit) returned a value, even though it is supposed to return null:

*IDENTITY_VAL_LOCAL function examples*

```
CREATE TABLE invoice_table
( invoice# INTEGER NOT NULL
                   GENERATED ALWAYS AS IDENTITY
, sale_date DATE NOT NULL
, customer_id CHAR(20) NOT NULL
, product_id INTEGER NOT NULL
, quantity INTEGER NOT NULL
, price DECIMAL(18,2) NOT NULL
, PRIMARY KEY (invoice#));

COMMIT;

INSERT INTO invoice_table
VALUES (DEFAULT, '2000-11-22', 'ABC', 123, 100, 10);

WITH temp (id) AS
  (VALUES (IDENTITY_VAL_LOCAL()))
SELECT *
FROM temp;    --> ANSWER: ID = 1

COMMIT;

WITH temp (id) AS
  (VALUES (IDENTITY_VAL_LOCAL()))
SELECT *
FROM temp;    --> ANSWER: ID = 1
```

In the next example, two separate inserts are done on the table defined above. The first inserts a

single row, and so sets the function value to "2". The second is a multi-row insert, and so is ignored by the function:

*IDENTITY_VAL_LOCAL function examples*

```
INSERT INTO invoice_table
  VALUES (DEFAULT, '2000-11-23', 'ABC', 123, 100, 10);
INSERT INTO invoice_table
  VALUES (DEFAULT, '2000-11-24', 'ABC', 123, 100, 10)
       , (DEFAULT, '2000-11-25', 'ABC', 123, 100, 10);

SELECT invoice# AS inv#
     , sale_date
     , IDENTITY_VAL_LOCAL() AS id
FROM invoice_table
ORDER BY 1;

COMMIT;
```

*ANSWER*

| INV# | SALE_DATE | ID |
|---|---|---|
| 1 | 11/22/2000 | 2 |
| 2 | 11/23/2000 | 2 |
| 3 | 11/24/2000 | 2 |
| 4 | 11/25/2000 | 2 |

One can also use the function to get the most recently inserted single row by the current user:

*IDENTITY_VAL_LOCAL usage in predicate*

```
SELECT invoice# AS inv#
     , sale_date
     , IDENTITY_VAL_LOCAL() AS id
FROM invoice_table
WHERE id = IDENTITY_VAL_LOCAL();
```

*ANSWER*

| INV# | SALE_DATE | ID |
|---|---|---|
| 2 | 11/23/2000 | 2 |

## 15.10. Sequences

A sequence is almost the same as an identity column, except that it is an object that exists outside of any particular table.

*Create sequence*

```
CREATE SEQUENCE fred
  AS DECIMAL(31)
  START WITH 100
  INCREMENT BY 2
  NO MINVALUE
  NO MAXVALUE
  NO CYCLE
  CACHE 20
  ORDER;

-- SEQ# VALUES ASSIGNED
-- 100 102 104 106 etc.
```

The options and defaults for a sequence are exactly the same as those for an identity column (see Rules and Restrictions). Likewise, one can alter a sequence in much the same way as one would alter the status of an identity column:

*Alter sequence attributes*

```
ALTER SEQUENCE fred
  RESTART WITH -55
  INCREMENT BY -5
  MINVALUE -1000
  MAXVALUE +1000
  NO CACHE
  NO ORDER
  CYCLE;

-- SEQ# VALUES ASSIGNED
-- -55 -60 -65 -70 etc.
```

The only sequence attribute that one cannot change with the ALTER command is the field type that is used to hold the current value.

### 15.10.1. Constant Sequence

If the increment is zero, the sequence will stay whatever value one started it with until it is altered. This can be useful if wants to have a constant that can be globally referenced:

*Sequence that doesn't change*

```
CREATE SEQUENCE biggest_sale_to_date
  AS INTEGER
  START WITH 345678
  INCREMENT BY 0;

-- SEQ# VALUES ASSIGNED
-- 345678, 345678, etc.
```

### 15.10.2. Getting the Sequence Value

There is no concept of a current sequence value. Instead one can either retrieve the next or the previous value (if there is one). And any reference to the next value will invariably cause the sequence to be incremented. The following example illustrates this:

*Selecting the NEXTVAL*

```
CREATE SEQUENCE fred;

COMMIT;

WITH temp1 (n1) AS
 (VALUES 1
    UNION ALL
  SELECT n1 + 1
  FROM temp1
  WHERE n1 < 5
 )
 SELECT NEXTVAL FOR fred AS seq#
 FROM temp1;
```

*ANSWER*

| SEQ# |
|---|
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |

### 15.10.3. NEXTVAL and PREVVAL - Usage Notes

- One retrieves the next or previous value using a "NEXTVAL FOR sequence-name", or a "PREVVAL for sequence-name" call.
- A NEXTVAL call generates and returns the next value in the sequence. Thus, each call will

consume the returned value. This remains true even if the statement that did the retrieval subsequently fails or is rolled back.

- A PREVVAL call returns the most recently generated value for the specified sequence for the current connection. Unlike when getting the next value, getting the prior value does not alter the state of the sequence, so multiple calls can retrieve the same value.
- If no NEXTVAL reference (to the target sequence) has been made for the current connection, any attempt to get the PREVVAL will result in a SQL error.

### 15.10.4. NEXTVAL and PREVVAL - Usable Statements

- SELECT INTO statement (within the select part), as long as there is no DISTINCT, GROUP BY, UNION, EXECPT, or INTERSECT.
- INSERT statement - with restrictions.
- UPDATE statement - with restrictions.
- SET host variable statement.

### 15.10.5. NEXTVAL - Usable Statements

- A trigger.

### 15.10.6. NEXTVAL and PREVVAL - Not Allowed In

- DELETE statement.
- Join condition of a full outer join.
- Anywhere in a CREATE TABLE or CREATE VIEW statement.

### 15.10.7. NEXTVAL - Not Allowed In

- CASE expression
- Join condition of a join.
- Parameter list of an aggregate function.
- SELECT statement where there is an outer select that contains a DISTINCT, GROUP BY, UNION, EXCEPT, or INTERSECT.
- Most sub-queries.

### 15.10.8. PREVVAL - Not Allowed In

- A trigger.

There are many more usage restrictions, but you presumably get the picture. See the Db2 SQL Reference for the complete list.

# 15.11. Usage Examples

Below a sequence is defined, then various next and previous values are retrieved:

*NEXTVAL and PREVVAL expressions*

```
CREATE SEQUENCE fred;

COMMIT;

WITH temp1 (prv) AS
  (VALUES (PREVVAL FOR fred))
SELECT *
FROM temp1; -->  PRV : <error>

WITH temp1 (nxt) AS
  (VALUES (NEXTVAL FOR fred))
SELECT *
FROM temp1; --> NXT: 1

WITH temp1 (prv) AS
  (VALUES (PREVVAL FOR fred))
SELECT *
FROM temp1; --> PRV: 1

WITH temp1 (n1) AS
  (VALUES 1
     UNION ALL
   SELECT n1 + 1
   FROM temp1
   WHERE n1 < 5
  )
SELECT NEXTVAL FOR fred AS nxt
     , PREVVAL FOR fred AS prv
FROM temp1;
```

| NXT | PRV |
|---|---|
| 2 | 1 |
| 3 | 1 |
| 4 | 1 |
| 5 | 1 |
| 6 | 1 |

One does not actually have to fetch a NEXTVAL result in order to increment the underlying sequence. In the next example, some of the rows processed are thrown away halfway thru the query, but their usage still affects the answer (of the subsequent query):

*NEXTVAL values used but not retrieved*

```
CREATE SEQUENCE fred;

COMMIT;

WITH temp1 AS
  (SELECT id
        , NEXTVAL FOR fred AS nxt
   FROM staff
   WHERE id < 100
  )
SELECT *
FROM temp1
WHERE id = 50 + (nxt * 0);
```

| ID | NXT |
|---|---|
| 50 | 5 |

```
WITH temp1 (nxt, prv) AS
  (VALUES (NEXTVAL FOR fred
         , PREVVAL FOR fred))
SELECT *
FROM temp1;
```

| NXT | PRV |
|---|---|
| 10 | 9 |

The somewhat funky predicate at the end of the first query above prevents Db2 from stopping the nested-table-expression when it gets to "id = 50". If this were to occur, the last query above would get a next value of 6, and a previous value of 5.

## 15.12. Multi-table Usage

Imagine that one wanted to maintain a unique sequence of values over multiple tables. One can do this by creating a before insert trigger on each table that replaces whatever value the user provides with the current one from a common sequence. Below is an example:

*Create tables that use a common sequence*

```
CREATE SEQUENCE cust#
  START WITH 1
  INCREMENT BY 1
  NO MAXVALUE
  NO CYCLE
  ORDER;

CREATE TABLE us_customer
( cust# INTEGER  NOT NULL
, cname CHAR(10) NOT NULL
, frst_sale DATE NOT NULL
, #sales INTEGER NOT NULL
, PRIMARY KEY (cust#));

CREATE TRIGGER us_cust_ins
  NO CASCADE BEFORE INSERT ON us_customer
  REFERENCING NEW AS nnn
  FOR EACH ROW MODE Db2SQL
  SET nnn.cust# = NEXTVAL FOR cust#;

CREATE TABLE intl_customer
( cust# INTEGER  NOT NULL
, cname CHAR(10) NOT NULL
, frst_sale DATE NOT NULL
, #sales INTEGER NOT NULL
, PRIMARY KEY (cust#));

CREATE TRIGGER intl_cust_ins
  NO CASCADE BEFORE INSERT ON intl_customer
  REFERENCING NEW AS nnn
  FOR EACH ROW MODE Db2SQL
  SET nnn.cust# = NEXTVAL FOR cust#;
```

If we now insert some rows into the above tables, we shall find that customer numbers are assigned in the correct order, thus:

*Insert into tables with common sequence*

```
SELECT cust#
     , cname
FROM FINAL TABLE
(INSERT INTO us_customer (cname, frst_sale, #sales)
 VALUES ('FRED', '2002-10-22', 1)
      , ('JOHN', '2002-10-23', 1));

SELECT cust#
     , cname
FROM FINAL TABLE
(INSERT INTO intl_customer (cname, frst_sale, #sales)
 VALUES ('SUE', '2002-11-12', 2)
      , ('DEB', '2002-11-13', 2));
```

*ANSWERS*

| CUST# | CNAME |
|---|---|
| 1 | FRED |
| 2 | JOHN |

| CUST# | CNAME |
|---|---|
| 3 | SUE |
| 4 | DEB |

One of the advantages of a standalone sequence over a functionally similar identity column is that one can use a PREVVAL expression to get the most recent value assigned (to the user), even if the previous usage was during a multi-row insert. Thus, after doing the above inserts, we can run the following query:

*Get previous value - select*

```
WITH temp (prev) AS
  (VALUES (PREVVAL FOR cust#))
SELECT *
FROM temp;
```

*ANSWER*

| PREV |
|---|
| 4 |

The following does the same as the above, but puts the result in a host variable:

*Get previous value - into host-variable*

```
VALUES PREVVAL FOR CUST# INTO :host-var
```

As with identity columns, the above result will not equal what is actually in the table(s) – if the most recent insert was subsequently rolled back.

## 15.13. Counting Deletes

In the next example, two sequences are created: One records the number of rows deleted from a table, while the other records the number of delete statements run against the same:

*Count deletes done to table*

```
CREATE SEQUENCE delete_rows
  START WITH 1
  INCREMENT BY 1
  NO MAXVALUE
  NO CYCLE
  ORDER;

CREATE SEQUENCE delete_stmts
  START WITH 1
  INCREMENT BY 1
  NO MAXVALUE
  NO CYCLE
  ORDER;

CREATE TABLE customer
( cust# INTEGER  NOT NULL
, cname CHAR(10) NOT NULL
, frst_sale DATE NOT NULL
, #sales INTEGER NOT NULL
, PRIMARY KEY (cust#));

CREATE TRIGGER cust_del_rows
  AFTER DELETE ON customer
  FOR EACH ROW MODE Db2SQL
  WITH temp1 (n1) AS (VALUES(1))
  SELECT NEXTVAL FOR delete_rows
  FROM temp1;

CREATE TRIGGER cust_del_stmts
  AFTER DELETE ON customer
  FOR EACH STATEMENT MODE Db2SQL
  WITH temp1 (n1) AS (VALUES(1))
  SELECT NEXTVAL FOR delete_stmts
  FROM temp1;
```

Be aware that the second trigger will be run, and thus will update the sequence, regardless of whether a row was found to delete or not.

## 15.14. Identity Columns vs. Sequences - a Comparison

First to compare the two types of sequences:

- Only one identity column is allowed per table, whereas a single table can have multiple sequences and/or multiple references to the same sequence.
- Identity column sequences cannot span multiple tables. Sequences can.
- Sequences require triggers to automatically maintain column values (e.g. during inserts) in tables. Identity columns do not.
- Sequences can be incremented during inserts, updates, deletes (via triggers), or selects, whereas identity columns only get incremented during inserts.
- Sequences can be incremented (via triggers) once per row, or once per statement. Identity columns are always updated per row inserted.
- Sequences can be dropped and created independent of any tables that they might be used to maintain values in. Identity columns are part of the table definition.
- Identity columns are supported by the load utility. Trigger induced sequences are not.

For both types of sequence, one can get the current value by embedding the DML statement inside a select (e.g. see Selecting identity column values inserted). Alternatively, one can use the relevant expression to get the current status. These differ as follows:

- The **IDENTITY_VAL_LOCAL** function returns null if no inserts to tables with identity columns have been done by the current user. In an equivalent situation, the PREVVAL expression gets a nasty SQL error.
- The **IDENTITY_VAL_LOCAL** function ignores multi-row inserts (without telling you). In a similar situation, the PREVVAL expression returns the last value generated.
- One cannot tell to which table an **IDENTITY_VAL_LOCAL** function result refers to. This can be a problem in one insert invokes another insert (via a trigger), which puts are row in another table with its own identity column. By contrast, in the PREVVAL function one explicitly identifies the sequence to be read.
- There is no equivalent of the NEXTVAL expression for identity columns.

## 15.15. Roll Your Own

If one really, really, needs to have a sequence of values with no gaps, then one can do it using an insert trigger, but there are costs, in processing time, concurrency, and functionality. To illustrate, consider the following table:

*Sample table, roll your own sequence*

```
CREATE TABLE sales_invoice
( invoice#    INTEGER        NOT NULL
, sale_date   DATE           NOT NULL
, customer_id CHAR(20)       NOT NULL
, product_id  INTEGER        NOT NULL
, quantity    INTEGER        NOT NULL
, price       DECIMAL(18, 2) NOT NULL
, PRIMARY KEY (invoice#));
```

The following trigger will be invoked before each row is inserted into the above table. It sets the new invoice# value to be the current highest invoice# value in the table, plus one:

*Sample trigger, roll your own sequence*

```
CREATE TRIGGER sales_insert
  NO CASCADE BEFORE
  INSERT ON sales_invoice
  REFERENCING NEW AS nnn
  FOR EACH ROW
  MODE Db2SQL
  SET nnn.invoice# =
    (SELECT COALESCE(MAX(invoice#),0) + 1
     FROM sales_invoice);
```

The good news about the above setup is that it will never result in gaps in the sequence of values. In particular, if a newly inserted row is rolled back after the insert is done, the next insert will simply use the same invoice# value. But there is also bad news: * Only one user can insert at a time, because the select (in the trigger) needs to see the highest invoice# in the table in order to complete. * Multiple rows cannot be inserted in a single SQL statement (i.e. a mass insert). The trigger is invoked before the rows are actually inserted, one row at a time, for all rows. Each row would see the same, already existing, high invoice#, so the whole insert would die due to a duplicate row violation.

There may be a tiny, tiny chance that if two users were to begin an insert at exactly the same time that they would both see the same high invoice# (in the before trigger), and so the last one to complete (i.e. to add a pointer to the unique invoice# index) would get a duplicate-row violation.

Below are some inserts to the above table. Ignore the values provided in the first field – they are replaced in the trigger. And observe that the third insert is rolled out:

*Sample inserts, roll your own sequence*

```
INSERT INTO sales_invoice VALUES (0, '2001-06-22' ,'ABC', 123, 10, 1);
INSERT INTO sales_invoice VALUES (0, '2001-06-23' ,'DEF', 453, 10, 1);
COMMIT;
INSERT INTO sales_invoice VALUES (0, '2001-06-24' ,'XXX', 888, 10, 1);
ROLLBACK;
INSERT INTO sales_invoice VALUES (0, '2001-06-25' ,'YYY', 999, 10, 1);
COMMIT;
```

*ANSWER*

| INVOICE# | SALE_DATE | CUSTOMER_ID | PRODUCT_ID | QUANTITY | PRICE |
|---|---|---|---|---|---|
| 1 | 06/22/2001 | ABC | 123 | 10 | 1.00 |
| 2 | 06/23/2001 | DEF | 453 | 10 | 1.00 |
| 3 | 06/25/2001 | YYY | 999 | 10 | 1.00 |

# 15.16. Support Multi-row Inserts

The next design is more powerful in that it supports multi-row inserts, and also more than one table if desired. It requires that there be a central location that holds the current high-value. In the example below, this value will be in a row in a special control table. Every insert into the related data table will, via triggers, first update, and then query, the row in the control table.

## 15.16.1. Control Table

The following table has one row per sequence of values being maintained:

*Control Table, DDL*

```
CREATE TABLE control_table
( table_name CHAR(18) NOT NULL
, table_nmbr INTEGER  NOT NULL
, PRIMARY KEY (table_name));
```

Now to populate the table with some initial sequence# values:

*Control Table, sample inserts*

```
INSERT INTO control_table VALUES ('invoice_table', 0);
INSERT INTO control_table VALUES ('2nd_data_tble', 0);
INSERT INTO control_table VALUES ('3rd_data_tble', 0);
```

## 15.16.2. Data Table

Our sample data table has two fields of interest:

- The **UNQVAL** column will be populated, using a trigger, with a GENERATE_UNIQUE function output value. This is done before the row is actually inserted. Once the insert has completed, we will no longer care about or refer to the contents of this field.
- The **INVOICE#** column will be populated, using triggers, during the insert process with a unique ascending value. However, for part of the time during the insert the field will have a null value, which is why it is defined as being both non-unique and allowing nulls.

*Sample Data Table, DDL*

```
CREATE TABLE invoice_table
( unqval      CHAR(13) FOR BIT DATA NOT NULL
, invoice#    INTEGER               NOT NULL
, sale_date   DATE                  NOT NULL
, customer_id CHAR(20)              NOT NULL
, product_id  INTEGER               NOT NULL
, quantity    INTEGER               NOT NULL
, price       DECIMAL(18, 2)        NOT NULL
, PRIMARY KEY(unqval));
```

Two insert triggers are required: The first acts before the insert is done, giving each new row a unique UNQVAL value:

*Before trigger*

```
CREATE TRIGGER invoice1
  NO CASCADE BEFORE INSERT ON invoice_table
  REFERENCING NEW AS nnn
  FOR EACH ROW MODE Db2SQL
  SET nnn.unqval   = GENERATE_UNIQUE()
    , nnn.invoice# = NULL;
```

The second trigger acts after the row is inserted. It first increments the control table by one, then updates invoice# in the current row with the same value. The UNQVAL field is used to locate the row to be changed in the second update:

*After trigger*

```
CREATE TRIGGER invoice2
  AFTER INSERT ON invoice_table
  REFERENCING NEW AS nnn
  FOR EACH ROW MODE Db2SQL
  BEGIN ATOMIC
    UPDATE control_table
      SET table_nmbr = table_nmbr + 1
      WHERE table_name = 'invoice_table';
    UPDATE invoice_table
      SET invoice# =
               (SELECT table_nmbr
                FROM control_table
                WHERE table_name = 'invoice_table')
    WHERE unqval = nnn.unqval
    AND invoice# IS NULL;
  END
```

The above two actions must be in a single trigger. If they are in two triggers, mass inserts will not work correctly because the first trigger (i.e. update) would be run (for all rows), followed by the second trigger (for all rows). In the end, every row inserted by the mass-insert would end up with the same invoice# value.

A final update trigger is required to prevent updates to the invoice# column:

*Update trigger*

```
CREATE TRIGGER invoice3
  NO CASCADE BEFORE UPDATE OF invoice# ON invoice_table
  REFERENCING OLD AS ooo
  NEW AS nnn
  FOR EACH ROW MODE Db2SQL
  WHEN (ooo.invoice# <> nnn.invoice#)
    SIGNAL SQLSTATE '71001' ('no updates allowed - you twit');
```

# 15.17. Design Comments

Though the above design works, it has certain practical deficiencies:

- The single row in the control table is a point of contention, because only one user can update it at a time. One must therefore commit often (perhaps more often than one would like to) in order to free up the locks on this row. Therefore, by implication, this design puts one is at the mercy of programmers.
- The two extra updates add a considerable overhead to the cost of the insert.
- The invoice number values generated by AFTER trigger cannot be obtained by selecting from an insert statement (see Insert Examples). In fact, selecting from the FINAL TABLE will result in a

SQL error. One has to instead select from the NEW TABLE, which returns the new rows before the AFTER trigger was applied.

As with ordinary sequences, this design enables one to have multiple tables referring to a single row in the control table, and thus using a common sequence.

# Chapter 16. Temporary Tables

## 16.1. Introduction

How one defines a temporary table depends in part upon how often, and for how long, one intends to use it:

- Within a query, single use.
- Within a query, multiple uses.
- For multiple queries in one unit of work.
- For multiple queries, over multiple units of work, in one thread.

## 16.2. Single Use in Single Statement

If one intends to use a temporary table just once, it can be defined as a nested table expression. In the following example, we use a temporary table to sequence the matching rows in the STAFF table by descending salary. We then select the 2nd through 3rd rows:

*Nested Table Expression*

```
SELECT id
     , salary
FROM (SELECT s.*
           , ROW_NUMBER() OVER(ORDER BY salary DESC) AS sorder
      FROM staff s
      WHERE id < 200
     ) AS xxx
WHERE sorder BETWEEN 2 AND 3
ORDER BY id;
```

*ANSWER*

| ID | SALARY |
|---|---|
| 50 | 20659.80 |
| 140 | 21150.00 |

A fullselect in parenthesis followed by a correlation name (see above) is also called a nested table expression.

Here is another way to express the same:

*Common Table Expression*

```
WITH xxx (id, salary, sorder) AS
(SELECT ID
      , salary
      , ROW_NUMBER() OVER(ORDER BY salary DESC) AS sorder
 FROM staff
 WHERE id < 200
)
SELECT id
     , salary
FROM xxx
WHERE sorder BETWEEN 2 AND 3
ORDER BY id;
```

*ANSWER*

| ID | SALARY |
|---|---|
| 50 | 20659.80 |
| 140 | 21150.00 |

# 16.3. Multiple Use in Single Statement

Imagine that one wanted to get the percentage contribution of the salary in some set of rows in the STAFF table - compared to the total salary for the same. The only way to do this is to access the matching rows twice; Once to get the total salary (i.e. just one row), and then again to join the total salary value to each individual salary - to work out the percentage.

Selecting the same set of rows twice in a single query is generally unwise because repeating the predicates increases the likelihood of typos being made. In the next example, the desired rows are first placed in a temporary table. Then the sum salary is calculated and placed in another temporary table. Finally, the two temporary tables are joined to get the percentage:

*Common Table Expression*

```
WITH rows_wanted AS
(SELECT *
 FROM staff
 WHERE id < 100
 AND UCASE(name) LIKE '%T%')
, sum_salary AS
(SELECT SUM(salary) AS sum_sal
FROM rows_wanted)
SELECT id
     , name
     , salary
     , sum_sal
     , INT((salary * 100) / sum_sal) AS pct
FROM rows_wanted
   , sum_salary
ORDER BY id;
```

*ANSWER*

| ID | NAME | SALARY | SUM_SAL | PCT |
|---|---|---|---|---|
| 70 | Rothman | 16502.83 | 34504.58 | 47 |
| 90 | Koonitz | 18001.75 | 34504.58 | 52 |

### 16.3.1. Multiple Use in Multiple Statements

To refer to a temporary table in multiple SQL statements in the same thread, one has to define a declared global temporary table. An example follows:

*Declared Global Temporary Table*

```
DECLARE GLOBAL TEMPORARY TABLE session.fred
( dept SMALLINT NOT NULL
, avg_salary DEC(7, 2)NOT NULL
, num_emps SMALLINT NOT NULL)
ON COMMIT PRESERVE ROWS;

COMMIT;

INSERT INTO session.fred
SELECT dept
     , AVG(salary)
     , COUNT(*)
FROM staff
WHERE id > 200
GROUP BY dept;

COMMIT;

SELECT COUNT(*) AS cnt
FROM session.fred;

DELETE FROM session.fred
WHERE dept > 80;

SELECT *
FROM session.fred;
```

*ANSWER#1*

| CNT |
|---|
| 4 |

*ANSWER#2*

| DEPT | AVG_SALARY | NUM_EMPS |
|---|---|---|
| 10 | 20168.08 | 3 |
| 51 | 15161.43 | 3 |
| 66 | 17215.24 | 5 |

Unlike an ordinary table, a declared global temporary table is not defined in the Db2 catalogue. Nor is it sharable by other users. It only exists for the duration of the thread (or less) and can only be seen by the person who created it. For more information, see Declared Global Temporary Tables.

### 16.3.2. Temporary Tables - in Statement

Three general syntaxes are used to define temporary tables in a query:

- Use a WITH phrase at the top of the query to define a common table expression.
- Define a fullselect in the FROM part of the query.
- Define a fullselect in the SELECT part of the query.

The following three queries, which are logically equivalent, illustrate the above syntax styles. Observe that the first two queries are explicitly defined as left outer joins, while the last one is implicitly a left outer join:

*Identical query (1 of 3) - using Common Table Expression*

```
WITH staff_dept AS
(SELECT dept        AS dept#
      , MAX(salary) AS max_sal
 FROM staff
 WHERE dept < 50
 GROUP BY dept
)
SELECT id
     , dept
     , salary
     , max_sal
FROM staff
LEFT OUTER JOIN staff_dept
ON dept = dept#
WHERE name LIKE 'S%'
ORDER BY id;
```

*ANSWER*

| ID | DEPT | SALARY | MAX_SAL |
|---|---|---|---|
| 10 | 20 | 18357.50 | 18357.50 |
| 190 | 20 | 14252.75 | 18357.50 |
| 200 | 42 | 11508.60 | 18352.80 |
| 220 | 51 | 17654.50 | |

*Identical query (2 of 3) - using fullselect in FROM*

```
SELECT id
     , dept
     , salary
     , max_sal
FROM staff
LEFT OUTER JOIN
    (SELECT dept        AS dept#
          , MAX(salary) AS max_sal
     FROM staff
     WHERE dept < 50
     GROUP BY dept) AS STAFF_dept
ON dept = dept#
WHERE name LIKE 'S%'
ORDER BY id;
```

*ANSWER*

| ID | DEPT | SALARY | MAX_SAL |
|---|---|---|---|
| 10 | 20 | 18357.50 | 18357.50 |
| 190 | 20 | 14252.75 | 18357.50 |
| 200 | 42 | 11508.60 | 18352.80 |
| 220 | 51 | 17654.50 | |

*Identical query (3 of 3) - using fullselect in SELECT*

```
SELECT id
     , dept
     , salary
     , (SELECT MAX(salary)
        FROM staff s2
        WHERE s1.dept = s2.dept
        AND s2.dept < 50
        GROUP BY dept) AS max_sal
FROM staff s1
WHERE name LIKE 'S%'
ORDER BY id;
```

*ANSWER*

| ID | DEPT | SALARY | MAX_SAL |
|---|---|---|---|
| 10 | 20 | 18357.50 | 18357.50 |
| 190 | 20 | 14252.75 | 18357.50 |
| 200 | 42 | 11508.60 | 18352.80 |

| ID | DEPT | SALARY | MAX_SAL |
|---|---|---|---|
| 220 | 51 | 17654.50 | |

# 16.4. Common Table Expression

A common table expression is a named temporary table that is retained for the duration of a SQL statement. There can be many temporary tables in a single SQL statement. Each must have a unique name and be defined only once. All references to a temporary table (in a given SQL statement run) return the same result. This is unlike tables, views, or aliases, which are derived each time they are called. Also unlike tables, views, or aliases, temporary tables never contain indexes. Certain rules apply to common table expressions:

- Column names must be specified if the expression is recursive, or if the query invoked returns duplicate column names.
- The number of column names (if any) that are specified must match the number of columns returned.
- If there is more than one common-table-expression, latter ones (only) can refer to the output from prior ones. Cyclic references are not allowed.
- A common table expression with the same name as a real table (or view) will replace the real table for the purposes of the query. The temporary and real tables cannot be referred to in the same query.
- Temporary table names must follow standard Db2 table naming standards.
- Each temporary table name must be unique within a query.
- Temporary tables cannot be used in sub-queries.

## 16.4.1. Select Examples

In this first query, we don't have to list the field names (at the top) because every field already has a name (given in the SELECT):

*Common Table Expression, using named fields*

```
WITH temp1 AS
(SELECT MAX(name) AS max_name
      , MAX(dept) AS max_dept
 FROM staff)
SELECT *
FROM temp1;
```

*ANSWER*

| MAX_NAME | MAX_DEPT |
|---|---|
| Yamaguchi | 84 |

In this next example, the fields being selected are unnamed, so names have to be specified in the

WITH statement:

*Common Table Expression, using unnamed fields*

```
WITH temp1 (max_name, max_dept) AS
(SELECT MAX(name)
      , MAX(dept)
 FROM staff)
SELECT *
FROM temp1;
```

*ANSWER*

| MAX_NAME | MAX_DEPT |
|---|---|
| Yamaguchi | 84 |

A single query can have multiple common-table-expressions. In this next example we use two expressions to get the department with the highest average salary:

*Query with two common table expressions*

```
WITH temp1 AS
(SELECT dept
      , AVG(salary) AS avg_sal
 FROM staff
 GROUP BY dept)
, temp2 AS
(SELECT MAX(avg_sal) AS max_avg
 FROM temp1)
SELECT *
FROM temp2;
```

*ANSWER*

| MAX_AVG |
|---|
| 20865.8625 |

FYI, the exact same query can be written using nested table expressions thus:

*Same as prior example, but using nested table expressions*

```
SELECT *
FROM
    (SELECT MAX(avg_sal) AS max_avg
     FROM (SELECT dept
                , AVG(salary) AS avg_sal
           FROM staff
           GROUP BY dept
          ) AS temp1
    ) AS temp2;
```

*ANSWER*

| MAX_AVG |
| --- |
| 20865.8625 |

The next query first builds a temporary table, then derives a second temporary table from the first, and then joins the two temporary tables together. The two tables refer to the same set of rows, and so use the same predicates. But because the second table was derived from the first, these predicates only had to be written once. This greatly simplified the code:

*Deriving second temporary table from first*

```
WITH temp1 AS
(SELECT id
      , name
      , dept
      , salary
 FROM staff
 WHERE id < 300
 AND dept <> 55
 AND name LIKE 'S%'
 AND dept NOT IN
     (SELECT deptnumb
      FROM org
      WHERE division = 'SOUTHERN'
      OR location = 'HARTFORD'
     )
)
, temp2 AS
(SELECT dept
      , MAX(salary) AS max_sal
 FROM temp1
 GROUP BY dept)
SELECT t1.id
     , t1.dept
     , t1.salary
     , t2.max_sal
FROM temp1 t1
   , temp2 t2
WHERE t1.dept = t2.dept
ORDER BY t1.id;
```

*ANSWER*

| ID | DEPT | SALARY | MAX_SAL |
|---|---|---|---|
| 10 | 20 | 18357.50 | 18357.50 |
| 190 | 20 | 14252.75 | 18357.50 |
| 200 | 42 | 11508.60 | 11508.60 |
| 220 | 51 | 17654.50 | 17654.50 |

## 16.5. Insert Usage

A common table expression can be used to an insert-select-from statement to build all or part of the set of rows that are inserted:

*Insert using common table expression*

```
INSERT INTO staff
WITH temp1 (max1) AS
(SELECT MAX(id) + 1
 FROM staff)
SELECT max1, 'A', 1, 'B', 2, 3, 4
FROM temp1;
```

As it happens, the above query can be written equally well in the raw:

*Equivalent insert (to above) without common table expression*

```
INSERT INTO staff
SELECT MAX(id) + 1, 'A', 1, 'B', 2, 3, 4
FROM staff;
```

# 16.6. Full-Select

A fullselect is an alternative way to define a temporary table. Instead of using a WITH clause at the top of the statement, the temporary table definition is embedded in the body of the SQL statement. Certain rules apply:

- When used in a select statement, a fullselect can either be generated in the FROM part of the query - where it will return a temporary table, or in the SELECT part of the query where it will return a column of data.
- When the result of a fullselect is a temporary table (i.e. in FROM part of a query), the table must be provided with a correlation name.
- When the result of a fullselect is a column of data (i.e. in SELECT part of query), each reference to the temporary table must only return a single value.

# 16.7. Full-Select in FROM Phrase

The following query uses a nested table expression to get the average of an average - in this case the average departmental salary (an average in itself) per division:

*Nested column function usage*

```
SELECT division
     , DEC(AVG(dept_avg),7,2) AS div_dept
     , COUNT(*) AS #dpts
     , SUM(#emps) AS #emps
FROM
    (SELECT division
          , dept
          , AVG(salary) AS dept_avg
          , COUNT(*) AS #emps
     FROM staff
        , org
     WHERE dept = deptnumb
     GROUP BY division
            , dept) AS xxx
GROUP BY division;
```

*ANSWER*

| DIVISION | DIV_DEPT | #DPTS | #EMPS |
|---|---|---|---|
| Corporate | 20865.86 | 1 | 4 |
| Eastern | 15670.32 | 3 | 13 |
| Midwest | 15905.21 | 2 | 9 |
| Western | 16875.99 | 2 | 9 |

The next query illustrates how multiple fullselects can be nested inside each other:

*Nested fullselects*

```
SELECT id
FROM (SELECT *
      FROM (SELECT id, years, salary
            FROM (SELECT *
                  FROM (SELECT *
                        FROM staff
                        WHERE dept < 77) AS t1
                  WHERE id < 300) AS t2
            WHERE job LIKE 'C%') AS t3
      WHERE salary < 18000) AS t4
WHERE years < 5;
```

*ANSWER*

| ID |
|---|
| 170 |

| ID |
|---|
| 180 |
| 230 |

A very common usage of a fullselect is to join a derived table to a real table. In the following example, the average salary for each department is joined to the individual staff row:

*Join fullselect to real table*

```
SELECT a.id
     , a.dept
     , a.salary
     , DEC(b.avgsal,7,2) AS avg_dept
FROM staff a
LEFT OUTER JOIN (SELECT dept        AS dept
                      , AVG(salary) AS avgsal
                 FROM staff
                 GROUP BY dept
                 HAVING AVG(salary) > 16000) AS b
ON a.dept = b.dept
WHERE a.id < 40
ORDER BY a.id;
```

*ANSWER*

| ID | DEPT | SALARY | AVG_DEPT |
|---|---|---|---|
| 10 | 20 | 18357.50 | 16071.52 |
| 20 | 20 | 78171.25 | 16071.52 |
| 30 | 38 | 77506.75 | |

# 16.8. Table Function Usage

If the fullselect query has a reference to a row in a table that is outside of the fullselect, then it needs to be written as a TABLE function call. In the next example, the preceding "A" table is referenced in the fullselect, and so the TABLE function call is required:

*Fullselect with external table reference*

```
SELECT a.id
     , a.dept
     , a.salary
     , b.deptsal
FROM staff a
, TABLE (SELECT b.dept
              , SUM(b.salary) AS deptsal
         FROM staff b
         WHERE b.dept = a.dept
         GROUP BY b.dept) AS b
WHERE a.id < 40
ORDER BY a.id;
```

*ANSWER*

| ID | DEPT | SALARY | DEPTSAL |
|---|---|---|---|
| 10 | 20 | 18357.50 | 64286.10 |
| 20 | 20 | 78171.25 | 64286.10 |
| 30 | 38 | 77506.75 | 77285.55 |

Below is the same query written without the reference to the "A" table in the fullselect, and thus without a TABLE function call:

*Fullselect without external table reference*

```
SELECT a.id
     , a.dept
     , a.salary
     , b.deptsal
FROM staff a
  , (SELECT b.dept
          , SUM(b.salary) AS deptsal
     FROM staff b
     GROUP BY b.dept) AS b
WHERE a.id < 40
AND b.dept = a.dept
ORDER BY a.id;
```

*ANSWER*

| ID | DEPT | SALARY | DEPTSAL |
|---|---|---|---|
| 10 | 20 | 18357.50 | 64286.10 |
| 20 | 20 | 78171.25 | 64286.10 |
| 30 | 38 | 77506.75 | 77285.55 |

Any externally referenced table in a fullselect must be defined in the query syntax (starting at the first FROM statement) before the fullselect. Thus, in the first example above, if the "A" table had been listed after the "B" table, then the query would have been invalid.

## 16.9. Full-Select in SELECT Phrase

A fullselect that returns a single column and row can be used in the SELECT part of a query:

*Use an uncorrelated Full-Select in a SELECT list*

```
SELECT id
     , salary
     , (SELECT MAX(salary)
        FROM staff) AS maxsal
FROM staff a
WHERE id < 60
ORDER BY id;
```

*ANSWER*

| ID | SALARY | MAXSAL |
|---|---|---|
| 10 | 18357.50 | 22959.20 |
| 20 | 78171.25 | 22959.20 |
| 30 | 77506.75 | 22959.20 |
| 40 | 18006.00 | 22959.20 |
| 50 | 20659.80 | 22959.20 |

A fullselect in the SELECT part of a statement must return only a single row, but it need not always be the same row. In the following example, the ID and SALARY of each employee is obtained - along with the max SALARY for the employee's department.

*Use a correlated Full-Select in a SELECT list*

```
SELECT id
     , salary
     , (SELECT MAX(salary)
        FROM staff b
        WHERE a.dept = b.dept) AS maxsal
FROM staff a
WHERE id < 60
ORDER BY id;
```

*ANSWER*

| ID | SALARY | MAXSAL |
|---|---|---|
| 10 | 18357.50 | 18357.50 |

| ID | SALARY | MAXSAL |
|---|---|---|
| 20 | 78171.25 | 18357.50 |
| 30 | 77506.75 | 18006.00 |
| 40 | 18006.00 | 18006.00 |
| 50 | 20659.80 | 20659.80 |

*Use correlated and uncorrelated Full-Selects in a SELECT list*

```
SELECT id
     , dept
     , salary
     , (SELECT MAX(salary)
        FROM staff b
        WHERE b.dept = a.dept)
     , (SELECT MAX(salary)
        FROM staff)
FROM staff a
WHERE id < 60
ORDER BY id;
```

*ANSWER*

| ID | DEPT | SALARY | 4 | 5 |
|---|---|---|---|---|
| 10 | 20 | 18357.50 | 18357.50 | 22959.20 |
| 20 | 20 | 78171.25 | 18357.50 | 22959.20 |
| 30 | 38 | 77506.75 | 18006.00 | 22959.20 |
| 40 | 38 | 18006.00 | 18006.00 | 22959.20 |
| 50 | 15 | 20659.80 | 20659.80 | 22959.20 |

## 16.10. INSERT Usage

The following query uses both an uncorrelated and correlated fullselect in the query that builds the set of rows to be inserted:

*Fullselect in INSERT*

```
INSERT INTO staff
SELECT id + 1
    , (SELECT MIN(name)
       FROM staff)
    , (SELECT dept
       FROM staff s2
       WHERE s2.id = s1.id - 100)
    , 'A', 1, 2, 3
FROM staff s1
WHERE id = (SELECT MAX(id)
            FROM staff);
```

# 16.11. UPDATE Usage

The following example uses an uncorrelated fullselect to assign a set of workers the average salary in the company - plus two thousand dollars.

*Use uncorrelated Full-Select to give workers company AVG salary (+$2000)*

```
UPDATE staff a
SET salary =
    (SELECT AVG(salary)+ 2000
     FROM staff)
WHERE id < 60;
```

*ANSWER*

| ID | DEPT | SALARY BEFORE | SALARY AFTER |
|---|---|---|---|
| 10 | 20 | 18357.50 | 18675.64 |
| 20 | 20 | 78171.25 | 18675.64 |
| 30 | 38 | 77506.75 | 18675.64 |
| 40 | 38 | 18006.00 | 18675.64 |
| 50 | 15 | 20659.80 | 18675.64 |

The next statement uses a correlated fullselect to assign a set of workers the average salary for their department - plus two thousand dollars. Observe that when there is more than one worker in the same department, that they all get the same new salary. This is because the fullselect is resolved before the first update was done, not after each.

*Use correlated Full-Select to give workers department AVG salary (+$2000)*

```
UPDATE staff a
SET salary =
    (SELECT AVG(salary) + 2000
     FROM staff b
     WHERE a.dept = b.dept)
WHERE id < 60;
```

*ANSWER*

| ID | DEPT | SALARY BEFORE | SALARY AFTER |
|---|---|---|---|
| 10 | 20 | 18357.50 | 18071.52 |
| 20 | 20 | 78171.25 | 18071.52 |
| 30 | 38 | 77506.75 | 17457.11 |
| 40 | 38 | 18006.00 | 17457.11 |
| 50 | 15 | 20659.80 | 17482.33 |

A fullselect is always resolved just once. If it is queried using a correlated expression, then the data returned each time may differ, but the table remains unchanged.

The next update is the same as the prior, except that two fields are changed:

*Update two fields by referencing Full-Select*

```
UPDATE staff a
SET (salary, years) =
    (SELECT AVG(salary) + 2000
          , MAX(years)
     FROM staff b
     WHERE a.dept = b.dept)
WHERE id < 60;
```

# 16.12. Declared Global Temporary Tables

If we want to temporarily retain some rows for processing by subsequent SQL statements, we can use a Declared Global Temporary Table. A temporary table only exists until the thread is terminated (or sooner). It is not defined in the Db2 catalogue, and neither its definition nor its contents are visible to other users. Multiple users can declare the same temporary table at the same time. Each will be independently working with their own copy.

**Usage Notes**

For a complete description of this feature, see the SQL reference. Below are some key points:

- The temporary table name can be any valid Db2 table name. The table qualifier, if provided, must be SESSION. If the qualifier is not provided, it is assumed to be SESSION.
- If the temporary table has been previously defined in this session, the WITH REPLACE clause can be used to override it. Alternatively, one can DROP the prior instance.
- An index can be defined on a global temporary table. The qualifier (i.e. SESSION) must be explicitly provided.
- Any column type can be used in the table, except for: BLOB, CLOB, DBCLOB, LONG VARCHAR, LONG VARGRAPHIC, DATALINK, reference, and structured data types.
- One can choose to preserve or delete (the default) the rows in the table when a commit occurs. Deleting the rows does not drop the table.
- Standard identity column definitions can be used if desired.
- Changes are not logged.

### 16.12.1. Sample SQL

Below is an example of declaring a global temporary table by listing the columns:

*Declare Global Temporary Table - define columns*

```
DECLARE GLOBAL TEMPORARY TABLE session.fred
( dept       SMALLINT  NOT NULL
, avg_salary DEC(7, 2) NOT NULL
, num_emps   SMALLINT  NOT NULL)
ON COMMIT DELETE ROWS;
```

In the next example, the temporary table is defined to have exactly the same columns as the existing STAFF table:

*Declare Global Temporary Table - like another table*

```
DECLARE GLOBAL TEMPORARY TABLE session.fred
LIKE staff INCLUDING COLUMN DEFAULTS
WITH REPLACE
ON COMMIT PRESERVE ROWS;
```

In the next example, the temporary table is defined to have a set of columns that are returned by a particular select statement. The statement is not actually run at definition time, so any predicates provided are irrelevant:

*Declare Global Temporary Table - like query output*

```
DECLARE GLOBAL TEMPORARY TABLE session.fred AS
(SELECT dept
      , MAX(id) AS max_id
      , SUM(salary) AS sum_sal
 FROM staff
 WHERE name <> 'IDIOT'
 GROUP BY dept)
DEFINITION ONLY
WITH REPLACE;
```

Indexes can be added to temporary tables in order to improve performance and/or to enforce uniqueness:

*Temporary table with index*

```
DECLARE GLOBAL TEMPORARY TABLE session.fred
LIKE staff
INCLUDING COLUMN DEFAULTS
WITH REPLACE ON COMMIT DELETE ROWS;

CREATE UNIQUE INDEX session.fredx ON Session.fred (id);

INSERT INTO session.fred
SELECT *
FROM staff
WHERE id < 200;

SELECT COUNT(*)
FROM session.fred; --> Returns 19

COMMIT;

SELECT COUNT(*)
FROM session.fred; --> Returns 0
```

A temporary table has to be dropped to reuse the same name:

*Dropping a temporary table*

```
DECLARE GLOBAL TEMPORARY TABLE session.fred
( dept SMALLINT NOT NULL
, avg_salary DEC(7, 2) NOT NULL
, num_emps SMALLINT NOT NULL)
ON COMMIT DELETE ROWS;

INSERT INTO session.fred
SELECT dept
     , AVG(salary)
     , COUNT(*)
FROM staff
GROUP BY dept;

SELECT COUNT(*)
FROM session.fred; --> Returns 8

DROP TABLE session.fred;

DECLARE GLOBAL TEMPORARY TABLE session.fred
(dept SMALLINT NOT NULL)
ON COMMIT DELETE ROWS;

SELECT COUNT(*)
FROM session.fred; --> Return 0
```

## 16.13. Tablespace

Before a user can create a declared global temporary table, a USER TEMPORARY tablespace that they have access to, has to be created. A typical definition follows:

*Create USER TEMPORARY tablespace*

```
CREATE USER TEMPORARY TABLESPACE FRED
MANAGED BY DATABASE
USING (FILE 'C:\Db2\TEMPFRED\FRED1' 1000
,FILE 'C:\Db2\TEMPFRED\FRED2' 1000
,FILE 'C:\Db2\TEMPFRED\FRED3' 1000);

GRANT USE OF TABLESPACE FRED TO PUBLIC;
```

## 16.14. Do NOT use to Hold Output

In general, do not use a Declared Global Temporary Table to hold job output data, especially if the table is defined ON COMMIT PRESERVE ROWS. If the job fails halfway through, the contents of the temporary table will be lost. If, prior to the failure, the job had updated and then committed Production data, it may be impossible to recreate the lost output because the committed rows

cannot be updated twice.

# Chapter 17. Recursive SQL

Recursive SQL enables one to efficiently resolve all manner of complex logical structures that can be really tough to work with using other techniques. On the down side, it is a little tricky to understand at first and it is occasionally expensive. In this chapter we shall first show how recursive SQL works and then illustrate some of the really cute things that one use it for.

Use Recursion To

- Create sample data.
- Select the first "n" rows.
- Generate a simple parser.
- Resolve a Bill of Materials hierarchy.
- Normalize and/or denormalize data structures.

When (Not) to Use Recursion

A good SQL statement is one that gets the correct answer, is easy to understand, and is efficient. Let us assume that a particular statement is correct. If the statement uses recursive SQL, it is never going to be categorized as easy to understand (though the reading gets much easier with experience). However, given the question being posed, it is possible that a recursive SQL statement is the simplest way to get the required answer. Recursive SQL statements are neither inherently efficient nor inefficient. Because they often involve a join, it is very important that suitable indexes be provided. Given appropriate indexes, it is quite probable that a recursive SQL statement is the most efficient way to resolve a particular business problem. It all depends upon the nature of the question: If every row processed by the query is required in the answer set (e.g. Find all people who work for Bob), then a recursive statement is likely to very efficient. If only a few of the rows processed by the query are actually needed (e.g. Find all airline flights from Boston to Dallas, then show only the five fastest) then the cost of resolving a large data hierarchy (or network), most of which is immediately discarded, can be very prohibitive. If one wants to get only a small subset of rows in a large data structure, it is very important that of the unwanted data is excluded as soon as possible in the processing sequence. Some of the queries illustrated in this chapter have some rather complicated code in them to do just this. Also, always be on the lookout for infinitely looping data structures.

Conclusion

Recursive SQL statements can be very efficient, if coded correctly, and if there are suitable indexes. When either of the above is not true, they can be very slow.

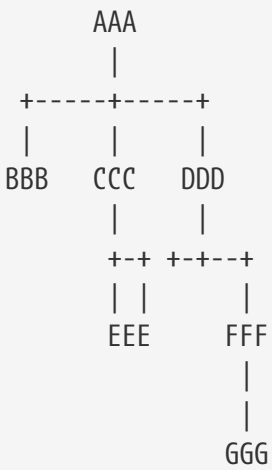
## 17.1. How Recursion Works

Below is a description of a very simple application. The table on the left contains a normalized representation of the hierarchical structure on the right. Each row in the table defines a relationship displayed in the hierarchy. The PKEY field identifies a parent key, the CKEY field has related child keys, and the NUM field has the number of times the child occurs within the related parent.

*HIERARCHY*

| PKEY | CKEY | NUM |
|---|---|---|
| AAA | BBB | 1 |
| AAA | CCC | 5 |
| AAA | DDD | 20 |
| CCC | EEE | 33 |
| DDD | EEE | 44 |
| DDD | FFF | 5 |
| FFF | GGG | 5 |

*Sample Table description - Recursion*




### 17.1.1. List Dependents of AAA

We want to use SQL to get a list of all the dependents of AAA. This list should include not only those items like CCC that are directly related, but also values such as GGG, which are indirectly related. The easiest way to answer this question (in SQL) is to use a recursive SQL statement that goes thus:

*SQL that does Recursion*

```
WITH parent (pkey, ckey) AS
(SELECT pkey
      , ckey
 FROM hierarchy
 WHERE pkey = 'AAA'
   UNION ALL
 SELECT C.pkey
      , C.ckey
 FROM hierarchy C
    , parent P
 WHERE P.ckey = C.pkey)
SELECT pkey
     , ckey
FROM parent;
```

*ANSWER*

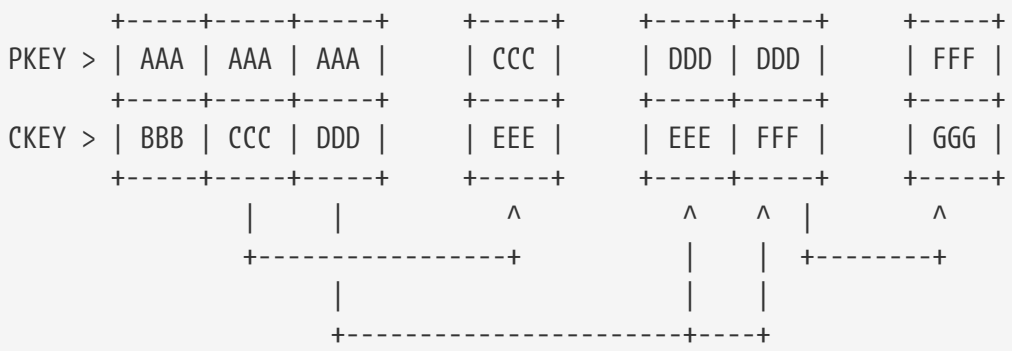
| PKEY | CKEY | PROCESSING SEQUENCE |
|---|---|---|
| AAA | BBB | 1st pass |
| AAA | CCC | 1st pass |
| AAA | DDD | 1st pass |
| CCC | EEE | 2nd pass |
| DDD | EEE | 3rd pass |
| DDD | FFF | 3rd pass |
| FFF | GGG | 4th pass |

The above statement is best described by decomposing it into its individual components, and then following of sequence of events that occur:

- The WITH statement at the top defines a temporary table called PARENT. The upper part of the UNION ALL is only invoked once. It does an initial population of the PARENT table with the three rows that have an immediate parent key of AAA.
- The lower part of the UNION ALL is run recursively until there are no more matches to the join. In the join, the current child value in the temporary PARENT table is joined to related parent values in the DATA table. Matching rows are placed at the front of the temporary PARENT table. This recursive processing will stop when all of the rows in the PARENT table have been joined to the DATA table.
- The SELECT phrase at the bottom of the statement sends the contents of the PARENT table back to the user's program.
- Another way to look at the above process is to think of the temporary PARENT table as a stack of data. This stack is initially populated by the query in the top part of the UNION ALL.
- Next, a cursor starts from the bottom of the stack and goes up. Each row obtained by the cursor

is joined to the DATA table. Any matching rows obtained from the join are added to the top of the stack (i.e. in front of the cursor). When the cursor reaches the top of the stack, the statement is done. The following diagram illustrates this process:

*Recursive processing sequence*




## 17.1.2. Notes & Restrictions

- Recursive SQL requires that there be a UNION ALL phrase between the two main parts of the statement. The UNION ALL, unlike the UNION, allows for duplicate output rows, which is what often comes out of recursive processing.
- If done right, recursive SQL is often fairly efficient. When it involves a join similar to the example shown above, it is important to make sure that this join is efficient. To this end, suitable indexes should be provided.
- The output of a recursive SQL is a temporary table (usually). Therefore, all temporary table usage restrictions also apply to recursive SQL output. See the section titled "Common Table Expression" for details.
- The output of one recursive expression can be used as input to another recursive expression in the same SQL statement. This can be very handy if one has multiple logical hierarchies to traverse (e.g. First find all of the states in the USA, then final all of the cities in each state).
- Any recursive coding, in any language, can get into an infinite loop - either because of bad coding, or because the data being processed has a recursive value structure. To prevent your SQL running forever, see the section titled Halting Recursive Processing.

## 17.1.3. Sample Table DDL & DML

*Sample Table DDL - Recursion*

```
CREATE TABLE hierarchy
( pkey CHAR(03) NOT NULL
, ckey CHAR(03) NOT NULL
, num SMALLINT NOT NULL
, PRIMARY KEY(pkey, ckey)
, CONSTRAINT dt1 CHECK (pkey <> ckey)
, CONSTRAINT dt2 CHECK (num > 0));

COMMIT;

CREATE UNIQUE INDEX hier_x1
ON hierarchy (ckey, pkey);

COMMIT;

INSERT INTO hierarchy VALUES
    ('AAA','BBB', 1)
  , ('AAA','CCC', 5)
  , ('AAA','DDD',20)
  , ('CCC','EEE',33)
  , ('DDD','EEE',44)
  , ('DDD','FFF', 5)
  , ('FFF','GGG', 5);

COMMIT;
```

# 17.2. Introductory Recursion

This section will use recursive SQL statements to answer a series of simple business questions using the sample HIERARCHY table described on Sample Table DDL - Recursion. Be warned that things are going to get decidedly more complex as we proceed.

## 17.2.1. List all Children #1

Find all the children of AAA. Don't worry about getting rid of duplicates, sorting the data, or any other of the finer details.

*List of children of AAA*

```
WITH parent (ckey) AS
(SELECT ckey
 FROM hierarchy
 WHERE pkey = 'AAA'
   UNION ALL
 SELECT C.ckey
 FROM hierarchy C
    , parent P
 WHERE P.ckey = C.pkey)
SELECT ckey
FROM parent;
```

*ANSWER*

| CKEY |
|---|
| BBB |
| CCC |
| DDD |
| EEE |
| EEE |
| FFF |
| GGG |

*HIERARCHY*

| PKEY | CKEY | NUM |
|---|---|---|
| AAA | BBB | 1 |
| AAA | CCC | 5 |
| AAA | DDD | 20 |
| CCC | EEE | 33 |
| DDD | EEE | 44 |
| DDD | FFF | 5 |
| FFF | GGG | 5 |

Much of the SQL shown in this section will loop forever if the target database has a recursive data structure. See Halting Recursive Processing for details on how to prevent this.

The above SQL statement uses standard recursive processing. The first part of the UNION ALL seeds the temporary table PARENT. The second part recursively joins the temporary table to the source data table until there are no more matches. The final part of the query displays the result set.

Imagine that the HIERARCHY table used above is very large and that we also want the above query to be as efficient as possible. In this case, two indexes are required; The first, on PKEY, enables the initial select to run efficiently. The second, on CKEY, makes the join in the recursive part of the query efficient. The second index is arguably more important than the first because the first is only used once, whereas the second index is used for each child of the toplevel parent.

### 17.2.2. List all Children #2

Find all the children of AAA, include in this list the value AAA itself. To satisfy the latter requirement we will change the first SELECT statement (in the recursive code) to select the parent itself instead of the list of immediate children. A DISTINCT is provided in order to ensure that only one line containing the name of the parent (i.e. "AAA") is placed into the temporary PARENT table. **NOTE**: _Before the introduction of recursive SQL processing, it often made sense to define the top-most level in a hierarchical data structure as being a parent-child of itself. For example, the HIERARCHY table might contain a row indicating that "AAA" is a child of "AAA". If the target table has data like this, add another predicate: `C.PKEY <> C.CKEY` to the recursive part of the SQL statement to stop the query from looping forever.

*List all children of AAA*

```
WITH parent (ckey) AS
(SELECT DISTINCT pkey
 FROM hierarchy
 WHERE pkey = 'AAA'
   UNION ALL
 SELECT C.ckey
 FROM hierarchy C
    , parent P
 WHERE P.ckey = C.pkey)
SELECT ckey
FROM parent;
```

*ANSWER*

| CKEY |
|---|
| AAA |
| BBB |
| CCC |
| DDD |
| EEE |
| EEE |
| FFF |
| GGG |

*HIERARCHY*

| PKEY | CKEY | NUM |
|---|---|---|
| AAA | BBB | 1 |
| AAA | CCC | 5 |
| AAA | DDD | 20 |
| CCC | EEE | 33 |
| DDD | EEE | 44 |
| DDD | FFF | 5 |
| FFF | GGG | 5 |

In most, but by no means all, business situations, the above SQL statement is more likely to be what the user really wanted than the SQL before. Ask before you code.

### 17.2.3. List Distinct Children

Get a distinct list of all the children of AAA. This query differs from the prior only in the use of the DISTINCT phrase in the final select.

*List distinct children of AAA*

```
WITH parent (ckey) AS
(SELECT DISTINCT pkey
 FROM hierarchy
 WHERE pkey = 'AAA'
   UNION ALL
 SELECT C.ckey
 FROM hierarchy C
    , parent P
 WHERE P.ckey = C.pkey)
SELECT DISTINCT ckey
FROM parent;
```

*ANSWER*

| CKEY |
|---|
| AAA |
| BBB |
| CCC |
| DDD |
| EEE |
| FFF |
| GGG |

*HIERARCHY*

| PKEY | CKEY | NUM |
|---|---|---|
| AAA | BBB | 1 |
| AAA | CCC | 5 |
| AAA | DDD | 20 |
| CCC | EEE | 33 |
| DDD | EEE | 44 |
| DDD | FFF | 5 |
| FFF | GGG | 5 |

The next thing that we want to do is build a distinct list of children of AAA that we can then use to join to other tables. To do this, we simply define two temporary tables. The first does the recursion and is called PARENT. The second, called DISTINCT_PARENT, takes the output from the first and removes duplicates.

*List distinct children of AAA*

```
WITH parent (ckey) AS
(SELECT DISTINCT pkey
 FROM hierarchy
 WHERE pkey = 'AAA'
   UNION ALL
 SELECT C.ckey
 FROM hierarchy C
    , parent P
 WHERE P.ckey = C.pkey)
, distinct_parent (ckey) AS
(SELECT DISTINCT ckey
 FROM parent)
SELECT ckey
FROM distinct_parent;
```

*ANSWER*

| CKEY |
|---|
| AAA |
| BBB |
| CCC |
| DDD |
| EEE |
| FFF |
| GGG |

*HIERARCHY*

| PKEY | CKEY | NUM |
|---|---|---|
| AAA | BBB | 1 |
| AAA | CCC | 5 |
| AAA | DDD | 20 |
| CCC | EEE | 33 |
| DDD | EEE | 44 |
| DDD | FFF | 5 |
| FFF | GGG | 5 |

### 17.2.4. Show Item Level

Get a list of all the children of AAA. For each value returned, show its level in the logical hierarchy relative to AAA.

*Show item level in hierarchy*

```
WITH parent (ckey, lvl) AS
(SELECT DISTINCT pkey
      , 0
 FROM hierarchy
 WHERE pkey = 'AAA'
   UNION ALL
 SELECT C.ckey
      , P.lvl +1
 FROM hierarchy C
    , parent P
 WHERE P.ckey = C.pkey)
SELECT ckey
     , lvl
FROM parent;
```

*ANSWER*

| CKEY | LVL |
|---|---|
| AAA | 0 |
| BBB | 1 |
| CCC | 1 |
| DDD | 1 |
| EEE | 2 |
| EEE | 2 |
| FFF | 2 |
| GGG | 3 |

The above statement has a derived integer field called LVL. In the initial population of the temporary table this level value is set to zero. When subsequent levels are reached, this value in incremented by one.

### 17.2.5. Select Certain Levels

Get a list of all the children of AAA that are less than three levels below AAA.

*Select rows where LEVEL < 3*

```
WITH parent (ckey, lvl) AS
(SELECT DISTINCT pkey
      , 0
 FROM hierarchy
 WHERE pkey = 'AAA'
   UNION ALL
 SELECT C.ckey
      , P.lvl +1
 FROM hierarchy C
    , parent P
 WHERE P.ckey = C.pkey)
SELECT ckey, lvl
FROM parent
WHERE lvl < 3;
```

*ANSWER*

| CKEY | LVL |
|---|---|
| AAA | 0 |
| BBB | 1 |
| CCC | 1 |
| DDD | 1 |
| EEE | 2 |
| EEE | 2 |
| FFF | 2 |

*HIERARCHY*

| PKEY | CKEY | NUM |
|---|---|---|
| AAA | BBB | 1 |
| AAA | CCC | 5 |
| AAA | DDD | 20 |
| CCC | EEE | 33 |
| DDD | EEE | 44 |

| PKEY | CKEY | NUM |
|---|---|---|
| DDD | FFF | 5 |
| FFF | GGG | 5 |

The above statement has two main deficiencies:

- It will run forever if the database contains an infinite loop.
- It may be inefficient because it resolves the whole hierarchy before discarding those levels that are not required.

To get around both of these problems, we can move the level check up into the body of the recursive statement. This will stop the recursion from continuing as soon as we reach the target level. We will have to add "+ 1" to the check to make it logically equivalent:

*Select rows where LEVEL < 3*

```
WITH parent (ckey, lvl) AS
(SELECT DISTINCT pkey
      , 0
 FROM hierarchy
 WHERE pkey = 'AAA'
   UNION ALL
 SELECT C.ckey
      , P.lvl +1
 FROM hierarchy C
    , parent P
 WHERE P.ckey = C.pkey
 AND P.lvl+1 < 3)
SELECT ckey, lvl
FROM parent;
```

*ANSWER*

| CKEY | LVL |
|---|---|
| AAA | 0 |
| BBB | 1 |
| CCC | 1 |
| DDD | 1 |
| EEE | 2 |
| EEE | 2 |
| FFF | 2 |

The only difference between this statement and the one before is that the level check is now done in the recursive part of the statement. This new level-check predicate has a dual function: It gives us the answer that we want, and it stops the SQL from running forever if the database happens to

contain an infinite loop (e.g. DDD was also a parent of AAA). One problem with this general statement design is that it can not be used to list only that data which pertains to a certain lower level (e.g. display only level 3 data). To answer this kind of question efficiently we can combine the above two queries, having appropriate predicates in both places (see next).

### 17.2.6. Select Explicit Level

Get a list of all the children of AAA that are exactly two levels below AAA.

*Select rows where LEVEL = 2*

```
WITH parent (ckey, lvl) AS
(SELECT DISTINCT pkey
      , 0
 FROM hierarchy
 WHERE pkey = 'AAA'
   UNION ALL
 SELECT C.ckey
      , P.lvl +1
FROM hierarchy C
   , parent P
 WHERE P.ckey  = C.pkey
 AND   P.lvl+1 < 3)
SELECT ckey
     , lvl
FROM parent
WHERE lvl = 2;
```

*ANSWER*

| CKEY | LVL |
|---|---|
| EEE | 2 |
| EEE | 2 |
| FFF | 2 |

*HIERARCHY*

| PKEY | CKEY | NUM |
|---|---|---|
| AAA | BBB | 1 |
| AAA | CCC | 5 |
| AAA | DDD | 20 |
| CCC | EEE | 33 |
| DDD | EEE | 44 |
| DDD | FFF | 5 |
| FFF | GGG | 5 |

In the recursive part of the above statement all of the levels up to and including that which is required are obtained. All undesired lower levels are then removed in the final select.

### 17.2.7. Trace a Path - Use Multiple Recursions

Multiple recursive joins can be included in a single query. The joins can run independently, or the output from one recursive join can be used as input to a subsequent. Such code enables one to do the following:

- Expand multiple hierarchies in a single query. For example, one might first get a list of all departments (direct and indirect) in a particular organization, and then use the department list as a seed to find all employees (direct and indirect) in each department.
- Go down, and then up, a given hierarchy in a single query. For example, one might want to find all of the children of AAA, and then all of the parents. The combined result is the list of objects that AAA is related to via a direct parent-child path.
- Go down the same hierarchy twice, and then combine the results to find the matches, or the non-matches. This type of query might be used to, for example, see if two companies own shares in the same subsidiary.
- The next example recursively searches the HIERARCHY table for all values that are either a child or a parent (direct or indirect) of the object DDD. The first part of the query gets the list of children, the second part gets the list of parents (but never the value DDD itself), and then the results are combined.

*Find all children and parents of DDD*

```
WITH children (kkey, lvl)
AS (SELECT ckey
          , 1
    FROM hierarchy
    WHERE pkey = 'DDD'
      UNION ALL
    SELECT H.ckey
          , C.lvl + 1
    FROM hierarchy H
       , children  C
    WHERE H.pkey = C.kkey)
, parents (kkey, lvl)
AS (SELECT pkey ,
           -1
    FROM hierarchy
    WHERE ckey = 'DDD'
      UNION ALL
    SELECT H.pkey
          , P.lvl - 1
    FROM hierarchy H
       , parents P
    WHERE H.ckey = P.kkey)
SELECT kkey
     , lvl
FROM children
  UNION ALL
SELECT kkey
     , lvl
FROM parents;
```

*ANSWER*

| KKEY | LVL |
|---|---|
| AAA | -1 |
| EEE | 1 |
| FFF | 1 |
| GGG | 2 |

### 17.2.8. Extraneous Warning Message

Some recursive SQL statements generate the following warning when the Db2 parser has reason to suspect that the statement may run forever:

```
SQL0347W The recursive common table expression "GRAEME.TEMP1" may contain an infinite
loop. SQLSTATE=01605
```

The text that accompanies this message provides detailed instructions on how to code recursive SQL so as to avoid getting into an infinite loop. The trouble is that even if you do exactly as told you may still get the silly message. To illustrate, the following two SQL statements are almost identical. Yet the first gets a warning and the second does not:

*Recursion - with warning message*

```
WITH temp1 (n1) AS
(SELECT id
 FROM staff
 WHERE id = 10
   UNION ALL
 SELECT n1 +10
 FROM temp1
 WHERE n1 < 50)
SELECT *
FROM temp1;
```

*ANSWER*

| N1 |
|---|
| warn |
| 10 |
| 20 |
| 30 |
| 40 |
| 50 |

*Recursion - without warning message*

```
WITH temp1 (n1) AS
(SELECT INT(id)
 FROM staff
 WHERE id = 10
   UNION ALL
 SELECT n1 +10
 FROM temp1
 WHERE n1 < 50)
SELECT *
FROM temp1;
```

*ANSWER*

| N1 |
|---|
| 10 |

| N1 |
|---|
| 20 |
| 30 |
| 40 |
| 50 |

If you know what you are doing, ignore the message.

### 17.2.9. Logical Hierarchy Flavours

Before getting into some of the really nasty stuff, we best give a brief overview of the various kinds of logical hierarchy that exist in the real world and how each is best represented in a relational database. Some typical data hierarchy flavours are shown below. Note that the three on the left form one, mutually exclusive, set and the two on the right another. Therefore, it is possible for a particular hierarchy to be both divergent and unbalanced (or balanced), but not both divergent and convergent.

*Hierarchy Flavours*

```
DIVERGENT CONVERGENT RECURSIVE BALANCED    UNBALANCED
========= ========== ========= ========    ==========
  AAA       AAA       AAA<--+      AAA         AAA
   |         |         |    |       |           |
 +-+-+     +-+-+     +-+-+  |     +-+-+       +-+-+
 |   |     |   |     |   |  |     |   |       |   |
BBB CCC   BBB CCC   BBB CCC>+    BBB CCC     BBB CCC
     |     |   |         |        |   |           |
   +-+-+   +-+-+-+     +-+-+      |   +---+     +-+-+
   |   |     |   |     |   |      |   |   |     |   |
  DDD EEE   DDD EEE   DDD EEE    DDD EEE FFF   DDD EEE
```

### 17.2.10. Divergent Hierarchy

In this flavour of hierarchy, no object has more than one parent. Each object can have none, one, or more than one, dependent child objects. Physical objects (e.g. Geographic entities) tend to be represented in this type of hierarchy. This type of hierarchy will often incorporate the concept of different layers in the hierarchy referring to differing kinds of object - each with its own set of attributes. For example, a Geographic hierarchy might consist of countries, states, cities, and street addresses. A single table can be used to represent this kind of hierarchy in a fully normalized form. One field in the table will be the unique key, another will point to the related parent. Other fields in the table may pertain either to the object in question, or to the relationship between the object and its parent. For example, in the following table the PRICE field has the price of the object, and the NUM field has the number of times that the object occurs in the parent.

*OBJECTS_RELATES*

*Table 13. Divergent Hierarchy - Table and Layout*

| KEYO | PKEY | NUM | PRICE |
|---|---|---|---|
| AAA | | | $10 |
| BBB | AAA | 1 | $21 |
| CCC | AAA | 5 | $23 |
| DDD | AAA | 20 | $25 |
| EEE | DDD | 44 | $33 |
| FFF | DDD | 5 | $34 |
| GGG | FFF | 5 | $44 |

Some database designers like to make the arbitrary judgment that every object has a parent, and in those cases where there is no "real" parent, the object considered to be a parent of itself. In the above table, this would mean that AAA would be defined as a parent of AAA. Please appreciate that this judgment call does not affect the objects that the database represents, but it can have a dramatic impact on SQL usage and performance. Prior to the introduction of recursive SQL, defining top level objects as being self-parenting was sometimes a good idea because it enabled one to resolve a hierarchy using a simple join without unions. This same process is now best done with recursive SQL. Furthermore, if objects in the database are defined as self-parenting, the recursive SQL will get into an infinite loop unless extra predicates are provided.

### 17.2.11. Convergent Hierarchy

**NUMBER OF TABLES**: A convergent hierarchy has many-to-many relationships that require two tables for normalized data storage. The other hierarchy types require but a single table.

In this flavour of hierarchy, each object can have none, one, or more than one, parent and/or dependent child objects.

Convergent hierarchies are often much more difficult to work with than similar divergent hierarchies. Logical entities, or man-made objects, (e.g. Company Divisions) often have this type of hierarchy. Two tables are required in order to represent this kind of hierarchy in a fully normalized form. One table describes the object, and the other describes the relationships between the objects.

*Convergent Hierarchy - Tables and Layout*

*OBJECTS*

| KEYO | PRICE |
|---|---|
| AAA | $10 |
| BBB | $21 |
| CCC | $23 |
| DDD | $25 |
| EEE | $33 |
| FFF | $34 |
| GGG | $44 |

*RELATIONSHIPS*

| PKEY | CKEY | NUM |
|---|---|---|
| AAA | BBB | 1 |
| AAA | CCC | 5 |
| AAA | DDD | 20 |
| CCC | EEE | 33 |
| DDD | EEE | 44 |
| DDD | FFF | 5 |
| FFF | GGG | 5 |

One has to be very careful when resolving a convergent hierarchy to get the answer that the user actually wanted. To illustrate, if we wanted to know how many children AAA has in the above structure the "correct" answer could be six, seven, or eight. To be precise, we would need to know if EEE should be counted twice and if AAA is considered to be a child of itself.

### 17.2.12. Recursive Hierarchy

Recursive data hierarchies will cause poorly written recursive SQL statements to run forever. See the section titled Halting Recursive Processing for details on how to prevent this, and how to check that a hierarchy is not recursive.

In this flavour of hierarchy, each object can have none, one, or more than one parent. Also, each object can be a parent and/or a child of itself via another object, or via itself directly. In the business world, this type of hierarchy is almost always wrong. When it does exist, it is often because a standard convergent hierarchy has gone a bit haywire. This database design is exactly the same as the one for a convergent hierarchy. Two tables are (usually) required in order to represent the hierarchy in a fully normalized form. One table describes the object, and the other describes the relationships between the objects.

*Recursive Hierarchy - Tables and Layout*

*OBJECTS*

| KEYO | PRICE |
|---|---|
| AAA | $10 |
| BBB | $21 |
| CCC | $23 |
| DDD | $25 |
| EEE | $33 |
| FFF | $34 |
| GGG | $44 |

*RELATIONSHIPS*

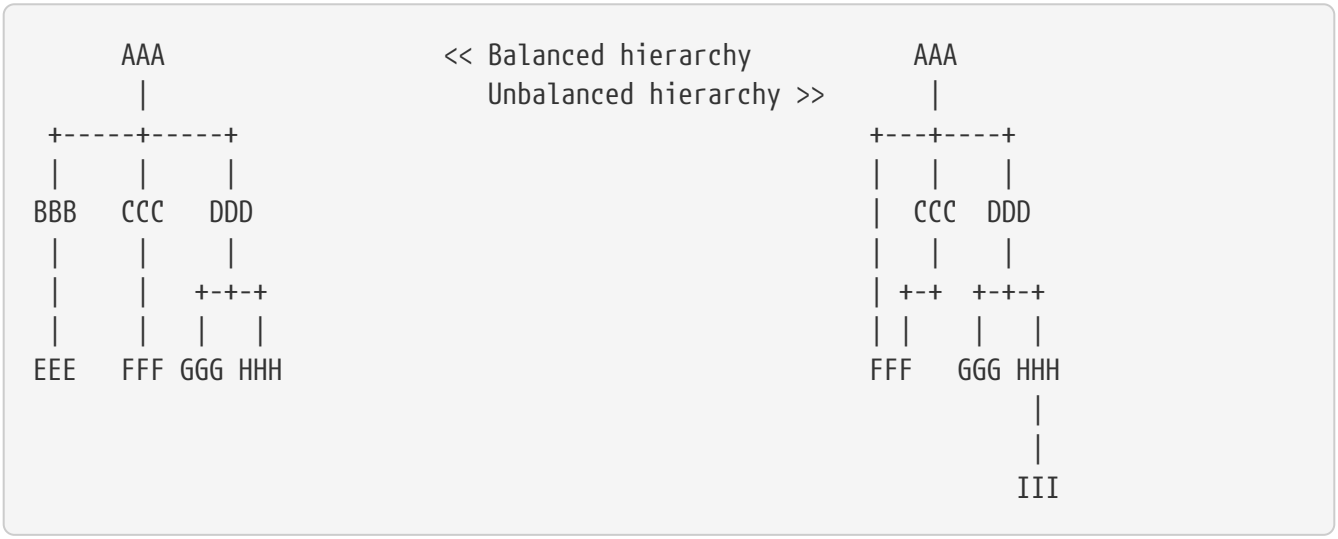
| PKEY | CKEY | NUM |
|---|---|---|
| AAA | BBB | 1 |
| AAA | CCC | 5 |
| AAA | DDD | 20 |
| CCC | EEE | 33 |
| DDD | AAA | 99 |
| DDD | FFF | 5 |
| DDD | EEE | 44 |
| FFF | GGG | 5 |

Prior to the introduction of recursive SQL, it took some non-trivial coding root out recursive data structures in convergent hierarchies. Now it is a no-brainer, see Halting Recursive Processing for details.

## 17.2.13. Balanced & Unbalanced Hierarchies

In some logical hierarchies the distance, in terms of the number of intervening levels, from the top parent entity to its lowest-level child entities is the same for all legs of the hierarchy. Such a hierarchy is considered to be balanced. An unbalanced hierarchy is one where the distance from a top-level parent to a lowest-level child is potentially different for each leg of the hierarchy.

*Balanced and Unbalanced Hierarchies*




Balanced hierarchies often incorporate the concept of levels, where a level is a subset of the values in the hierarchy that are all of the same time and are also the same distance from the top level parent. For example, in the balanced hierarchy above each of the three levels shown might refer to a different category of object (e.g. country, state, city). By contrast, in the unbalanced hierarchy above is probable that the objects being represented are all of the same general category (e.g. companies that own other companies). Divergent hierarchies are the most likely to be balanced. Furthermore, balanced and/or divergent hierarchies are the kind that are most often used to do

data summation at various intermediate levels. For example, a hierarchy of countries, states, and cities, is likely to be summarized at any level.

## 17.2.14. Data & Pointer Hierarchies

The difference between a data and a pointer hierarchy is not one of design, but of usage. In a pointer schema, the main application tables do not store a description of the logical hierarchy. Instead, they only store the base data. Separate to the main tables are one, or more, related tables that define which hierarchies each base data row belongs to.

Typically, in a pointer hierarchy, the main data tables are much larger and more active than the hierarchical tables. A banking application is a classic example of this usage pattern. There is often one table that contains core customer information and several related tables that enable one to do analysis by customer category. A data hierarchy is an altogether different beast. An example would be a set of tables that contain information on all that parts that make up an aircraft. In this kind of application the most important information in the database is often that which pertains to the relationships between objects. These tend to be very complicated often incorporating the attributes: quantity, direction, and version. Recursive processing of a data hierarchy will often require that one does a lot more than just find all dependent keys. For example, to find the gross weight of an aircraft from such a database one will have to work with both the quantity and weight of all dependent objects. Those objects that span sub-assembles (e.g. a bolt connecting to engine to the wing) must not be counted twice, missed out, nor assigned to the wrong sub-grouping. As always, such questions are essentially easy to answer, the trick is to get the right answer.

## 17.2.15. Halting Recursive Processing

One occasionally encounters recursive hierarchical data structures (i.e. where the parent item points to the child, which then points back to the parent). This section describes how to write recursive SQL statements that can process such structures without running forever. There are three general techniques that one can use: * Stop processing after reaching a certain number of levels. * Keep a record of where you have been, and if you ever come back, either fail or in some other way stop recursive processing. * Keep a record of where you have been, and if you ever come back, simply ignore that row and keep on resolving the rest of hierarchy.

#### Sample Table DDL & DML

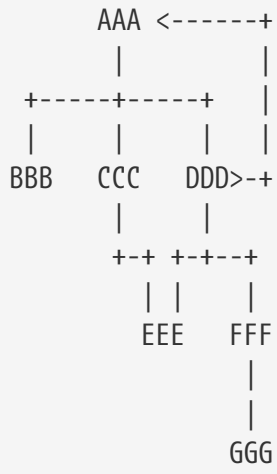
The following table is a normalized representation of the recursive hierarchy on the right. Note that AAA and DDD are both a parent and a child of each other.

*Recursive Hierarchy - Sample Table and Layout*

*TROUBLE*

| PKEY | CKEY |
|---|---|
| AAA | BBB |
| AAA | CCC |
| AAA | DDD |
| CCC | EEE |

| PKEY | CKEY |
|---|---|
| DDD | AAA |
| DDD | FFF |
| DDD | EEE |
| FFF | GGG |




Below is the DDL and DML that was used to create the above table.

*Sample Table DDL - Recursive Hierarchy*

```
CREATE TABLE trouble
( pkey CHAR(03) NOT NULL
, ckey CHAR(03) NOT NULL);

CREATE UNIQUE INDEX tble_x1 ON trouble (pkey, ckey);
CREATE UNIQUE INDEX tble_x2 ON trouble (ckey, pkey);

INSERT INTO trouble VALUES
    ('AAA','BBB')
  , ('AAA','CCC')
  , ('AAA','DDD')
  , ('CCC','EEE')
  , ('DDD','AAA')
  , ('DDD','EEE')
  , ('DDD','FFF')
  , ('FFF','GGG');
```

## 17.2.16. Other Loop Types

In the above table, the beginning object (i.e. AAA) is part of the data loop. This type of loop can be detected using simpler SQL than what is given here. But a loop that does not include the beginning object (e.g. AAA points to BBB, which points to CCC, which points back to BBB) requires the somewhat complicated SQL that is used in this section.

### Stop After "n" Levels

Find all the children of AAA. In order to avoid running forever, stop after four levels.

*Stop Recursive SQL after "n" levels*

```
WITH parent (pkey, ckey, lvl) AS
(SELECT DISTINCT pkey
      , pkey
      , 0
 FROM trouble
 WHERE pkey = 'AAA'
   UNION ALL
 SELECT C.pkey
      , C.ckey
      , P.lvl + 1
 FROM trouble C
    , parent P
WHERE P.ckey = C.pkey
AND P.lvl + 1 < 4)
SELECT *
FROM parent;
```

*ANSWER*

| PKEY | CKEY | LVL |
|---|---|---|
| AAA | AAA | 0 |
| AAA | BBB | 1 |
| AAA | CCC | 1 |
| AAA | DDD | 1 |
| CCC | EEE | 2 |
| DDD | AAA | 2 |
| DDD | EEE | 2 |
| DDD | FFF | 2 |
| AAA | BBB | 3 |
| AAA | CCC | 3 |
| AAA | DDD | 3 |
| FFF | GGG | 3 |

*TROUBLE*

| PKEY | CKEY |
|---|---|
| AAA | BBB |

| PKEY | CKEY |
|---|---|
| AAA | CCC |
| AAA | DDD |
| CCC | EEE |
| DDD | AAA |
| DDD | FFF |
| DDD | EEE |
| FFF | GGG |

In order for the above statement to get the right answer, we need to know before beginning the maximum number of valid dependent levels (i.e. non-looping) there are in the hierarchy. This information is then incorporated into the recursive predicate (see: P.LVI + 1 < 4). If the number of levels is not known, and we guess wrong, we may not find all the children of AAA. For example, if we had stopped at "2" in the above query, we would not have found the child GGG. A more specific disadvantage of the above statement is that the list of children contains duplicates. These duplicates include those specific values that compose the infinite loop (i.e. AAA and DDD), and also any children of either of the above.

#### Stop When Loop Found

A far better way to stop recursive processing is to halt when, and only when, we determine that we have been to the target row previously. To do this, we need to maintain a record of where we have been, and then check this record against the current key value in each row joined to. Db2 does not come with an in-built function that can do this checking, so we shall define our own.

#### #Define Function

Below is the definition code for a user-defined Db2 function that is very similar to the standard LOCATE function. It searches for one string in another, block by block. For example, if one was looking for the string "ABC", this function would search the first three bytes, then the next three bytes, and so on. If a match is found, the function returns the relevant block number, else zero.

*LOCATE_BLOCK user defined function*

```
CREATE FUNCTION LOCATE_BLOCK(searchstr VARCHAR(30000)
                           , lookinstr VARCHAR(30000))
RETURNS INTEGER
BEGIN ATOMIC
  DECLARE lookinlen, searchlen INT;
  DECLARE locatevar, returnvar INT DEFAULT 0;
  DECLARE beginlook
  INT DEFAULT 1;
  SET lookinlen = LENGTH(lookinstr);
  SET searchlen = LENGTH(searchstr);
  WHILE locatevar = 0 AND beginlook <= lookinlen
  DO
    SET locatevar = LOCATE(searchstr,SUBSTR(lookinstr
                                          , beginlook
                                          , searchlen));
    SET beginlook = beginlook + searchlen;
    SET returnvar = returnvar + 1;
  END WHILE;
  IF locatevar = 0 THEN
    SET returnvar = 0;
  END IF;
  RETURN returnvar;
END
```

Below is an example of the function in use. Observe that the function did not find the string "th" in the name "Smith" because the two characters did not start in an position that was some multiple of the length of the test string:

*LOCATE_BLOCK function example*

```
SELECT id
     , name
     , LOCATE('th', name)       AS l1
     , LOCATE_BLOCK('th', name) AS l2
FROM staff
WHERE LOCATE('th', name) > 1;
```

*ANSWER*

| ID | NAME | L1 | L2 |
|---|---|---|---|
| 70 | Rothman | 3 | 2 |
| 220 | Smith | 4 | 0 |

The LOCATE_BLOCK function shown above is the minimalist version, without any error checking. If it were used in a Production environment, it would have checks for nulls, and for various invalid input values.

#Use Function

Now all we need to do is build a string, as we do the recursion, that holds every key value that has previously been accessed. This can be done using simple concatenation:

*Show path, and rows in loop*

```
WITH parent (pkey, ckey, lvl, path, loop) AS
(SELECT DISTINCT pkey
      , pkey
      , 0
      , VARCHAR(pkey, 20)
      , 0
 FROM trouble
 WHERE pkey = 'AAA'
   UNION ALL
 SELECT C.pkey
      , C.ckey
      , P.lvl + 1
      , P.path || C.ckey
      , LOCATE_BLOCK(C.ckey,P.path)
 FROM trouble C
    , parent P
 WHERE P.ckey = C.pkey
 AND P.lvl + 1 < 4)
SELECT *
FROM parent;
```

*ANSWER*

| PKEY | CKEY | LVL | PATH | LOOP |
|---|---|---|---|---|
| AAA | AAA | 0 | AAA | 0 |
| AAA | BBB | 1 | AAABBB | 0 |
| AAA | CCC | 1 | AAACCC | 0 |
| AAA | DDD | 1 | AAADDD | 0 |
| CCC | EEE | 2 | AAACCCEEE | 0 |
| DDD | AAA | 2 | AAADDDAAA | 1 |
| DDD | EEE | 2 | AAADDDEEE | 0 |
| DDD | FFF | 2 | AAADDDFFF | 0 |
| AAA | BBB | 3 | AAADDDAAABBB | 0 |
| AAA | CCC | 3 | AAADDDAAACCC | 0 |
| AAA | DDD | 3 | AAADDDAAADDD | 2 |
| FFF | GGG | 3 | AAADDDFFFGGG | 0 |

Now we can get rid of the level check, and instead use the LOCATE_BLOCK function to avoid loops

in the data:

*Use LOCATE_BLOCK function to stop recursion*

```
WITH parent (pkey, ckey, lvl, path) AS
(SELECT DISTINCT pkey
      , pkey
      , 0
      , VARCHAR(pkey,20)
 FROM trouble
 WHERE pkey = 'AAA'
   UNION ALL
 SELECT C.pkey
      , C.ckey
      , P.lvl + 1
      , P.path || C.ckey
 FROM trouble C
    , parent P
 WHERE P.ckey = C.pkey
 AND LOCATE_BLOCK(C.ckey,P.path) = 0)
SELECT *
FROM parent;
```

*ANSWER*

| PKEY | CKEY | LVL | PATH |
|---|---|---|---|
| AAA | AAA | 0 | AAA |
| AAA | BBB | 1 | AAABBB |
| AAA | CCC | 1 | AAACCC |
| AAA | DDD | 1 | AAADDD |
| CCC | EEE | 2 | AAACCCEEE |
| DDD | EEE | 2 | AAADDDEEE |
| DDD | FFF | 2 | AAADDDFFF |
| FFF | GGG | 3 | AAADDDFFFGGG |

The next query is the same as the previous, except that instead of excluding all loops from the answer-set, it marks them as such, and gets the first item, but goes no further.

*Use LOCATE_BLOCK function to stop recursion*

```
WITH parent (pkey, ckey, lvl, path, loop) AS
(SELECT DISTINCT pkey
      , pkey
      , 0
      , VARCHAR(pkey,20)
      , 0
  FROM trouble
  WHERE pkey = 'AAA'
    UNION ALL
  SELECT C.pkey
       , C.ckey
       , P.lvl + 1
       , P.path || C.ckey
       , LOCATE_BLOCK(C.ckey,P.path) DDD AAA
  FROM trouble C
     , parent P
  WHERE P.ckey = C.pkey
  AND P.loop = 0)
SELECT *
FROM parent;
```

*ANSWER*

| PKEY | CKEY | LVL | PATH | LOOP |
|---|---|---|---|---|
| AAA | AAA | 0 | AAA | 0 |
| AAA | BBB | 1 | AAABBB | 0 |
| AAA | CCC | 1 | AAACCC | 0 |
| AAA | DDD | 1 | AAADDD | 0 |
| CCC | EEE | 2 | AAACCCEEE | 0 |
| DDD | AAA | 2 | AAADDDAAA | 1 |
| DDD | EEE | 2 | AAADDDEEE | 0 |
| DDD | FFF | 2 | AAADDDFFF | 0 |
| FFF | GGG | 3 | AAADDDFFFGGG | 0 |

The next query tosses in another predicate (in the final select) to only list those rows that point back to a previously processed parent:

*List rows that point back to a parent*

```
WITH parent (pkey, ckey, lvl, path, loop) AS
(SELECT DISTINCT pkey
      , pkey
      , 0
      , VARCHAR(pkey,20)
      , 0
 FROM trouble
 WHERE pkey = 'AAA'
   UNION ALL
 SELECT C.pkey
      , C.ckey
      , P.lvl + 1
      , P.path || C.ckey
      , LOCATE_BLOCK(C.ckey,P.path)
 FROM trouble C
    , parent P
 WHERE P.ckey = C.pkey
 AND P.loop = 0)
SELECT pkey
    , ckey
FROM parent
WHERE loop > 0;
```

*ANSWER*

| PKEY | CKEY |
|---|---|
| DDD | AAA |

To delete the offending rows from the table, all one has to do is insert the above values into a temporary table, then delete those rows in the TROUBLE table that match. However, before one does this, one has decide which rows are the ones that should not be there. In the above query, we started processing at AAA, and then said that any row that points back to AAA, or to some child or AAA, is causing a loop. We thus identified the row from DDD to AAA as being a problem. But if we had started at the value DDD, we would have said instead that the row from AAA to DDD was the problem. The point to remember her is that the row you decide to delete is a consequence of the row that you decided to define as your starting point.

*Delete rows that loop back to a parent*

```
DECLARE GLOBAL TEMPORARY TABLE SESSION.del_list
( pkey CHAR(03) NOT NULL
, ckey CHAR(03) NOT NULL)
ON COMMIT PRESERVE ROWS;

INSERT INTO SESSION.del_list
WITH parent (pkey, ckey, lvl, path, loop) AS
(SELECT DISTINCT pkey
      , pkey
      , 0
      , VARCHAR(pkey,20)
      , 0
 FROM trouble
 WHERE pkey = 'AAA'
   UNION ALL
 SELECT C.pkey
      , C.ckey
      , P.lvl + 1
      , P.path || C.ckey
      , LOCATE_BLOCK(C.ckey,P.path)
 FROM trouble C
    , parent P
 WHERE P.ckey = C.pkey
 AND P.loop = 0)
SELECT pkey
     , ckey
FROM parent
WHERE loop > 0;

DELETE FROM trouble
WHERE (pkey,ckey) IN
  (SELECT pkey
        , ckey
   FROM SESSION.del_list);
```

#### Working with Other Key Types

The LOCATE_BLOCK solution shown above works fine, as long as the key in question is a fixed length character field. If it isn't, it can be converted to one, depending on what it is:

- Cast VARCHAR columns as type CHAR.
- Convert other field types to character using the HEX function.

#### Keeping the Hierarchy Clean

Rather that go searching for loops, one can toss in a couple of triggers that will prevent the table from every getting data loops in the first place. There will be one trigger for inserts, and another for updates. Both will have the same general logic:

- For each row inserted/updated, retain the new PKEY value.
- Recursively scan the existing rows, starting with the new CKEY value.
- Compare each existing CKEY value retrieved to the new PKEY value. If it matches, the changed row will cause a loop, so flag an error.
- If no match is found, allow the change.

Here is the insert trigger:

*INSERT trigger*

```
CREATE TRIGGER TBL_INS
NO CASCADE BEFORE INSERT ON trouble
REFERENCING NEW AS NNN
FOR EACH ROW MODE Db2SQL
WITH temp (pkey, ckey) AS
  (VALUES (NNN.pkey
         , NNN.ckey)
    UNION ALL
   SELECT TTT.pkey
        , CASE
            WHEN TTT.ckey = TBL.pkey THEN RAISE_ERROR('70001','LOOP FOUND')
            ELSE TBL.ckey
          END
   FROM trouble TBL
      , temp TTT
   WHERE TTT.ckey = TBL.pkey)
SELECT *
FROM temp;
```

*TROUBLE*

| PKEY | CKEY |
|---|---|
| AAA | BBB |
| AAA | CCC |
| AAA | DDD |
| CCC | EEE |
| DDD | AAA |
| DDD | FFF |
| DDD | EEE |
| FFF | GGG |

Here is the update trigger:

*UPDATE trigger*

```
CREATE TRIGGER TBL_UPD
NO CASCADE BEFORE UPDATE OF pkey, ckey ON trouble
REFERENCING NEW AS NNN
FOR EACH ROW MODE Db2SQL
WITH temp (pkey, ckey) AS
(VALUES (NNN.pkey
       , NNN.ckey)
   UNION ALL
 SELECT TTT.pkey
      , CASE
          WHEN TTT.ckey = TBL.pkey THEN RAISE_ERROR('70001','LOOP FOUND')
          ELSE TBL.ckey
        END
 FROM trouble TBL
    , temp TTT
 WHERE TTT.ckey = TBL.pkey)
SELECT *
FROM temp;
```

Given the above preexisting TROUBLE data (absent the DDD to AAA row), the following statements would be rejected by the above triggers:

*Invalid DML statements*

```
INSERT INTO trouble VALUES('GGG','AAA');
UPDATE trouble SET ckey = 'AAA' WHERE pkey = 'FFF';
UPDATE trouble SET pkey = 'GGG' WHERE ckey = 'DDD';
```

Observe that neither of the above triggers use the LOCATE_BLOCK function to find a loop. This is because these triggers are written assuming that the table is currently loop free. If this is not the case, they may run forever. The LOCATE_BLOCK function enables one to check every row processed, to see if one has been to that row before. In the above triggers, only the start position is checked for loops. So if there was a loop that did not encompass the start position, the LOCATE_BLOCK check would find it, but the code used in the triggers would not.

# 17.3. Clean Hierarchies and Efficient Joins

## 17.3.1. Introduction

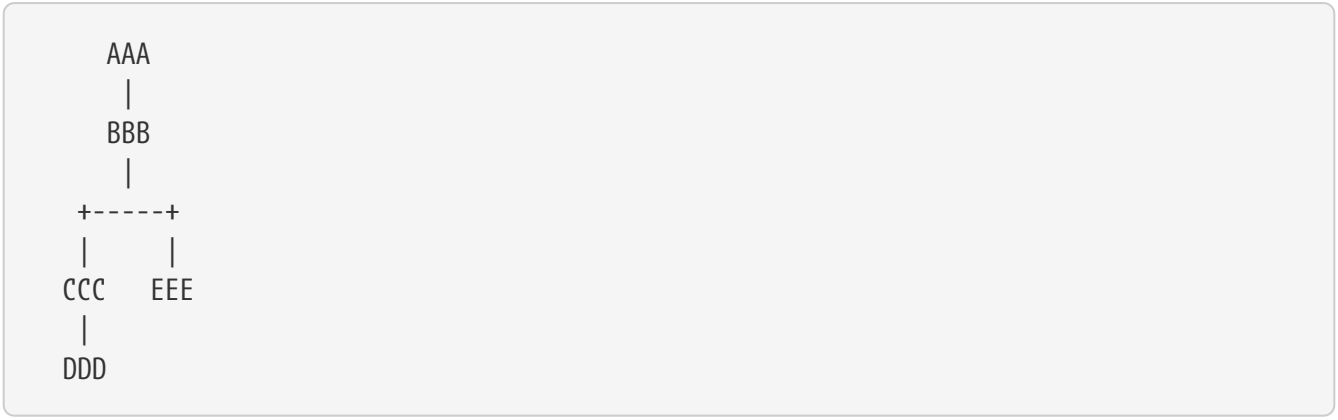
One of the more difficult problems in any relational database system involves joining across multiple hierarchical data structures. The task is doubly difficult when one or more of the hierarchies involved is a data structure that has to be resolved using recursive processing. In this section, we will describe how one can use a mixture of tables and triggers to answer this kind of query very efficiently. A typical question might go as follows: Find all matching rows where the customer is in some geographic region, and the item sold is in some product category, and person who made the sale is in some company sub-structure. If each of these qualifications involves

expanding a hierarchy of object relationships of indeterminate and/or nontrivial depth, then a simple join or standard data denormalization will not work. In Db2, one can answer this kind of question by using recursion to expand each of the data hierarchies. Then the query would join (sans indexes) the various temporary tables created by the recursive code to whatever other data tables needed to be accessed. Unfortunately, the performance will probably be lousy. Alternatively, one can often efficiently answer this general question using a set of suitably indexed summary tables that are an expanded representation of each data hierarchy. With these tables, the Db2 optimizer can much more efficiently join to other data tables, and so deliver suitable performance. In this section, we will show how to make these summary tables and, because it is a prerequisite, also show how to ensure that the related base tables do not have recursive data structures. Two solutions will be described: One that is simple and efficient, but which stops updates to key values. And another that imposes fewer constraints, but which is a bit more complicated.

## 17.3.2. Limited Update Solution

Below on the left is a hierarchy of data items. This is a typical unbalanced, non-recursive data hierarchy. In the center is a normalized representation of this hierarchy. The only thing that is perhaps a little unusual here is that an item at the top of a hierarchy (e.g. AAA) is deemed to be a parent of itself. On the right is an exploded representation of the same hierarchy.

*Data Hierarchy, with normalized and exploded representations*




*HIERARCHY#1*

| KEYY | PKEY | DATA |
|---|---|---|
| AAA | AAA | SOME DATA |
| BBB | AAA | MORE DATA |
| CCC | BBB | MORE JUNK |
| DDD | CCC | MORE JUNK |
| EEE | BBB | JUNK DATA |

*EXPLODED#1*

| PKEY | CKEY | LVL |
|---|---|---|
| AAA | AAA | 0 |

| PKEY | CKEY | LVL |
|---|---|---|
| AAA | BBB | 1 |
| AAA | CCC | 2 |
| AAA | DDD | 3 |
| AAA | EEE | 2 |
| BBB | BBB | 0 |
| BBB | CCC | 1 |
| BBB | DDD | 2 |
| BBB | EEE | 1 |
| CCC | CCC | 0 |
| CCC | DDD | 1 |
| DDD | DDD | 0 |
| EEE | EEE | 0 |

Below is the CREATE code for the above normalized table and a dependent trigger:

*Hierarchy table that does not allow updates to PKEY*

```
CREATE TABLE hierarchy#1
( keyy CHAR(3) NOT NULL
, pkey CHAR(3) NOT NULL
, data VARCHAR(10)
, CONSTRAINT hierarchy11 PRIMARY KEY(keyy)
, CONSTRAINT hierarchy12 FOREIGN KEY(pkey)
REFERENCES hierarchy#1 (keyy) ON DELETE CASCADE);

CREATE TRIGGER HIR#1_UPD
NO CASCADE BEFORE UPDATE OF pkey ON hierarchy#1
REFERENCING NEW AS NNN
            OLD AS OOO
FOR EACH ROW MODE Db2SQL
WHEN (NNN.pkey <> OOO.pkey)
    SIGNAL SQLSTATE '70001' ('CAN NOT UPDATE pkey');
```

Note the following:

- The KEYY column is the primary key, which ensures that each value must be unique, and that this field can not be updated.
- The PKEY column is a foreign key of the KEYY column. This means that this field must always refer to a valid KEYY value. This value can either be in another row (if the new row is being inserted at the bottom of an existing hierarchy), or in the new row itself (if a new independent data hierarchy is being established).
- The ON DELETE CASCADE referential integrity rule ensures that when a row is deleted, all

dependent rows are also deleted.

- The TRIGGER prevents any updates to the PKEY column. This is a BEFORE trigger, which means that it stops the update before it is applied to the database.

All of the above rules and restrictions act to prevent either an insert or an update for ever acting on any row that is not at the bottom of a hierarchy. Consequently, it is not possible for a hierarchy to ever exist that contains a loop of multiple data items.

### 17.3.3. Creating an Exploded Equivalent

Once we have ensured that the above table can never have recursive data structures, we can define a dependent table that holds an exploded version of the same hierarchy. Triggers will be used to keep the two tables in sync. Here is the CREATE code for the table:

*Exploded table CREATE statement*

```
CREATE TABLE exploded#1
( pkey CHAR(4) NOT NULL
, ckey CHAR(4) NOT NULL
, lvl SMALLINT NOT NULL
, PRIMARY KEY(pkey,ckey));
```

The following trigger deletes all dependent rows from the exploded table whenever a row is deleted from the hierarchy table:

*Trigger to maintain exploded table after delete in hierarchy table*

```
CREATE TRIGGER EXP#1_DEL
AFTER DELETE ON hierarchy#1
REFERENCING OLD AS OOO
FOR EACH ROW MODE Db2SQL
DELETE FROM exploded#1
WHERE ckey = OOO.keyy;
```

The next trigger is run every time a row is inserted into the hierarchy table. It uses recursive code to scan the hierarchy table upwards, looking for all parents of the new row. The result set is then inserted into the exploded table:

*Trigger to maintain exploded table after insert in hierarchy table*

```
CREATE TRIGGER EXP#1_INS
AFTER INSERT ON hierarchy#1
REFERENCING NEW AS NNN
FOR EACH ROW MODE Db2SQL
INSERT INTO exploded#1
  WITH temp(pkey, ckey, lvl) AS
  (VALUES (NNN.keyy
         , NNN.keyy
         , 0)
     UNION ALL
   SELECT N.pkey
        , NNN.keyy
        , T.lvl +1
   FROM temp T
      , hierarchy#1 N
   WHERE N.keyy = T.pkey
   AND N.keyy <> N.pkey)
SELECT *
FROM temp;
```

*HIERARCHY#1*

| KEYY | PKEY | DATA |
|---|---|---|
| AAA | AAA | S... |
| BBB | AAA | M... |
| CCC | BBB | M... |
| DDD | CCC | M... |
| EEE | BBB | J... |

*EXPLODED#1*

| PKEY | CKEY | LVL |
|---|---|---|
| AAA | AAA | 0 |
| AAA | BBB | 1 |
| AAA | CCC | 2 |
| AAA | DDD | 3 |
| AAA | EEE | 2 |
| BBB | BBB | 0 |
| BBB | CCC | 1 |
| BBB | DDD | 2 |
| BBB | EEE | 1 |

| PKEY | CKEY | LVL |
|---|---|---|
| CCC | CCC | 0 |
| CCC | DDD | 1 |
| DDD | DDD | 0 |
| EEE | EEE | 0 |

There is no update trigger because updates are not allowed to the hierarchy table.

### 17.3.4. Querying the Exploded Table

Once supplied with suitable indexes, the exploded table can be queried like any other table. It will always return the current state of the data in the related hierarchy table.

*Querying the exploded table*

```
SELECT *
FROM exploded#1
WHERE pkey = :host-var
ORDER BY pkey
       , ckey
       , lvl;
```

### 17.3.5. Full Update Solution

Not all applications want to limit updates to the data hierarchy as was done above. In particular, they may want the user to be able to move an object, and all its dependents, from one valid point (in a data hierarchy) to another. This means that we cannot prevent valid updates to the PKEY value. Below is the CREATE statement for a second hierarchy table. The only difference between this table and the previous one is that there is now an ON UPDATE RESTRICT clause. This prevents updates to PKEY that do not point to a valid KEYY value – either in another row, or in the row being updated:

*Hierarchy table that allows updates to PKEY*

```
CREATE TABLE hierarchy#2
( keyy CHAR(3) NOT NULL
, pkey CHAR(3) NOT NULL
, data VARCHAR(10)
, CONSTRAINT NO_loopS21 PRIMARY KEY(keyy)
, CONSTRAINT NO_loopS22 FOREIGN KEY(pkey)
REFERENCES hierarchy#2 (keyy) ON DELETE CASCADE
                              ON UPDATE RESTRICT);
```

The previous hierarchy table came with a trigger that prevented all updates to the PKEY field. This table comes instead with a trigger than checks to see that such updates do not result in a recursive data structure. It starts out at the changed row, then works upwards through the chain of PKEY values. If it ever comes back to the original row, it flags an error:

*Trigger to check for recursive data structures before update of PKEY*

```
CREATE TRIGGER HIR#2_UPD
NO CASCADE BEFORE UPDATE OF pkey ON hierarchy#2
REFERENCING NEW AS NNN
            OLD AS OOO
FOR EACH ROW MODE Db2SQL
WHEN (NNN.pkey <> OOO.pkey
AND NNN.pkey <> NNN.keyy)
    WITH temp (keyy, pkey) AS
    (VALUES (NNN.keyy
           , NNN.pkey)
       UNION ALL
     SELECT LP2.keyy
          , CASE
              WHEN LP2.keyy = NNN.keyy THEN RAISE_ERROR('70001','LOOP FOUND')
              ELSE LP2.pkey
            END
     FROM hierarchy#2 LP2
        , temp TMP
     WHERE TMP.pkey = LP2.keyy
     AND TMP.keyy <> TMP.pkey)
SELECT *
FROM temp;
```

*HIERARCHY#2*

| KEYY | PKEY | DATA |
|---|---|---|
| AAA | AAA | S... |
| BBB | AAA | M... |
| CCC | BBB | M... |
| DDD | CCC | M... |
| EEE | BBB | J... |

The above is a BEFORE trigger, which means that it gets run before the change is applied to the database. By contrast, the triggers that maintain the exploded table are all AFTER triggers. In general, one uses before triggers check for data validity, while after triggers are used to propagate changes.

### 17.3.6. Creating an Exploded Equivalent

The following exploded table is exactly the same as the previous. It will be maintained in sync with changes to the related hierarchy table:

*Exploded table CREATE statement*

```
CREATE TABLE exploded#2
( pkey CHAR(4) NOT NULL
, ckey CHAR(4) NOT NULL
, lvl SMALLINT NOT NULL
, PRIMARY KEY(pkey,ckey));
```

Three triggers are required to maintain the exploded table in sync with the related hierarchy table. The first two, which handle deletes and inserts, are the same as what were used previously. The last, which handles updates, is new (and quite tricky). The following trigger deletes all dependent rows from the exploded table whenever a row is deleted from the hierarchy table:

*Trigger to maintain exploded table after delete in hierarchy table*

```
CREATE TRIGGER EXP#2_DEL
AFTER DELETE ON hierarchy#2
REFERENCING OLD AS OOO
FOR EACH ROW MODE Db2SQL
DELETE FROM exploded#2
  WHERE ckey = OOO.keyy;
```

The next trigger is run every time a row is inserted into the hierarchy table. It uses recursive code to scan the hierarchy table upwards, looking for all parents of the new row. The resultset is then inserted into the exploded table:

*Trigger to maintain exploded table after insert in hierarchy table*

```
CREATE TRIGGER EXP#2_INS
AFTER INSERT ON hierarchy#2
REFERENCING NEW AS NNN
FOR EACH ROW MODE Db2SQL
INSERT INTO exploded#2
  WITH temp(pkey, ckey, lvl) AS
(SELECT NNN.keyy
      , NNN.keyy
      , 0
 FROM hierarchy#2
 WHERE keyy = NNN.keyy
   UNION ALL
 SELECT N.pkey
      , NNN.keyy
      , T.lvl +1
 FROM temp T
    , hierarchy#2 N
 WHERE N.keyy = T.pkey
 AND N.keyy <> N.pkey)
SELECT *
FROM temp;
```

*HIERARCHY#2*

| KEYY | PKEY | DATA |
|---|---|---|
| AAA | AAA | S… |
| BBB | AAA | M… |
| CCC | BBB | M… |
| DDD | CCC | M… |
| EEE | BBB | J… |

*EXPLODED#2*

| PKEY | CKEY | LVL |
|---|---|---|
| AAA | AAA | 0 |
| AAA | BBB | 1 |
| AAA | CCC | 2 |
| AAA | DDD | 3 |
| AAA | EEE | 2 |
| BBB | BBB | 0 |
| BBB | CCC | 1 |
| BBB | DDD | 2 |
| BBB | EEE | 1 |
| CCC | CCC | 0 |
| CCC | DDD | 1 |
| DDD | DDD | 0 |
| EEE | EEE | 0 |

The next trigger is run every time a PKEY value is updated in the hierarchy table. It deletes and then reinserts all rows pertaining to the updated object, and all it's dependents. The code goes as follows: Delete all rows that point to children of the row being updated. The row being updated is also considered to be a child. In the following insert, first use recursion to get a list of all of the children of the row that has been updated. Then work out the relationships between all of these children and all of their parents. Insert this second result-set back into the exploded table.

*Trigger to run after update of PKEY in hierarchy table*

```
CREATE TRIGGER EXP#2_UPD
AFTER UPDATE OF pkey ON hierarchy#2
REFERENCING OLD AS OOO
            NEW AS NNN
FOR EACH ROW MODE Db2SQL
BEGIN ATOMIC
  DELETE FROM exploded#2
  WHERE ckey IN
      (SELECT ckey
       FROM exploded#2
       WHERE pkey = OOO.keyy);
  INSERT INTO exploded#2
  WITH temp1(ckey) AS
  (VALUES (NNN.keyy)
     UNION ALL
   SELECT N.keyy
   FROM temp1 T
      , hierarchy#2 N
   WHERE N.pkey = T.ckey
   AND N.pkey <> N.keyy)
  , temp2(pkey, ckey, lvl) AS
  (SELECT ckey
        , ckey
        , 0
   FROM temp1
     UNION ALL
   SELECT N.pkey
        , T.ckey
        , T.lvl +1
   FROM temp2 T
      , hierarchy#2 N
   WHERE N.keyy = T.pkey
   AND N.keyy <> N.pkey)
  SELECT *
  FROM temp2;
END
```

The above trigger lacks a statement terminator because it contains atomic SQL, which means that the semi-colon can not be used. Choose anything you like.

### 17.3.7. Querying the Exploded Table

Once supplied with suitable indexes, the exploded table can be queried like any other table. It will always return the current state of the data in the related hierarchy table.

*Querying the exploded table*

```
SELECT *
FROM exploded#2
WHERE pkey = :host-var
ORDER BY pkey
       , ckey
       , lvl;
```

Below are some suggested indexes:

- PKEY, CKEY (already defined as part of the primary key).
- CKEY, PKEY (useful when joining to this table).

# Chapter 18. Triggers

A trigger initiates an action whenever a row, or set of rows, is changed. The change can be either an insert, update or delete.

The Db2 Application Development Guide: Programming Server Applications is an excellent source of information on using triggers. The SQL Reference has all the basics.

## 18.1. Trigger Types

- A **BEFORE** trigger is run before the row is changed. It is typically used to change the values being entered (e.g. set a field to the current date), or to flag an error. It cannot be used to initiate changes in other tables.
- An **AFTER** trigger is run after the row is changed. It can do everything a before trigger can do, plus modify data in other tables or systems (e.g. it can insert a row into an audit table after an update).
- An **INSTEAD OF** trigger is used in a view to do something instead of the action that the user intended (e.g. do an insert instead of an update). There can be only one instead of trigger per possible DML type on a given view.

See the chapter titled Retaining a Record for a sample application that uses INSTEAD OF triggers to record all changes to the data in a set of tables.

## 18.2. Action Type

Each trigger applies to a single kind of DML action (i.e. insert, update, or delete). With the exception of instead of triggers, there can be as many triggers per action and per table as desired. An update trigger can be limited to changes to certain columns.

**Object Type**

- A table can have both BEFORE and AFTER triggers. The former have to be defined FOR EACH ROW.
- A view can have INSTEAD OF triggers (up to three - one per DML type).

**Referencing**

In the body of the trigger the object being changed can be referenced using a set of optional correlation names:

- **OLD** refers to each individual row before the change (does not apply to an insert).
- **NEW** refers to each individual row after the change (does not apply to a delete).
- **OLD_TABLE** refers to the set of rows before the change (does not apply to an insert).
- **NEW_TABLE** refers to the set of rows after the change (does to apply to a delete).

**Application Scope**

- A trigger defined FOR EACH STATEMENT is invoked once per statement.
- A trigger defined FOR EACH ROW is invoked once per individual row changed.

If one defines two FOR EACH ROW triggers, the first is applied for all rows before the second is run. To do two separate actions per row, one at a time, one has to define a single trigger that includes the two actions in a single compound SQL statement.

**When Check**

One can optionally include some predicates so that the body of the trigger is only invoked when certain conditions are true.

**Trigger Usage**

A trigger can be invoked whenever one of the following occurs:

- A row in a table is inserted, updated, or deleted.
- An (implied) row in a view is inserted, updated, or deleted.
- A referential integrity rule on a related table causes a cascading change (i.e. delete or set null) to the triggered table.
- A trigger on an unrelated table or view is invoked - and that trigger changes rows in the triggered table.

If no rows are changed, a trigger defined FOR EACH ROW is not run, while a trigger defined FOR EACH STATEMENT is still run. To prevent the latter from doing anything when this happens, add a suitable WHEN check.

# 18.3. Trigger Examples

This section uses a set of simple sample tables to illustrate general trigger usage.

## 18.3.1. Sample Tables

*Sample Tables*

```
CREATE TABLE cust_balance
( cust# INTEGER NOT NULL
                GENERATED ALWAYS AS IDENTITY
, status CHAR(2) NOT NULL
, balance DECIMAL(18,2) NOT NULL
, num_trans INTEGER NOT NULL
, cur_ts TIMESTAMP NOT NULL
, PRIMARY KEY (cust#));

-- Every state of a row in the balance table will be recorded in the history table.
CREATE TABLE cust_history
( cust# INTEGER NOT NULL
, trans# INTEGER NOT NULL
, balance DECIMAL(18,2) NOT NULL
, bgn_ts TIMESTAMP NOT NULL
, end_ts TIMESTAMP NOT NULL
, PRIMARY KEY (cust#, bgn_ts));

-- Every valid change to the balance table will be recorded in the transaction table.
CREATE TABLE cust_trans
( min_cust# INTEGER
, max_cust# INTEGER
, rows_tot INTEGER NOT NULL
, change_val DECIMAL(18,2)
, change_type CHAR(1) NOT NULL
, cur_ts TIMESTAMP NOT NULL
, PRIMARY KEY (cur_ts));
```

## 18.4. Before Row Triggers - Set Values

The first trigger below overrides whatever the user enters during the insert, and before the row is inserted, sets both the cur-ts and number-of-trans columns to their correct values:

*Before insert trigger - set values*

```
CREATE TRIGGER cust_bal_ins1
NO CASCADE BEFORE INSERT
ON cust_balance
REFERENCING NEW AS nnn
FOR EACH ROW
MODE Db2SQL
SET nnn.cur_ts = CURRENT TIMESTAMP
  , nnn.num_trans = 1;
```

The following trigger does the same before an update:

*Before update trigger - set values*

```
CREATE TRIGGER cust_bal_upd1
NO CASCADE BEFORE UPDATE
ON cust_balance
REFERENCING NEW AS nnn
            OLD AS ooo
FOR EACH ROW
MODE Db2SQL
SET nnn.cur_ts = CURRENT TIMESTAMP
  , nnn.num_trans = ooo.num_trans + 1;
```

## 18.5. Before Row Trigger - Signal Error

The next trigger will flag an error (and thus fail the update) if the customer balance is reduced by too large a value:

*Before Trigger - flag error*

```
CREATE TRIGGER cust_bal_upd2
NO CASCADE BEFORE UPDATE OF balance
ON cust_balance
REFERENCING NEW AS nnn
            OLD AS ooo
FOR EACH ROW
MODE Db2SQL
WHEN (ooo.balance - nnn.balance > 1000)
   SIGNAL SQLSTATE VALUE '71001'
   SET MESSAGE_TEXT = 'Cannot withdraw > 1000';
```

## 18.6. After Row Triggers - Record Data States

The three triggers in this section record the state of the data in the customer table. The first is invoked after each insert. It records the new data in the customer-history table:

*After Trigger - record insert*

```
CREATE TRIGGER cust_his_ins1
AFTER INSERT ON cust_balance
REFERENCING NEW AS nnn
FOR EACH ROW
MODE Db2SQL
INSERT INTO cust_history
VALUES (nnn.cust#
      , nnn.num_trans
      , nnn.balance
      , nnn.cur_ts
      , '9999-12-31-24.00.00');
```

The next trigger is invoked after every update of a row in the customer table. It first runs an update (of the old history row), and then does an insert. Because this trigger uses a compound SQL statement, it cannot use the semi-colon as the statement delimiter:

*After Trigger - record update*

```
CREATE TRIGGER cust_his_upd1
AFTER UPDATE ON cust_balance
REFERENCING OLD AS ooo
            NEW AS nnn
FOR EACH ROW
MODE Db2SQL
BEGIN ATOMIC
  UPDATE cust_history
  SET end_ts = CURRENT TIMESTAMP
  WHERE cust# = ooo.cust#
  AND bgn_ts = ooo.cur_ts;
  INSERT INTO cust_history
  VALUES (nnn.cust#
        , nnn.num_trans
        , nnn.balance
        , nnn.cur_ts
        , '9999-12-31-24.00.00');
END
```

**Notes**

- The above trigger relies on the fact that the customer-number cannot change (note: it is generated always) to link the two rows in the history table together. In other words, the old row will always have the same customer-number as the new row.
- The above also trigger relies on the presence of the cust_bal_upd1 before trigger (see Trigger Examples) to set the nnn.cur_ts value to the current timestamp.

The final trigger records a delete by doing an update to the history table:

*After Trigger - record delete*

```
CREATE TRIGGER cust_his_del1
AFTER DELETE ON cust_balance
REFERENCING OLD AS ooo
FOR EACH ROW
MODE Db2SQL
UPDATE cust_history
SET end_ts = CURRENT TIMESTAMP
WHERE cust# = ooo.cust#
AND bgn_ts = ooo.cur_ts;
```

# 18.7. After Statement Triggers - Record Changes

The following three triggers record every type of change (i.e. insert, update, or delete) to any row, or set of rows (including an empty set) in the customer table. They all run an insert that records the type and number of rows changed:

*After Trigger - record insert*

```
CREATE TRIGGER trans_his_ins1
AFTER INSERT ON cust_balance
REFERENCING NEW_TABLE AS newtab
FOR EACH STATEMENT
MODE Db2SQL
INSERT INTO cust_trans
  SELECT MIN(cust#)
  , MAX(cust#)
  , COUNT(*)
  , SUM(balance)
  , 'I'
  , CURRENT TIMESTAMP
FROM newtab;
```

*After Trigger - record update*

```
CREATE TRIGGER trans_his_upd1
AFTER UPDATE ON cust_balance
REFERENCING OLD_TABLE AS oldtab
NEW_TABLE AS newtab
FOR EACH STATEMENT
MODE Db2SQL
INSERT INTO cust_trans
  SELECT MIN(nt.cust#)
       , MAX(nt.cust#)
       , COUNT(*)
       , SUM(nt.balance - ot.balance)
       , 'U'
       , CURRENT TIMESTAMP
  FROM oldtab ot
     , newtab nt
  WHERE ot.cust# = nt.cust#;
```

*After Trigger - record delete*

```
CREATE TRIGGER trans_his_del1
AFTER DELETE ON cust_balance
REFERENCING OLD_TABLE AS oldtab
FOR EACH STATEMENT
MODE Db2SQL
INSERT INTO cust_trans
  SELECT MIN(cust#)
       , MAX(cust#)
       , COUNT(*)
       , SUM(balance)
       , 'D'
       , CURRENT TIMESTAMP
  FROM oldtab;
```

**Notes**

- If the DML statement changes no rows, the OLD or NEW table referenced by the trigger will be empty, but still exist, and a SELECT COUNT(*) on the (empty) table will return a zero, which will then be inserted.
- Any DML statements that failed (e.g. stopped by the before trigger), or that were subsequently rolled back, will not be recorded in the transaction table.

### 18.7.1. Examples of Usage

The following DML statements were run against the customer table:

*Sample DML statements*

```
INSERT INTO cust_balance (status, balance) VALUES ('C',123.45);
INSERT INTO cust_balance (status, balance) VALUES ('C',000.00);
INSERT INTO cust_balance (status, balance) VALUES ('D', -1.00);
UPDATE cust_balance
  SET balance = balance + 123
WHERE cust# <= 2;
UPDATE cust_balance
  SET balance = balance * -1
WHERE cust# = -1;
UPDATE cust_balance
  SET balance = balance - 123
WHERE cust# = 1;
DELETE FROM cust_balance
WHERE cust# = 3;
```

## 18.8. Tables After DML

At the end of the above, the three tables had the following data:

*Table 14. Customer-balance table rows*

| CUST# | STATUS | BALANCE | NUM_TRANS | CUR_TS |
|---|---|---|---|---|
| 1 | C | 123.45 | 3 | 2005-05-31-19.58.46.096000 |
| 2 | C | 123.00 | 2 | 2005-05-31-19.58.46.034000 |

*Table 15. Customer-history table rows*

| CUST# | TRANS# | BALANCE | BGN_TS | END_TS |
|---|---|---|---|---|
| 1 | 1 | 123.45 | 2005-05-31-19.58.45.971000 | 2005-05-31-19.58.46.034000 |
| 1 | 2 | 246.45 | 2005-05-31-19.58.46.034000 | 2005-05-31-19.58.46.096000 |
| 1 | 3 | 123.45 | 2005-05-31-19.58.46.096000 | 9999-12-31-24.00.00.000000 |
| 2 | 1 | 0.00 | 2005-05-31-19.58.45.987000 | 2005-05-31-19.58.46.034000 |
| 2 | 2 | 123.00 | 2005-05-31-19.58.46.034000 | 9999-12-31-24.00.00.000000 |
| 3 | 1 | -1.00 | 2005-05-31-19.58.46.003000 | 2005-05-31-19.58.46.096003 |

*Table 16. Customer-transaction table rows*

| MIN_CUST# | MAX_CUST# | ROWS | CHANGE_VAL | CHANGE_TYPE | CUR_TS |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 123.45 | I | 2005-05-31-19.58.45.971000 |
| 2 | 2 | 1 | 0.00 | I | 2005-05-31-19.58.45.987000 |
| 3 | 3 | 1 | -1.00 | I | 2005-05-31-19.58.46.003000 |
| 1 | 2 | 2 | 246.00 | U | 2005-05-31-19.58.46.034000 |
| - | - | 0 | - | U | 2005-05-31-19.58.46.065000 |

| MIN_CUST# | MAX_CUST# | ROWS | CHANGE_VAL | CHANGE_TYPE | CUR_TS |
|---|---|---|---|---|---|
| 1 | 1 | 1 | -123.00 | U | 2005-05-31-19.58.46.096000 |
| 3 | 3 | 1 | 1.00 | D | 2005-05-31-19.58.46.096003 |

# Chapter 19. Protecting Your Data

There is no use having a database if the data in it is not valid. This chapter introduces some of the tools that exist in Db2 to enable one to ensure the validity of the data in your application.

**Issues Covered**

- Enforcing field uniqueness.
- Enforcing field value ranges.
- Generating key and values.
- Maintaining summary columns.
- Enforcing relationships between and within tables.
- Creating columns that have current timestamp of last change.

**Issues Not Covered**

- Data access authorization.
- Recovery and backup.

## 19.1. Sample Application

Consider the following two tables, which make up a very simple application:

*Sample application tables*

```
CREATE TABLE customer_balance
( cust_id        INTEGER
, cust_name      VARCHAR(20)
, cust_sex       CHAR(1)
, num_sales      SMALLINT
, total_sales    DECIMAL(12, 2)
, master_cust_id INTEGER
, cust_insert_ts TIMESTAMP
, cust_update_ts TIMESTAMP);

CREATE TABLE us_sales
( invoice#       INTEGER
, cust_id        INTEGER
, sale_value     DECIMAL(18, 2)
, sale_insert_ts TIMESTAMP
, sale_update_ts TIMESTAMP);
```

## 19.2. Customer Balance Table

We want Db2 to enforce the following business rules:

- CUST_ID will be a unique positive integer value, always ascending, never reused, and automatically generated by Db2. This field cannot be updated by a user.
- CUST_NAME has the customer name. It can be anything, but not blank.
- CUST_SEX must be either "M" or "F".
- NUM_SALES will have a count of the sales (for the customer), as recorded in the related US-sales table. The value will be automatically maintained by Db2. It cannot be updated directly by a user.
- TOTAL_SALES will have the sum sales (in US dollars) for the customer. The value will be automatically updated by Db2. It cannot be updated directly by a user.
- MASTER_CUST_ID will have, if there exists, the customer-ID of the customer that this customer is a dependent of. If there is no master customer, the value is null. If the master customer is deleted, this row will also be deleted (if possible).
- CUST_INSERT_TS has the timestamp when the row was inserted. The value is automatically generated by Db2. Any attempt to change will induce an error.
- CUST_UPDATE_TS has the timestamp when the row, or a dependent US_SALES row, was last updated by a user. The value is automatically generated by Db2. Any attempt to change directly will induce an error.
- A row can only be deleted when there are no corresponding rows in the US-sales table (i.e. for the same customer).

**US Sales Table**

We want Db2 to enforce the following business rules:

- INVOICE#: will be a unique ascending integer value. The uniqueness will apply to the US-sales table, plus any international sales tables (i.e. to more than one table).
- CUST_ID is the customer ID, as recorded in the customer-balance table. No row can be inserted into the US-sales table except that there is a corresponding row in the customerbalance table. Once inserted, this value cannot be updated.
- SALE_VALUE is the value of the sale, in US dollars. When a row is inserted, this value is added to the related total-sales value in the customer-balance table. If the value is subsequently updated, the total-sales value is maintained in sync.
- SALE_INSERT_TS has the timestamp when the row was inserted. The value is automatically generated by Db2. Any attempt to change will induce an error.
- SALE_UPDATE_TS has the timestamp when the row was last updated. The value is automatically generated by Db2. Any attempt to change will induce an error.
- Deleting a row from the US-sales table has no impact on the customer-balance table (i.e. the total-sales is not decremented). But a row can only be deleted from the latter when there are no more related rows in the US-sales table.

## 19.3. Enforcement Tools

To enforce the above business rules, we are going to have to use:

- Unique indexes.
- Secondary non-unique indexes (needed for performance).
- Primary and foreign key definitions.
- User-defined distinct data types.
- Nulls-allowed and not-null columns.
- Column value constraint rules.
- Before and after triggers.
- Generated row change timestamps.
- Distinct Data Types

Two of the fields are to contain US dollars, the implication being the data in these columns should not be combined with columns that contain Euros, or Japanese Yen, or my shoe size. To this end, we will define a distinct data type for US dollars:

*Create US-dollars data type*

```
CREATE DISTINCT TYPE us_dollars
AS decimal(18, 2) WITH COMPARISONS;
```

See Distinct Types for a more detailed discussion of this topic.

## 19.4. Customer-Balance Table

Now that we have defined the data type, we can create our first table:

*Customer-Balance table DDL*

```
CREATE TABLE customer_balance
( cust_id        INTEGER NOT NULL
                       GENERATED ALWAYS AS IDENTITY
                      ( START WITH 1
                      , INCREMENT BY 1
                      , NO CYCLE
                      , NO CACHE)
, cust_name      VARCHAR(20) NOT NULL
, cust_sex       CHAR(1) NOT NULL
, num_sales      SMALLINT NOT NULL
, total_sales    us_dollars NOT NULL
, master_cust_id INTEGER
, cust_insert_ts TIMESTAMP NOT NULL
, cust_update_ts TIMESTAMP NOT NULL
, PRIMARY KEY (cust_id)
, CONSTRAINT c1 CHECK (cust_name <> '')
, CONSTRAINT c2 CHECK (cust_sex = 'F' OR cust_sex = 'M')
, CONSTRAINT c3 FOREIGN KEY (master_cust_id)
    REFERENCES customer_balance (cust_id)
    ON DELETE CASCADE);
```

The following business rules are enforced above:

- The customer-ID is defined as an identity column (see Identity Columns and Sequences), which means that the value is automatically generated by Db2 using the rules given. The field cannot be updated by the user.
- The customer-ID is defined as the primary key, which automatically generates a unique index on the field, and also enables us to reference the field using a referential integrity rule. Being a primary key prevents updates, but we had already prevented them because the field is an identity column.
- The total-sales column uses the type us-dollars.
- Constraints C1 and C2 enforce two data validation rules.
- Constraint C3 relates the current row to a master customer, if one exists. Furthermore, if the master customer is deleted, this row is also deleted.
- All of the columns, except for the master-customer-id, are defined as NOT NULL, which means that a value must be provided. We still have several more business rules to enforce - relating to automatically updating fields and/or preventing user updates. These will be enforced using triggers.

## 19.5. US-Sales Table

Now for the related US-sales table:

*US-Sales table DDL*

```
CREATE TABLE us_sales
( invoice# INTEGER NOT NULL
, cust_id INTEGER NOT NULL
, sale_value us_dollars NOT NULL
, sale_insert_ts TIMESTAMP NOT NULL
, sale_update_ts TIMESTAMP NOT NULL
                           GENERATED ALWAYS
                           FOR EACH ROW ON UPDATE
                           AS ROW CHANGE TIMESTAMP

, PRIMARY KEY (invoice#)
, CONSTRAINT u1 CHECK (sale_value > us_dollars(0))
, CONSTRAINT u2 FOREIGN KEY (cust_id)
      REFERENCES customer_balance
      ON DELETE RESTRICT);

COMMIT;

CREATE INDEX us_sales_cust ON us_sales (cust_id);
```

The following business rules are enforced above:

- The invoice# is defined as the primary key, which automatically generates a unique index on the field, and also prevents updates.
- The sale-value uses the type us-dollars.
- Constraint U1 checks that the sale-value is always greater than zero.
- Constraint U2 checks that the customer-ID exists in the customer-balance table, and also prevents rows from being deleted from the latter if there is a related row in this table.
- All of the columns are defined as NOT NULL, so a value must be provided for each.
- A secondary non-unique index is defined on customer-ID, so that deletes to the customerbalance table (which require checking this table for related customer-ID rows) are as efficient as possible.
- The CUST_UPDATE_TS column is generated always (by Db2) and gets a unique value that is the current timestamp.

### 19.5.1. Generated Always Timestamp Columns

A TIMESTAMP column that is defined as GENERATED ALWAYS will get a value that is unique for all rows in the table. This value will usually be the CURRENT TIMESTAMP of the last insert or update of the row. However, if more than row was inserted or updated in a single stmt, the secondary rows (updated) will get a value that is equal to the CURRENT TIMESTAMP special register, plus "n" microseconds, where "n" goes up in steps of 1. One consequence of the above logic is that some rows changed will get a timestamp value that is ahead of the CURRENT TIMESTAMP special register. This can cause problems if one is relying on this value to find all rows that were changed before the start of the query. To illustrate, imagine that one inserted multiple rows (in a single insert) into the

US_SALES table, and then immediately ran the following query:

*Select run after multi-row insert*

```
SELECT *
FROM us_sales
WHERE sale_update_ts <= CURRENT TIMESTAMP;
```

In some environments (e.g. Windows) the CURRENT TIMESTAMP special register value may be the same from one stmt to the next. If this happens, the above query will find the first row just inserted, but not any subsequent rows, because their SALE_UPDATE_TS value will be greater than the CURRENT TIMESTAMP special register.

Certain restrictions apply:

- Only one TIMESTAMP column can be defined GENERATED ALWAYS per table. The column must be defined NOT NULL.
- The TIMESTAMP column is updated, even if no other value in the row changes. So if one does an update that sets SALE_VALUE = SALE_VALUE + 0, the SALE_UPDATE_TS column will be updated on all matching rows.

The ROW CHANGE TIMESTAMP special register can be used get the last time that the row was updated, even when one does not know the name of the column that holds this data:

*Row change timestamp usage*

```
SELECT ROW CHANGE TIMESTAMP FOR us_sales
FROM us_sales
WHERE invoice# = 5;
```

The (unique) TIMESTAMP value obtained above can be used to validate that the target row has not been updated when a subsequent UPDATE is done:

*Update that checks for intervening updates*

```
UPDATE us_sales
SET sale_value = DECIMAL(sale_value) + 1
WHERE invoice# = 5
AND ROW CHANGE TIMESTAMP for us_sales = '2007-11-10-01.02.03';
```

### 19.5.2. Triggers

Triggers can sometimes be quite complex little programs. If coded incorrectly, they can do an amazing amount of damage. As such, it pays to learn quite a lot before using them. Below are some very brief notes, but please refer to the official Db2 documentation for a more detailed description. See also Triggers for a brief chapter on triggers. Individual triggers are defined on a table, and for a particular type of DML statement:

- Insert
- Update
- Delete

A trigger can be invoked once per:

- Row changed.
- Statement run.

A trigger can be invoked:

- Before the change is made.
- After the change is made.

Before triggers change input values before they are entered into the table and/or flag an error. After triggers do things after the row is changed. They may make more changes (to the target table, or to other tables), induce an error, or invoke an external program. SQL statements that select the changes made by DML (see Insert Examples) cannot see the changes made by an after trigger if those changes impact the rows just changed. The action of one "after" trigger can invoke other triggers, which may then invoke other triggers, and so on. Before triggers cannot do this because they can only act upon the input values of the DML statement that invoked them. When there are multiple triggers for a single table/action, each trigger is run for all rows before the next trigger is invoked - even if defined "for each row". Triggers are invoked in the order that they were created.

### 19.5.3. Customer-Balance - Insert Trigger

For each row inserted into the Customer-Balance table we need to do the following:

- Set the num-sales to zero.
- Set the total-sales to zero.
- Set the update-timestamp to the current timestamp.
- Set the insert-timestamp to the current timestamp.

All of this can be done using a simple before trigger:

*Set values during insert*

```
CREATE TRIGGER cust_balance_ins1
NO CASCADE BEFORE INSERT
ON customer_balance
REFERENCING NEW AS nnn
FOR EACH ROW
MODE Db2SQL
SET nnn.num_sales = 0
  , nnn.total_sales = 0
  , nnn.cust_insert_ts = CURRENT TIMESTAMP
  , nnn.cust_update_ts = CURRENT TIMESTAMP;
```

### 19.5.4. Customer-Balance - Update Triggers

For each row updated in the Customer-Balance table we need to do:

- Set the update-timestamp to the current timestamp.
- Prevent updates to the insert-timestamp, or sales fields.
- We can use the following trigger to maintain the update-timestamp:

*Set update-timestamp during update*

```
CREATE TRIGGER cust_balance_upd1
NO CASCADE BEFORE UPDATE OF cust_update_ts
ON customer_balance
REFERENCING NEW AS nnn
FOR EACH ROW
MODE Db2SQL
SET nnn.cust_update_ts = CURRENT TIMESTAMP;
```

We can prevent updates to the insert-timestamp with the following trigger:

*Prevent update of insert-timestamp*

```
CREATE TRIGGER cust_balance_upd2
NO CASCADE BEFORE UPDATE OF cust_insert_ts
ON customer_balance
FOR EACH ROW
MODE Db2SQL
SIGNAL SQLSTATE VALUE '71001'
SET MESSAGE_TEXT = 'Cannot update CUST insert-ts';
```

We don't want users to update the two sales counters directly. But the two fields do have to be updated (by a trigger) whenever there is a change to the us-sales table. The solution is to have a trigger that prevents updates if there is no corresponding row in the us-sales table where the update-timestamp is greater than or equal to the current timestamp:

*Prevent update of sales fields*

```
CREATE TRIGGER cust_balance_upd3
NO CASCADE BEFORE UPDATE OF num_sales
                          , total_sales
ON customer_balance
REFERENCING NEW AS nnn
FOR EACH ROW
MODE Db2SQL
WHEN (CURRENT TIMESTAMP > (SELECT MAX(sss.sale_update_ts)
                           FROM us_sales sss
                           WHERE nnn.cust_id = sss.cust_id))
  SIGNAL SQLSTATE VALUE '71001'
  SET MESSAGE_TEXT = 'Feilds only updated via US-Sales';
```

### 19.5.5. US-Sales - Insert Triggers

For each row inserted into the US-sales table we need to do the following:

- Determine the invoice-number, which is unique over multiple tables.
- Set the update-timestamp to the current timestamp.
- Set the insert-timestamp to the current timestamp.
- Add the sale-value to the existing total-sales in the customer-balance table.
- Increment the num-sales counter in the customer-balance table.

The invoice-number is supposed to be unique over several tables, so we cannot generate it using an identity column. Instead, we have to call the following external sequence:

*Define sequence*

```
CREATE SEQUENCE us_sales_seq AS INTEGER
    START WITH 1
    INCREMENT BY 1
    NO CYCLE
    NO CACHE
    ORDER;
```

Once we have the above, the following trigger will take of the first three items:

*Insert trigger*

```
CREATE TRIGGER us_sales_ins1
NO CASCADE BEFORE INSERT
ON us_sales
REFERENCING NEW AS nnn
FOR EACH ROW
MODE Db2SQL
  SET nnn.invoice# = NEXTVAL FOR us_sales_seq
    , nnn.sale_insert_ts = CURRENT TIMESTAMP;
```

We need to use an "after" trigger to maintain the two related values in the Customer-Balance table. This will invoke an update to change the target row:

*Propagate change to Customer-Balance table*

```
CREATE TRIGGER sales_to_cust_ins1
AFTER INSERT ON us_sales
REFERENCING NEW AS nnn
FOR EACH ROW
MODE Db2SQL
UPDATE customer_balance ccc
  SET ccc.num_sales = ccc.num_sales + 1
    , ccc.total_sales = DECIMAL(ccc.total_sales) + DECIMAL(nnn.sale_value)
  WHERE ccc.cust_id = nnn.cust_id;
```

### 19.5.6. US-Sales - Update Triggers

For each row updated in the US-sales table we need to do the following:

- Prevent the customer-ID or insert-timestamp from being updated.
- Propagate the change to the sale-value to the total-sales in the customer-balance table.

The next trigger prevents updates to the Customer-ID and insert-timestamp:

*Prevent updates to selected columns*

```
CREATE TRIGGER us_sales_upd2
NO CASCADE BEFORE UPDATE OF cust_id
                          , sale_insert_ts
ON us_sales
FOR EACH ROW
MODE Db2SQL
  SIGNAL SQLSTATE VALUE '71001'
  SET MESSAGE_TEXT = 'Can only update sale_value';
```

We need to use an "after" trigger to maintain sales values in the Customer-Balance table:

*Propagate change to Customer-Balance table*

```
CREATE TRIGGER sales_to_cust_upd1
AFTER UPDATE OF sale_value
ON us_sales
REFERENCING NEW AS nnn
            OLD AS ooo
FOR EACH ROW
MODE Db2SQL
UPDATE customer_balance ccc
    SET ccc.total_sales = DECIMAL(ccc.total_sales)
                        - DECIMAL(ooo.sale_value)
                        + DECIMAL(nnn.sale_value)
WHERE ccc.cust_id = nnn.cust_id;
```

## 19.6. Conclusion

The above application will now have logically consistent data. There is, of course, nothing to prevent an authorized user from deleting all rows, but whatever rows are in the two tables will obey the business rules that we specified at the start.

Tools Used

- Primary key - to enforce uniqueness, prevent updates, enable referential integrity.
- Unique index - to enforce uniqueness.
- Non-unique index - for performance during referential integrity check.
- Sequence object - to automatically generate key values for multiple tables.
- Identity column - to automatically generate key values for 1 table.
- Not-null columns - to prevent use of null values.
- Column constraints - to enforce basic domain-range rules.
- Distinct types - to prevent one type of data from being combined with another type.
- Referential integrity - to enforce relationships between rows/tables, and to enable cascading deletes when needed.
- Before triggers - to prevent unwanted changes and set certain values.
- After triggers - to propagate valid changes.
- Automatically generated timestamp value that is always the current timestamp or (in the case of a multi-row update), the current timestamp plus a few microseconds.

# Chapter 20. Retaining a Record

This chapter was written back in the versions before V9.7. It is helpful showing how to use triggers for a specific use case but if you want to implement such a scenario in Db2 you should use the "Time Travel" feature. A chapter about the "Time Travel" will be written in one of the new versions of the book.

This chapter will describe a rather complex table/view/trigger schema that will enable us to offer several features that are often asked for:

- Record every change to the data in an application (auditing).
- Show the state of the data, as it was, at any point in the past (historical analysis).
- Follow the sequence of changes to any item (e.g. customer) in the database.
- Do "what if" analysis by creating virtual copies of the real world, and then changing them as desired, without affecting the real-world data.

Some sample code to illustrate the above concepts will be described below.

## 20.1. Schema Design

**Recording Changes**

Below is a very simple table that records relevant customer data:

*Customer table*

```
CREATE TABLE customer
( cust#     INTEGER NOT NULL
, cust_name CHAR(10)
, cust_mgr  CHAR(10)
, PRIMARY KEY(cust#));
```

One can insert, update, and delete rows in the above table. The latter two actions destroy data, and so are incompatible with using this table to see all (prior) states of the data. One way to record all states of the above table is to create a related customer-history table, and then to use triggers to copy all changes in the main table to the history table. Below is one example of such a history table:

*Customer-history table*

```
CREATE TABLE customer_his
( cust#     INTEGER     NOT NULL
, cust_name CHAR(10)
, cust_mgr  CHAR(10)
, cur_ts    TIMESTAMP   NOT NULL
, cur_actn  CHAR(1)     NOT NULL
, cur_user  VARCHAR(10) NOT NULL
, prv_cust# INTEGER
, prv_ts    TIMESTAMP
, PRIMARY KEY(cust#, cur_ts));

CREATE UNIQUE INDEX customer_his_x1
ON customer_his(cust#, prv_ts, cur_ts);
```

The secondary index shown above will make the following view processing, which looks for a row that replaces the current, much more efficient.

## 20.2. Table Design

The history table has the same fields as the original Customer table, plus the following:

- CUR-TS: The current timestamp of the change.
- CUR-ACTN: The type of change (i.e. insert, update, or delete).
- CUR-USER: The user who made the change (for auditing purposes).
- PRV-CUST#: The previous customer number. This field enables one follow the sequence of changes for a given customer. The value is null if the action is an insert.
- PRV-TS: The timestamp of the last time the row was changed (null for inserts).

Observe that this history table does not have an end-timestamp. Rather, each row points back to the one that it (optionally) replaces. One advantage of such a schema is that there can be a many-to-one relationship between any given row, and the row, or rows, that replace it. When we add versions into the mix, this will become important.

## 20.3. Triggers

Below is the relevant insert trigger. It replicates the new customer row in the history table, along with the new fields. Observe that the two "previous" fields are set to null:

*Insert trigger*

```
CREATE TRIGGER customer_ins
AFTER INSERT ON customer
REFERENCING NEW AS nnn
FOR EACH ROW
MODE Db2SQL
INSERT INTO customer_his
VALUES (nnn.cust#
      , nnn.cust_name
      , nnn.cust_mgr
      , CURRENT TIMESTAMP
      , 'I'
      , USER
      , NULL
      , NULL);
```

Below is the update trigger. Because the customer table does not have a record of when it was last changed, we have to get this value from the history table - using a sub-query to find the most recent row:

*Update trigger*

```
CREATE TRIGGER customer_upd
AFTER UPDATE ON customer
REFERENCING NEW AS nnn
            OLD AS ooo
FOR EACH ROW
MODE Db2SQL
INSERT INTO customer_his
VALUES (nnn.cust#
      , nnn.cust_name
      , nnn.cust_mgr
      , CURRENT TIMESTAMP
      , 'U'
      , USER
      , ooo.cust#
      , (SELECT MAX(cur_ts)
         FROM customer_his hhh
         WHERE ooo.cust# = hhh.cust#));
```

Below is the delete trigger. It is similar to the update trigger, except that the action is different and we are under no obligation to copy over the old non-key-data columns - but we can if we wish:

*Delete trigger*

```
CREATE TRIGGER customer_del
AFTER DELETE ON customer
REFERENCING OLD AS ooo
FOR EACH ROW
MODE Db2SQL
INSERT INTO customer_his
VALUES (ooo.cust#
      , NULL
      , NULL
      , CURRENT TIMESTAMP
      , 'D'
      , USER
      , ooo.cust#
      , (SELECT MAX(cur_ts)
         FROM customer_his hhh
         WHERE ooo.cust# = hhh.cust#));
```

## 20.4. Views

We are now going to define a view that will let the user query the customer-history table – as if it were the ordinary customer table, but to look at the data as it was at any point in the past. To enable us to hide all the nasty SQL that is required to do this, we are going to ask that the user first enter a row into a profile table that has two columns:

- The user's Db2 USER value.
- The point in time at which the user wants to see the customer data.

Here is the profile table definition:

*Profile table*

```
CREATE TABLE profile
( user_id VARCHAR(10) NOT NULL
, bgn_ts TIMESTAMP NOT NULL DEFAULT '9999-12-31-24.00.00'
, PRIMARY KEY(user_id));
```

Below is a view that displays the customer data, as it was at the point in time represented by the timestamp in the profile table. The view shows all customer-history rows, as long as:

- The action was not a delete.
- The current-timestamp is ⇐ the profile timestamp.
- There does not exist any row that "replaces" the current row (and that row has a current timestamp that is ⇐ to the profile timestamp).

Now for the code:

*View of Customer history*

```
CREATE VIEW customer_vw AS
SELECT hhh.*
     , ppp.bgn_ts
FROM customer_his hhh
    , profile ppp
WHERE ppp.user_id = USER
AND hhh.cur_ts <= ppp.bgn_ts
AND hhh.cur_actn <> 'D'
AND NOT EXISTS
    (SELECT *
     FROM customer_his nnn
     WHERE nnn.prv_cust# = hhh.cust#
     AND nnn.prv_ts = hhh.cur_ts
     AND nnn.cur_ts <= ppp.bgn_ts);
```

The above sample schema shows just one table, but it can easily be extended to support every table is a very large application. One could even write some scripts to make the creation of the history tables, triggers, and views, all but automatic.

## 20.5. Limitations

The above schema has the following limitations:

- Every data table has to have a unique key.
- The cost of every insert, update, and delete, is essentially doubled.
- Data items that are updated very frequently (e.g. customer daily balance) may perform poorly when queried because many rows will have to be processed in order to find the one that has not been replaced.
- The view uses the USER special register, which may not be unique per actual user.

## 20.6. Multiple Versions of the World

The next design is similar to the previous, but we are also going to allow users to both see and change the world - as it was in the past, and as it is now, without affecting the real-world data. These extra features require a much more complex design:

- We cannot use a base table and a related history table, as we did above. Instead we have just the latter, and use both views and INSTEAD OF triggers to make the users think that they are really seeing and/or changing the former.
- We need a version table - to record when the data in each version (i.e. virtual copy of the real world) separates from the real world data.
- Data integrity features, like referential integrity rules, have to be hand-coded in triggers, rather that written using standard Db2 code.

### 20.6.1. Version Table

The following table has one row per version created:

*Version table*

```
CREATE TABLE version
( vrsn INTEGER NOT NULL
, vrsn_bgn_ts TIMESTAMP NOT NULL
, CONSTRAINT version1 CHECK(vrsn >= 0)
, CONSTRAINT version2 CHECK(vrsn < 1000000000)
, PRIMARY KEY(vrsn));
```

The following rules apply to the above:

- Each version has a unique number. Up to one billion can be created.
- Each version must have a begin-timestamp, which records at what point in time it separates from the real world. This value must be ⇐ the current time.
- Rows cannot be updated or deleted in this table - only inserted. This rule is necessary to ensure that we can always trace all changes - in every version.
- The real-world is deemed to have a version number of zero, and a begin-timestamp value of high-values.

### 20.6.2. Profile Table

The following profile table has one row per user (i.e. USER special register) that reads from or changes the data tables. It records what version the user is currently using (note: the version timestamp data is maintained using triggers):

*Profile table*

```
CREATE TABLE profile
( user_id     VARCHAR(10) NOT NULL
, vrsn        INTEGER     NOT NULL
, vrsn_bgn_ts TIMESTAMP   NOT NULL
, CONSTRAINT profile1 FOREIGN KEY(vrsn)
      REFERENCES version(vrsn)
      ON DELETE RESTRICT
, PRIMARY KEY(user_id));
```

### 20.6.3. Customer (data) Table

Below is a typical data table. This one holds customer data:

*Customer table*

```
CREATE TABLE customer_his
( cust#     INTEGER     NOT NULL
, cust_name CHAR(10)    NOT NULL
, cust_mgr  CHAR(10)
, cur_ts    TIMESTAMP   NOT NULL
, cur_vrsn  INTEGER     NOT NULL
, cur_actn  CHAR(1)     NOT NULL
, cur_user  VARCHAR(10) NOT NULL
, prv_cust# INTEGER
, prv_ts    TIMESTAMP
, prv_vrsn  INTEGER
, CONSTRAINT customer1 FOREIGN KEY(cur_vrsn)
      REFERENCES version(vrsn)
      ON DELETE RESTRICT
, CONSTRAINT customer2 CHECK(cur_actn IN ('I','U','D'))
, PRIMARY KEY(cust#,cur_vrsn,cur_ts));

CREATE INDEX customer_x2
ON customer_his(prv_cust#
              , prv_ts
              , prv_vrsn);
```

Note the following:

- The first three fields are the only ones that the user will see.
- The users will never update this table directly. They will make changes to a view of the table, which will then invoke INSTEAD OF triggers.
- The foreign key check (on version) can be removed - if it is forbidden to ever delete any version. This check stops the removal of versions that have changed data.
- The constraint on CUR_ACTN is just a double-check - to make sure that the triggers that will maintain this table do not have an error. It can be removed, if desired.
- The secondary index will make the following view more efficient.

The above table has the following hidden fields:

- CUR-TS: The current timestamp of the change.
- CUR-VRSN: The version in which change occurred. Zero implies reality.
- CUR-ACTN: The type of change (i.e. insert, update, or delete).
- CUR-USER: The user who made the change (for auditing purposes).
- PRV-CUST#: The previous customer number. This field enables one follow the sequence of changes for a given customer. The value is null if the action is an insert.
- PRV-TS: The timestamp of the last time the row was changed (null for inserts).
- PRV-VRNS: The version of the row being replaced (null for inserts).

### 20.6.4. Views

The following view displays the current state of the data in the above customer table – based on the version that the user is currently using:

*Customer view - 1 of 2*

```
CREATE VIEW customer_vw AS
SELECT *
FROM customer_his hhh
   , profile ppp
WHERE ppp.user_id = USER
AND hhh.cur_actn <> 'D'
AND  (  (ppp.vrsn = 0 AND hhh.cur_vrsn = 0)
     OR (ppp.vrsn > 0 AND hhh.cur_vrsn = 0 AND hhh.cur_ts < ppp.vrsn_bgn_ts)
     OR (ppp.vrsn > 0 AND hhh.cur_vrsn = ppp.vrsn)
     )
AND NOT EXISTS
  (SELECT *
   FROM customer_his nnn
   WHERE nnn.prv_cust# = hhh.cust#
   AND nnn.prv_ts = hhh.cur_ts
   AND nnn.prv_vrsn = hhh.cur_vrsn
   AND (  (ppp.vrsn = 0 AND nnn.cur_vrsn = 0)
       OR (ppp.vrsn > 0 AND nnn.cur_vrsn = 0 AND nnn.cur_ts < ppp.vrsn_bgn_ts)
       OR (ppp.vrsn > 0 AND nnn.cur_vrsn = ppp.vrsn)
       )
  );
```

The above view shows all customer rows, as long as:

- The action was not a delete.
- The version is either zero (i.e. reality), or the user's current version.
- If the version is reality, then the current timestamp is < the version begin-timestamp (as duplicated in the profile table).
- There does not exist any row that "replaces" the current row (and that row has a current timestamp that is ⇐ to the profile (version) timestamp).

To make things easier for the users, we will create another view that sits on top of the above view. This one only shows the business fields:

*Customer view - 2 of 2*

```
CREATE VIEW customer AS
SELECT cust#
     , cust_name
     , cust_mgr
FROM customer_vw;
```

All inserts, updates, and deletes, are done against the above view, which then propagates down to the first view, whereupon they are trapped by INSTEAD OF triggers. The changes are then applied (via the triggers) to the underlying tables.

### 20.6.5. Insert Trigger

The following INSTEAD OF trigger traps all inserts to the first view above, and then applies the insert to the underlying table - with suitable modifications:

*Insert trigger*

```
CREATE TRIGGER customer_ins
INSTEAD OF INSERT ON customer_vw
REFERENCING NEW AS nnn
FOR EACH ROW
MODE Db2SQL
INSERT INTO customer_his
VALUES(nnn.cust#
     , nnn.cust_name
     , nnn.cust_mgr
     , CURRENT TIMESTAMP
     , (SELECT vrsn
        FROM profile
        WHERE user_id = USER)
     , CASE
         WHEN 0 < (SELECT COUNT(*)
                   FROM customer
                   WHERE cust# = nnn.cust#)
           THEN RAISE_ERROR('71001','ERROR: Duplicate cust#')
         ELSE 'I'
       END
     , USER
     , NULL
     , NULL
     , NULL);
```

Observe the following:

- The basic customer data is passed straight through.
- The current timestamp is obtained from Db2.
- The current version is obtained from the user's profile-table row.
- A check is done to see if the customer number is unique. One cannot use indexes to enforce such rules in this schema, so one has to code accordingly.
- The previous fields are all set to null.

### 20.6.6. Update Trigger

The following INSTEAD OF trigger traps all updates to the first view above, and turns them into an

insert to the underlying table - with suitable modifications:

*Update trigger*

```
CREATE TRIGGER customer_upd
INSTEAD OF UPDATE ON customer_vw
REFERENCING NEW AS nnn
            OLD AS ooo
FOR EACH ROW
MODE Db2SQL
INSERT INTO customer_his
VALUES (nnn.cust#
      , nnn.cust_name
      , nnn.cust_mgr
      , CURRENT TIMESTAMP
      , ooo.vrsn
      , CASE
          WHEN nnn.cust# <> ooo.cust#
            THEN RAISE_ERROR('72001','ERROR: Cannot change cust#')
          ELSE 'U'
        END
      , ooo.user_id
      , ooo.cust#
      , ooo.cur_ts
      , ooo.cur_vrsn);
```

In this particular trigger, updates to the customer number (i.e. business key column) are not allowed. This rule is not necessary, it simply illustrates how one would write such code if one so desired.

### 20.6.7. Delete Trigger

The following INSTEAD OF trigger traps all deletes to the first view above, and turns them into an insert to the underlying table - with suitable modifications:

*Delete trigger*

```
CREATE TRIGGER customer_del
INSTEAD OF DELETE ON customer_vw
REFERENCING OLD AS ooo
FOR EACH ROW
MODE Db2SQL
INSERT INTO customer_his
VALUES (ooo.cust#
      , ooo.cust_name
      , ooo.cust_mgr
      , CURRENT TIMESTAMP
      , ooo.vrsn
      , 'D'
      , ooo.user_id
      , ooo.cust#
      , ooo.cur_ts
      , ooo.cur_vrsn);
```

## 20.7. Summary

The only thing that the user need see in the above schema in the simplified (second) view that lists the business data columns. They would insert, update, and delete the rows in this view as if they were working on a real table. Under the covers, the relevant INSTEAD OF trigger would convert whatever they did into a suitable insert to the underlying table.

This schema supports the following:

- To do "what if" analysis, all one need do is insert a new row into the version table – with a begin timestamp that is the current time. This insert creates a virtual copy of every table in the application, which one can then update as desired.
- To do historical analysis, one simply creates a version with a begin-timestamp that is at some point in the past. Up to one billion versions are currently supported.
- To switch between versions, all one need do is update one's row in the profile table.
- One can use recursive SQL (not shown here) to follow the sequence of changes to any particular item, in any particular version.

This schema has the following limitations:

- Every data table has to have a unique (business) key.
- Data items that are updated very frequently (e.g. customer daily balance) may perform poorly when queried because many rows will have to be processed in order to find the one that has not been replaced.
- The views use the USER special register, which may not be unique per actual user.
- Data integrity features, like referential integrity rules, cascading deletes, and unique key checks, have to be hand-coded in the INSTEAD OF triggers.

- Getting the triggers right is quite hard. If the target application has many tables, it might be worthwhile to first create a suitable data-dictionary, and then write a script that generates as much of the code as possible.

# Chapter 21. Using SQL to Make SQL

This chapter describes how to use SQL to make SQL. For example, one might want to make DDL statements to create views on a set of tables.

## 21.1. Export Command

The following query will generate a set of queries that will count the rows in each of the selected Db2 catalogue views:

*Generate SQL to count rows*

```
SELECT        'SELECT COUNT(*) FROM '
       CONCAT RTRIM(tabschema)
       CONCAT '.'
       CONCAT tabname
       CONCAT ';'
FROM syscat.tables
WHERE tabschema = 'SYSCAT'
AND tabname LIKE 'N%'
ORDER BY tabschema
       , tabname;
```

*ANSWER*

```
SELECT COUNT(*) FROM SYSCAT.NAMEMAPPINGS;
SELECT COUNT(*) FROM SYSCAT.NODEGROUPDEF;
SELECT COUNT(*) FROM SYSCAT.NODEGROUPS;
```

If we wrap the above inside an EXPORT statement, and define no character delimiter, we will be able to create a file with the above answer - and nothing else. This could in turn be run as if were some SQL statement that we had written:

*Export generated SQL statements*

```
EXPORT TO C:\FRED.TXT OF DEL
MODIFIED BY NOCHARDEL
SELECT      'SELECT COUNT(*) FROM '
     CONCAT RTRIM(tabschema)
     CONCAT '.'
     CONCAT tabname
     CONCAT ';'
FROM syscat.tables
WHERE tabschema = 'SYSCAT'
AND tabname LIKE 'N%'
ORDER BY tabschema
       , tabname;
```

## 21.2. Export Command Notes

The key EXPORT options used above are:

- The file name is "C\FRED.TXT".
- The data is sent to a delimited (i.e. DEL) file.
- The delimited output file uses no character delimiter (i.e. NOCHARDEL).

The remainder of this chapter will assume that we are using the EXPORT command, and will describe various ways to generate more elaborate SQL statements.

## 21.3. SQL to Make SQL

The next query is the same as the prior two, except that we have added the table name to each row of output:

*Generate SQL to count rows*

```
SELECT    'SELECT '''
   CONCAT tabname
   CONCAT ''', COUNT(*) FROM '
   CONCAT RTRIM(tabschema)
   CONCAT '.'
   CONCAT tabname
   CONCAT ';'
FROM syscat.tables
WHERE tabschema = 'SYSCAT'
AND tabname LIKE 'N%'
ORDER BY tabschema
       , tabname;
```

*ANSWER*

```
SELECT 'NAMEMAPPINGS', COUNT(*) FROM SYSCAT.NAMEMAPPINGS;
SELECT 'NODEGROUPDEF', COUNT(*) FROM SYSCAT.NODEGROUPDEF;
SELECT 'NODEGROUPS', COUNT(*) FROM SYSCAT.NODEGROUPS;
```

We can make more readable output by joining the result set to four numbered rows, and then breaking the generated query down into four lines:

*Generate SQL to count rows*

```
WITH temp1 (num) AS
(VALUES (1), (2), (3), (4))
SELECT CASE num
         WHEN 1 THEN 'SELECT ''' || tabname || ''' AS tname'
         WHEN 2 THEN '     , COUNT(*)' || ' AS #rows'
         WHEN 3 THEN 'FROM ' || RTRIM(tabschema) || '.' || tabname || ';'
         WHEN 4 THEN ''
       END
FROM syscat.tables
   , temp1
WHERE tabschema = 'SYSCAT',
AND tabname LIKE 'N%'
ORDER BY tabschema
       , tabname
       , num;
```

ANSWER

```
SELECT 'NAMEMAPPINGS' AS tname
     , COUNT(*) AS #rows
FROM SYSCAT.NAMEMAPPINGS;
SELECT 'NODEGROUPDEF' AS tname
     , COUNT(*) AS #rows
FROM SYSCAT.NODEGROUPDEF;
SELECT 'NODEGROUPS' AS tname
     , COUNT(*) AS #rows
FROM SYSCAT.NODEGROUPS;
```

So far we have generated separate SQL statements for each table that matches. But imagine that instead we wanted to create a single statement that processed all tables. For example, we might want to know the sum of the rows in all of the matching tables. There are two ways to do this, but neither of them are very good:

- We can generate a single large query that touches all of the matching tables. A query can be up to 2MB long, so we could reliably use this technique as long as we had less than about 5,000 tables to process.
- We can declare a global temporary table, then generate insert statements (one per matching table) that insert a count of the rows in the table. After running the inserts, we can sum the counts in the temporary table.

The next example generates a single query that counts all of the rows in the matching tables:

*Generate SQL to count rows (all tables)_*

```
WITH temp1 (num) AS
(VALUES (1), (2), (3), (4))
 SELECT CASE num
          WHEN 1 THEN 'SELECT SUM(C1)'
          when 2 then 'FROM ('
          WHEN 3 THEN 'SELECT COUNT(*) AS C1 FROM '
               CONCAT RTRIM(tabschema)
               CONCAT '.'
               CONCAT tabname
               CONCAT CASE dd
                        WHEN 1 THEN ''
                        ELSE ' UNION ALL'
                      END
          WHEN 4 THEN ') AS xxx;'
        END
 FROM (SELECT tab.*
            , ROW_NUMBER() OVER(ORDER BY tabschema ASC
                                       , tabname ASC) AS aa
            , ROW_NUMBER() OVER(ORDER BY tabschema DESC
                                       , tabname DESC) AS dd
       FROM syscat.tables tab
       WHERE tabschema = 'SYSCAT'
       AND tabname LIKE 'N%') AS xxx
    , emp1
WHERE (num <= 2 AND aa = 1)
OR    (num = 3)
OR    (num = 4 AND dd = 1)
ORDER BY tabschema ASC
       , tabname ASC
       , num ASC;
```

ANSWER

```
SELECT SUM(C1)
FROM (SELECT COUNT(*) AS C1 FROM SYSCAT.NAMEMAPPINGS UNION ALL
      SELECT COUNT(*) AS C1 FROM SYSCAT.NODEGROUPDEF UNION ALL
      SELECT COUNT(*) AS C1 FROM SYSCAT.NODEGROUPS
) AS xxx;
```

The above query works as follows:

- A temporary table (i.e. temp1) is generated with one column and four rows.
- A nested table expression (i.e. xxx) is created with the set of matching rows (tables).
- Within the nested table expression the ROW_NUMBER function is used to define two new columns. The first will have the value 1 for the first matching row, and the second will have the value 1 for the last matching row.

- The xxx and temp1 tables are joined. Two new rows (i.e. num ⇐ 2) are added to the front, and one new row (i.e. num = 4) is added to the back.
- The first two new rows (i.e. num = 1 and 2) are used to make the first part of the generated query.
- The last new row (i.e. num = 4) is used to make the tail end of the generated query.
- All other rows (i.e. num = 3) are used to create the core of the generated query.

In the above query no SQL is generated if no rows (tables) match. Alternatively, we might want to generate a query that returns zero if no rows match.

# Chapter 22. Running SQL Within SQL

This chapter describes how to generate and run SQL statements within SQL statements.

## 22.1. Introduction

Consider the following query:

*Sample query*

```
SELECT empno
     , lastname
     , workdept
     , salary
FROM employee
WHERE empno = '000250';
```

The above query exhibits all the usual strengths and weaknesses of the SQL language. It is easy to understand, simple to write, and assuming suitable indexes, efficient to run. But the query is annoyingly rigid in the sense that the both the internal query logic (i.e. which rows to fetch from what tables), and the set of columns to be returned, are fixed in the query syntax. Reasonably intelligent programs accessing suitably well-structured data might want to run queries like the following:

*Sample pseudo-query*

```
SELECT all-columns
FROM all-relevant-tables
WHERE all-predicates-are-true
```

It would of course be possible to compose the required query in the program and then run it. But there are some situations where it would be nice if we could also generate and then run the above pseudo-query inside the SQL language itself. This can be done, if there are two simple enhancements to the language:

- The ability to generate and run SQL within SQL.
- A way to make the query output-column-independent.

## 22.2. Generate SQL within SQL

To test the first concept above I wrote some very simple user-defined scalar functions (see Db2 SQL Functions) that enable one to generate and run SQL within SQL. In these functions the first row/column value fetched is returned. To illustrate, consider the following pseudoquery:

*Sample pseudo-query*

```
SELECT COUNT(*)
FROM all-relevant-tables
WHERE empno = '000250';
```

In the above pseudo-query we want to count all matching rows in all matching tables where the EMPNO is a given value. If we use the Db2 catalogue tables as a source dictionary, and we call a user-defined scalar function that can run SQL within SQL (see Db2 SQL Functions for the function definition), we can write the following query:

*Count matching rows in all matching tables*

```
SELECT CHAR(tabname,15) AS tabname
     , get_INTEGER(
          ' SELECT COUNT(*)' ||
          ' FROM ' || tabschema || '.' || tabname ||
          ' WHERE ' || colname || ' = ''000250'''
       ) AS num_rows
FROM syscat.columns
WHERE tabschema = USER
AND colname = 'EMPNO'
AND typename = 'CHARACTER'
ORDER BY tabname;
```

*ANSWER*

| TABNAME | NUM_ROWS |
|---|---|
| EMP_PHOTO | 0 |
| VEMP | 1 |
| VEMPDPT1 | 1 |
| VEMPPROJACT | 9 |
| VSTAFAC2 | 9 |

## 22.3. Make Query Column-Independent

The second issue to address was how to make the SQL language output-column-independent. This capability is needed in order to support the following type of pseudo-query:

*Sample pseudo-query*

```
SELECT all-columns
FROM all-relevant-tables
WHERE empno = '000250';
```

The above query cannot be written in SQL because the set of columns to be returned can not be

determined until the set of matching tables are identified. To get around this constraint, I wrote a very simple Db2 table function in Java (see Transpose Function) that accepts any valid query as input, runs it, and then returns all of the rows and columns fetched. But before returning anything, the function transposes each row/column instance into a single row – with a set of fixed columns returned that describe each row/column data instance. The function is used below to run the above pseudo-query:

*Transpose query output*

```
WITH temp1 AS
(SELECT tabname
      , VARCHAR(
            ' SELECT *' ||
            ' FROM ' || tabschema || '.' || tabname ||
            ' WHERE ' || colname || ' = ''000250'''
        ) AS SQL_text
FROM syscat.columns
WHERE tabschema = USER
AND colname = 'EMPNO'
AND typename = 'CHARACTER'
)
SELECT CHAR(t1.tabname,10) AS tabname
  , t2.row_number AS row#
  , t2.col_num AS col#
  , CHAR(t2.col_name,15) AS colname
  , CHAR(t2.col_type,15) AS coltype
  , CHAR(t2.col_value,20) AS colvalue
FROM temp1 t1
   , TABLE(tab_transpose(sql_text)) AS t2
ORDER BY t1.tabname
       , t2.row_number
       , t2.col_num;
```

Below are the first three "rows" of the answer:

| TABNAME | ROW# | COL# | COLNAME | COLTYPE | COLVALUE |
|---|---|---|---|---|---|
| EMPLOYEE | 1 | 1 | EMPNO | CHAR | 000250 |
| EMPLOYEE | 1 | 2 | FIRSTNME | VARCHAR | DANIEL |
| EMPLOYEE | 1 | 3 | MIDINIT | CHAR | S |
| EMPLOYEE | 1 | 4 | LASTNAME | VARCHAR | SMITH |
| EMPLOYEE | 1 | 5 | WORKDEPT | CHAR | D21 |
| EMPLOYEE | 1 | 6 | PHONENO | CHAR | 0961 |
| EMPLOYEE | 1 | 7 | HIREDATE | DATE | 1999-10-30 |
| EMPLOYEE | 1 | 8 | JOB | CHAR | CLERK |
| EMPLOYEE | 1 | 9 | EDLEVEL | SMALLINT | 15 |

| TABNAME | ROW# | COL# | COLNAME | COLTYPE | COLVALUE |
|---|---|---|---|---|---|
| EMPLOYEE | 1 | 10 | SEX | CHAR | M |
| EMPLOYEE | 1 | 11 | BIRTHDATE | DATE | 1969-11-12 |
| EMPLOYEE | 1 | 12 | SALARY | DECIMAL | 49180.00 |
| EMPLOYEE | 1 | 13 | BONUS | DECIMAL | 400.00 |
| EMPLOYEE | 1 | 14 | COMM | DECIMAL | 1534.00 |
| EMPPROJACT | 1 | 1 | EMPNO | CHAR | 000250 |
| EMPPROJACT | 1 | 2 | PROJNO | CHAR | AD3112 |
| EMPPROJACT | 1 | 3 | ACTNO | SMALLINT | 60 |
| EMPPROJACT | 1 | 4 | EMPTIME | DECIMAL | 1.00 |
| EMPPROJACT | 1 | 5 | EMSTDATE | DATE | 2002-01-01 |
| EMPPROJACT | 1 | 6 | EMENDATE | DATE | 2002-02-01 |
| EMPPROJACT | 2 | 1 | EMPNO | CHAR | 000250 |
| EMPPROJACT | 2 | 2 | PROJNO | CHAR | AD3112 |
| EMPPROJACT | 2 | 3 | ACTNO | SMALLINT | 60 |
| EMPPROJACT | 2 | 4 | EMPTIME | DECIMAL | 0.50 |
| EMPPROJACT | 2 | 5 | EMSTDATE | DATE | 2002-02-01 |
| EMPPROJACT | 2 | 6 | EMENDATE | DATE | 2002-03-15 |

## 22.4. Business Uses

At this point, I've got an interesting technical solution looking for a valid business problem. Some possible uses follow:

### 22.4.1. Frictionless Query

Imagine a relational database application where the table definitions are constantly changing. The programs using the data are able adapt accordingly, in which case the intermediate SQL queries have to also be equally adaptable. The application could maintain a data dictionary that was updated in sync with the table changes. Each query would reference the dictionary at the start of its processing, and then build the main body of the query (i.e. that which obtains the desired application data) as needed. I did some simple experiments using this concept. It worked, but I could see no overwhelming reason why one would use it, as opposed to building the query external to Db2, and then running it.

### 22.4.2. Adaptive Query

One could write a query where the internal query logic changed – depending on what data was encountered along the way. I tested this concept, and found that it works, but one still needs to define the general processing logic of the query somewhere. It was often easier to code a series of optional joins (in the query) to get the same result.

### 22.4.3. Meta-Data to Real-Data Join

A meta-data to real-data join can only be done using the SQL enhancements described above.

Some examples of such a join include:

- List all tables containing a row where EMPID = '123'.
- List all rows (in any table) that duplicate a given row.
- Confirm that two "sets of tables" have identical data.
- Scan all plan-tables looking for specific access paths.
- Find the largest application table that has no index.

These types of query are relatively rare, but they certainly do exist, and they are legitimate business queries.

### 22.4.4. Meta Data Dictionaries

In the above examples the Db2 catalogue was used as the source of meta-data that describes the relationships between the tables accessed by the query. This works up to a point, but the Db2 catalogue is not really designed for this task. Thus it would probably be better to use a purpose-built meta-data dictionary. Whenever application tables were changed, the meta-data dictionary would be updated accordingly - or might in fact be the source of the change. SQL queries generated using the meta-data dictionary would automatically adjust as the table changes were implemented.

## 22.5. Db2 SQL Functions

This section describes how to join meta-data to real data in a single query. In other words, a query will begin by selecting a list of tables from the Db2 catalogue. It will then access each table in the list. Such a query cannot be written using ordinary SQL, because the set of tables to be accessed is not known to the statement. But it can be written if the query references a very simple user-defined scalar function and related stored procedure. To illustrate, the following query will select a list of tables, and for each matching table get a count of the rows in the same:

*List tables, and count rows in same*

```
SELECT CHAR(tabschema,8) AS schema
     , CHAR(tabname,20) AS tabname
     , return_INTEGER(
          'SELECT COUNT(*) ' ||
          'FROM ' || tabschema || '.' || tabname
       ) AS #rows
FROM syscat.tables
WHERE tabschema = 'SYSCAT'
AND tabname LIKE 'RO%'
ORDER BY tabschema
       , tabname
FOR FETCH ONLY
WITH UR;
```

*ANSWER*

| SCHEMA | TABNAME | #ROWS |
|---|---|---|
| SYSCAT | ROUTINEAUTH | 168 |
| SYSCAT | ROUTINEDEP | 41 |
| SYSCAT | ROUTINEPARMS | 2035 |
| SYSCAT | ROUTINES | 314 |

## 22.6. Function and Stored Procedure Used

The above query calls a user-defined scalar function called return_INTEGER that accepts as input any valid single-column query and returns (you guessed it) an integer value that is the first row fetched by the query. The function is actually nothing more than a stub:

*return_INTEGER function*

```
CREATE FUNCTION return_INTEGER (in_stmt VARCHAR(4000))
RETURNS INTEGER
LANGUAGE SQL
READS SQL DATA
NO EXTERNAL ACTION
BEGIN ATOMIC
  DECLARE out_val INTEGER;
  CALL return_INTEGER(in_stmt, out_val);
  RETURN out_val;
END
```

The real work is done by a stored procedure that is called by the function:

*return_INTEGER stored procedure*

```
CREATE PROCEDURE return_INTEGER (IN in_stmt VARCHAR(4000)
                                 , OUT out_val INTEGER)
LANGUAGE SQL
READS SQL DATA
NO EXTERNAL ACTION
BEGIN
  DECLARE c1 CURSOR FOR s1;
  PREPARE s1 FROM in_stmt;
  OPEN c1;
  FETCH c1 INTO out_val;
  CLOSE c1;
  RETURN;
END
```

The combined function and stored-procedure logic goes as follow:

- Main query calls function - sends query text.
- Function calls stored-procedure - sends query text.
- Stored-procedure prepares, opens, fetches first row, and then closes query.
- Stored procedure returns result of first fetch back to the function
- Function returns the result back to the main query.

## 22.7. Different Data Types

One needs to have a function and related stored-procedure for each column type that can be returned. Below is a DECIMAL example:

*return_DECIMAL function*

```
CREATE PROCEDURE return_DECIMAL (IN in_stmt VARCHAR(4000)
                                 , OUT out_val DECIMAL(31, 6))
LANGUAGE SQL
READS SQL DATA
NO EXTERNAL ACTION
BEGIN
  DECLARE c1 CURSOR FOR s1;
  PREPARE s1 FROM in_stmt;
  OPEN c1;
  FETCH c1 INTO out_val;
  CLOSE c1;
  RETURN;
END
```

*return_DECIMAL stored procedure*

```
CREATE FUNCTION return_DECIMAL (in_stmt VARCHAR(4000))
RETURNS DECIMAL(31, 6)
LANGUAGE SQL
READS SQL DATA
NO EXTERNAL ACTION
BEGIN ATOMIC
  DECLARE out_val DECIMAL(31,6);
  CALL return_DECIMAL(in_stmt,out_val);
  RETURN out_val;
END
```

## 22.8. Usage Examples

The query below lists those tables that have never had RUNSTATS run (i.e. the stats-time is null), and that currently have more than 1,000 rows:

*List tables never had RUNSTATS*

```
SELECT CHAR(tabschema,8) AS schema
     , CHAR(tabname,20) AS tabname
     , #rows
FROM (SELECT tabschema
           , tabname
           , return_INTEGER(
               ' SELECT COUNT(*)' ||
               ' FROM ' || tabschema || '.' || tabname ||
               ' FOR FETCH ONLY WITH UR'
             ) AS #rows
      FROM syscat.tables tab
      WHERE tabschema LIKE 'SYS%'
      AND type = 'T'
      AND stats_time IS NULL) AS xxx
WHERE #rows > 1000
ORDER BY #rows DESC
FOR FETCH ONLY
WITH UR;
```

*ANSWER*

| SCHEMA | TABNAME | #ROWS |
|---|---|---|
| SYSIBM | SYSCOLUMNS | 3518 |
| SYSIBM | SYSROUTINEPARMS | 2035 |

## 22.9. Efficient Queries

The query shown above would typically process lots of rows, but this need not be the case. The next example lists all tables with a department column and at least one row for the 'A00' department. Only a single matching row is fetched from each table, so as long as there is a suitable index on the department column, the query should fly:

*List tables with a row for A00 department*

```
SELECT  CHAR(tab.tabname,15) AS tabname
      , CHAR(col.colname,10) AS colname
      , CHAR(COALESCE(return_VARCHAR(
                        ' SELECT ''Y''' ||
                        ' FROM ' || tab.tabschema || '.' || tab.tabname ||
                        ' WHERE ' || col.colname || ' = ''A00''' ||
                        ' FETCH FIRST 1 ROWS ONLY ' ||
                        ' OPTIMIZE FOR 1 ROW ' ||
                        ' WITH UR'
                    ), 'N'), 1) AS has_dept
FROM syscat.columns col
   , syscat.tables tab
WHERE col.tabschema = USER
AND col.colname IN ('DEPTNO', 'WORKDEPT')
AND col.tabschema = tab.tabschema
AND col.tabname = tab.tabname
AND tab.type = 'T'
FOR FETCH ONLY
WITH UR;
```

*ANSWER*

| TABNAME | COLNAME | HAS_DEPT |
|---|---|---|
| DEPARTMENT | DEPTNO | Y |
| EMPLOYEE | WORKDEPT | Y |
| PROJECT | DEPTNO | N |

The next query is the same as the previous, except that it only searches those matching tables that have a suitable index on the department field:

*List suitably-indexed tables with a row for A00 department*

```
SELECT CHAR(tab.tabname,15) AS tabname
     , CHAR(col.colname,10) AS colname
     , CHAR(COALESCE(return_VARCHAR(
                        ' SELECT ''Y''' ||
                        ' FROM ' || tab.tabschema || '.' || tab.tabname ||
                        ' WHERE ' || col.colname || ' = ''A00''' ||
                        ' FETCH FIRST 1 ROWS ONLY ' ||
                        ' OPTIMIZE FOR 1 ROW ' ||
                        ' WITH UR'
                    ), 'N'), 1) AS has_dept
FROM syscat.columns col
   , syscat.tables tab
WHERE col.tabschema = USER
AND col.colname IN ('DEPTNO', 'WORKDEPT')
AND col.tabschema = tab.tabschema
AND col.tabname = tab.tabname
AND tab.type = 'T'
AND col.colname IN (SELECT SUBSTR(idx.colnames, 2, LENGTH(col.colname))
                    FROM syscat.indexes idx
                    WHERE tab.tabschema = idx.tabschema
                    AND tab.tabname = idx.tabname)
FOR FETCH ONLY
WITH UR;
```

*ANSWER*

| TABNAME | COLNAME | HAS_DEPT |
|---|---|---|
| DEPARTMENT | DEPTNO | Y |

Using logic very similar to the above, one can efficiently ask questions like: "list all tables in the application that have references to customer-number 1234 in indexed fields". Even if the query has to process hundreds of tables, each with billions of rows, it should return an answer in less than ten seconds. In the above examples we knew what columns we wanted to process, but not the tables. But for some questions we don't even need to know the column name. For example, we could scan all indexed DATE columns in an application - looking for date values that are more than five years old. Once again, such a query should run in seconds.

# 22.10. Java Functions

We can do the same as the above by calling a user-defined-function that invokes a java program, but we can also do much more. This section will cover the basics.

## 22.10.1. Scalar Functions

The following code creates a user-defined scalar function that sends a query to a java program, and gets back the first row/column fetched when the query is run:

*CREATE FUNCTION code*

```
CREATE FUNCTION get_Integer(VARCHAR(4000))
RETURNS INTEGER
LANGUAGE JAVA
EXTERNAL NAME 'Graeme2!get_Integer'
PARAMETER STYLE Db2GENERAL
NO EXTERNAL ACTION
NOT DETERMINISTIC
READS SQL DATA
FENCED;
```

Below is the corresponding java code:

*CREATE FUNCTION java code*

```
import java.lang.*;
import COM.ibm.db2.app.*;
import java.sql.*;
import java.math.*;
import java.io.*;
public class Graeme2 extends UDF {
    public void get_Integer(String inStmt
                          , int outValue) throws Exception {
        try {
            Connection con = DriverManager.getConnection ("jdbc:default:connection");
            PreparedStatement stmt = con.prepareStatement(inStmt);
            ResultSet rs = stmt.executeQuery();
            if (rs.next() == true && rs.getString(1) != null) {
                set(2, rs.getInt(1));
            }
            rs.close();
            stmt.close();
            con.close();
            }
        catch (SQLException sqle) {
            setSQLstate("38999");
            setSQLmessage("SQLCODE = " + sqle.getSQLState());
            return;
        }
    }
}
```

Java Logic

- Establish connection.
- Prepare the SQL statement (i.e. input string).
- Execute the SQL statement (i.e. open cursor).

- If a row is found, and the value (of the first column) is not null, return value.
- Close cursor.
- Return.

### 22.10.2. Usage Example

*Java function usage example*

```
SELECT workdept AS dept
     , empno
     , salary
     , get_Integer(
          ' SELECT count(*)' ||
          ' FROM employee' ||
          ' where workdept = ''' || workdept || ''' ') AS #rows
FROM employee
WHERE salary < 35500
ORDER BY workdept
       , empno;
```

*ANSWER*

| DEPT | EMPNO | SALARY | #ROWS |
|---|---|---|---|
| E11 | 000290 | 35340.00 | 7 |
| E21 | 200330 | 35370.00 | 6 |
| E21 | 200340 | 31840.00 | 6 |

### 22.10.3. Tabular Functions

So far, all we have done in this chapter is get single values from tables. Now we will retrieve sets of rows from tables. To do this we need to define a tabular function:

*CREATE FUNCTION code*

```
CREATE FUNCTION tab_Varchar (VARCHAR(4000))
RETURNS TABLE (row_number INTEGER
             , row_value VARCHAR(254))
LANGUAGE JAVA
EXTERNAL NAME 'Graeme2!tab_Varchar'
PARAMETER STYLE DB2GENERAL
NO EXTERNAL ACTION
NOT DETERMINISTIC
DISALLOW PARALLEL
READS SQL DATA
FINAL CALL
FENCED;
```

Below is the corresponding java code. Observe that two columns are returned – a row-number and the value fetched:

*CREATE FUNCTION java code*

```
import java.lang.*;
import COM.ibm.db2.app.*;
import java.sql.*;
import java.math.*;
import java.io.*;
public class Graeme2 extends UDF {
    Connection con;
    Statement stmt;
    ResultSet rs;
    int rowNum;
    public void tab_Varchar(String inStmt, int outNumber, String outValue) throws
Exception {
    switch (getCallType())
    {
        case SQLUDF_TF_FIRST:
            break;
        case SQLUDF_TF_OPEN:
            rowNum = 1;
            try {
                con = DriverManager.getConnection("jdbc:default:connection");
                stmt = con.createStatement();
                rs = stmt.executeQuery(inStmt);
                }
            catch(SQLException sqle) {
                setSQLstate("38999");
                setSQLmessage("SQLCODE = " + sqle.getSQLState());
                return;
           }
           break;
        case SQLUDF_TF_FETCH:
            if (rs.next() == true) {
                set(2, rowNum);
                if (rs.getString(1) != null) {
                set(3, rs.getString(1));
                }
            rowNum++;
           }
           else {
            setSQLstate ("02000");
           }
           break;
        case SQLUDF_TF_CLOSE:
            rs.close();
            stmt.close();
            con.close();
            break;
```

```
        case SQLUDF_TF_FINAL:
            break;
        }
    }
}
```

### 22.10.4. Java Logic

Java programs that send data to Db2 table functions use a particular type of CASE logic to return the output data. In particular, a row is returned at the end of every FETCH process.

**OPEN:**

- Establish connection.
- Prepare the SQL statement (i.e. input string).
- Execute the SQL statement (i.e. open cursor).
- Set row-number variable to one.

**FETCH:**

- If row exists, set row-number output value.
- If value fetched is not null, set output value.
- Increment row-number variable.

**CLOSE:**

- Close cursor.
- Return.

### 22.10.5. Usage Example

The following query lists all EMPNO values that exist in more than four tables:

*Use Tabular Function*

```
WITH make_queries AS
(SELECT tab.tabschema
      , tab.tabname
      , ' SELECT EMPNO ' ||
        ' FROM ' || tab.tabschema || '.' || tab.tabname
        AS sql_text
FROM syscat.tables tab
   , syscat.columns col
WHERE tab.tabschema = USER
AND tab.type = 'T'
AND col.tabschema = tab.tabschema
AND col.tabname = tab.tabname
AND col.colname = 'EMPNO'
AND col.typename = 'CHARACTER'
AND col.length = 6)
, run_queries AS
(SELECT qqq.*
      , ttt.*
 FROM make_queries qqq,
      TABLE(tab_Varchar(sql_text)) AS ttt)
SELECT CHAR(row_value,10)                    AS empno
     , COUNT(*)                              AS #rows
     , COUNT(DISTINCT tabschema || tabname) AS #tabs
     , CHAR(MIN(tabname), 18)                AS min_tab
     , CHAR(MAX(tabname), 18)                AS max_tab
FROM run_queries
GROUP BY row_value
HAVING COUNT(DISTINCT tabschema || tabname) > 3
ORDER BY row_value
FOR FETCH ONLY WITH UR;
```

*ANSWER*

| EMPNO | #ROWS | #TABS | MIN_TAB | MAX_TAB |
|---|---|---|---|---|
| 000130 | 7 | 4 | EMP_PHOTO | EMPPROJACT |
| 000140 | 10 | 4 | EMP_PHOTO | EMPPROJACT |
| 000150 | 7 | 4 | EMP_PHOTO | EMPPROJACT |
| 000190 | 7 | 4 | EMP_PHOTO | EMPPROJACT |

### 22.10.6. Transpose Function

Below is some pseudo-code for a really cool query:

*Cool query pseudo-code*

```
SELECT all columns
FROM unknown tables
WHERE any unknown columns = '%ABC%'
```

In the above query we want to retrieve an unknown number of unknown types of columns (i.e. all columns in each matching row) from an unknown set of tables where any unknown column in the row equals 'ABC'. Needless to say, the various (unknown) tables will have differing types and numbers of columns. The above query is remarkably easy to write in SQL (see Transpose Function) and reasonably efficient to run, if we invoke a cute little java program that transposes columns into rows. The act of transposition means that each row/column instance retrieved becomes a separate row. So the following result:

*Select rows*

```
SELECT * FROM empprojact WHERE empno = '000150';
```

*ANSWER*

| EMPNO | PROJNO | ACTNO | EMPTIME | EMSTDATE | EMENDATE |
|---|---|---|---|---|---|
| 000150 | MA2112 | 60 | 1.00 | 01/01/2002 | 07/15/2002 |
| 000150 | MA2112 | 180 | 1.00 | 07/15/2002 | 02/01/2003 |

Becomes this result:

*Select rows – then transpose*

```
SELECT SMALLINT(row_number) AS row#
     , col_num AS col#
     , CHAR(col_name,13) AS col_name
     , CHAR(col_type,10) AS col_type
     , col_length AS col_len
     , SMALLINT(LENGTH(col_value)) AS val_len
     , SUBSTR(col_value,1,20) AS col_value
FROM TABLE(tab_Transpose(
             ' SELECT*' ||
             ' FROM empprojact' ||
             ' WHERE empno = ''000150'''')) AS ttt
ORDER BY 1,2;
```

*ANSWER*

| ROW# | COL# | COL_NAME | COL_TYPE | COL_LEN | VAL_LEN | COL_VALUE |
|---|---|---|---|---|---|---|
| 1 | 1 | EMPNO | CHAR | 6 | 6 | 000150 |
| 1 | 2 | PROJNO | CHAR | 6 | 6 | MA2112 |

| ROW# | COL# | COL_NAME | COL_TYPE | COL_LEN | VAL_LEN | COL_VALUE |
|---|---|---|---|---|---|---|
| 1 | 3 | ACTNO | SMALLINT | 6 | 2 | 60 |
| 1 | 4 | EMPTIME | DECIMAL | 7 | 4 | 1.00 |
| 1 | 5 | EMSTDATE | DATE | 10 | 10 | 2002-01-01 |
| 1 | 6 | EMENDATE | DATE | 10 | 10 | 2002-07-15 |
| 2 | 1 | EMPNO | CHAR | 6 | 6 | 000150 |
| 2 | 2 | PROJNO | CHAR | 6 | 6 | MA2112 |
| 2 | 3 | ACTNO | SMALLINT | 6 | 3 | 180 |
| 2 | 4 | EMPTIME | DECIMAL | 7 | 4 | 1.00 |
| 2 | 5 | EMSTDATE | DATE | 10 | 10 | 2002-07-15 |
| 2 | 6 | EMENDATE | DATE | 10 | 10 | 2003-02-01 |

The user-defined transpose function invoked above accepts a query as input. It executes the query then returns the query result as one row per row/column instance found. The function output table has the following columns:

- ROW_NUMBER: The number of the row fetched.
- NUM_COLS: The number of columns fetched per row.
- COL_NUM: The column-number for the current row. This value, in combination with the prior row-number value, identifies a unique output row.
- COL_NAME: The name of the data column - as given in the query. If there is no name, the value is the column number.
- COL_TYPE: The Db2 column-type for the value.
- COL_LENGTH: The Db2 column-length (note: not data item length) for the value.
- COL_VALUE: The row/column instance value itself. If the data column is too long, or of an unsupported type (e.g. CLOB, DBCLOB, or XML), null is returned.

The transpose function always returns the same set of columns, regardless of which table is being accessed. So we can use it to write a query where we don’t know which tables we want to select from. In the next example, we select all columns from all rows in all tables where the EMPNO column has a certain value:

*Select rows in any table – answer*

```
WITH make_queries AS
(SELECT tab.tabschema
      , tab.tabname,
      ' SELECT *' ||
      ' FROM ' || tab.tabname ||
      ' WHERE empno = ''000150'''
          AS sql_text
 FROM syscat.tables tab
    , syscat.columns col
 WHERE tab.tabschema = USER
 AND tab.type = 'T'
 AND col.tabschema = tab.tabschema
 AND col.tabname = tab.tabname
 AND col.colname = 'EMPNO'
 AND col.typename = 'CHARACTER'
 AND col.length = 6)
, run_queries AS
(SELECT qqq.*
     , ttt.*
FROM make_queries qqq
  , TABLE(tab_Transpose(sql_text)) AS ttt)
SELECT SUBSTR(tabname,1,11) AS tab_name
     , SMALLINT(row_number) AS row#
     , col_num AS col#
     , CHAR(col_name,13) AS col_name
     , CHAR(col_type,10) AS col_type
     , col_length AS col_len
     , SMALLINT(LENGTH(col_value)) AS val_len
     , SUBSTR(col_value,1,20) AS col_value
FROM run_queries
ORDER BY 1,2,3;
```

When we run the above, we get the following answer:

| TAB_NAME | ROW# | COL# | COL_NAME | COL_TYPE | COL_LEN | VAL_LEN | COL_VALUE |
|---|---|---|---|---|---|---|---|
| EMP_PHOTO | 1 | 1 | EMPNO | CHAR | 6 | 6 | 000150 |
| EMP_PHOTO | 1 | 2 | PHOTO_FORMAT | VARCHAR | 10 | 6 | bitmap |
| EMP_PHOTO | 1 | 3 | PICTURE | BLOB | 204800 | - | - |
| EMP_PHOTO | 1 | 4 | EMP_ROWID | CHAR | 40 | 40 | |

| TAB_NAME | ROW# | COL# | COL_NAME | COL_TYPE | COL_LEN | VAL_LEN | COL_VALUE |
|---|---|---|---|---|---|---|---|
| EMP_PHOTO | 2 | 1 | EMPNO | CHAR | 6 | 6 | 000150 |
| EMP_PHOTO | 2 | 2 | PHOTO_FORMAT | VARCHAR | 10 | 3 | gif |
| EMP_PHOTO | 2 | 3 | PICTURE | BLOB | 204800 | - | - |
| EMP_PHOTO | 2 | 4 | EMP_ROWID | CHAR | 40 | 40 | |
| EMP_RESUME | 1 | 1 | EMPNO | CHAR | 6 | 6 | 000150 |
| EMP_RESUME | 1 | 2 | RESUME_FORMAT | VARCHAR | 10 | 5 | ascii |
| EMP_RESUME | 1 | 3 | RESUME | CLOB | 5120 | - | - |
| EMP_RESUME | 1 | 4 | EMP_ROWID | CHAR | 40 | 40 | |
| EMP_RESUME | 2 | 1 | EMPNO | CHAR | 6 | 6 | 000150 |
| EMP_RESUME | 2 | 2 | RESUME_FORMAT | VARCHAR | 10 | 4 | html |
| | EMP_RESUME | 2 | 3 | RESUME | CLOB | 5120 | - |
| - | EMP_RESUME | 2 | 4 | EMP_ROWID | CHAR | 40 | 40 |
| | EMPLOYEE | 1 | 1 | EMPNO | CHAR | 6 | 6 |
| 000150 | EMPLOYEE | 1 | 2 | FIRSTNME | VARCHAR | 12 | 5 |
| BRUCE | EMPLOYEE | 1 | 3 | MIDINIT | CHAR | 1 | 1 |
| | EMPLOYEE | 1 | 4 | LASTNAME | VARCHAR | 15 | 7 |
| ADAMSON | EMPLOYEE | 1 | 5 | WORKDEPT | CHAR | 3 | 3 |
| D11 | EMPLOYEE | 1 | 6 | PHONENO | CHAR | 4 | 4 |
| 4510 | EMPLOYEE | 1 | 7 | HIREDATE | DATE | 10 | 10 |
| 2002-02-12 | EMPLOYEE | 1 | 8 | JOB | CHAR | 8 | 8 |
| DESIGNER | EMPLOYEE | 1 | 9 | EDLEVEL | SMALLINT | 6 | 2 |
| 16 | EMPLOYEE | 1 | 10 | SEX | CHAR | 1 | 1 |

| TAB_NAME | ROW# | COL# | COL_NAME | COL_TYPE | COL_LEN | VAL_LEN | COL_VALUE |
|---|---|---|---|---|---|---|---|
| M | EMPLOYEE | 1 | 11 | BIRTHDATE | DATE | 10 | 10 |
| 1977-05-17 | EMPLOYEE | 1 | 12 | SALARY | DECIMAL | 11 | 8 |
| 55280.00 | EMPLOYEE | 1 | 13 | BONUS | DECIMAL | 11 | 6 |
| 500.00 | EMPLOYEE | 1 | 14 | COMM | DECIMAL | 11 | 7 |
| 2022.00 | EMPPROJACT | 1 | 1 | EMPNO | CHAR | 6 | 6 |
| 000150 | EMPPROJACT | 1 | 2 | PROJNO | CHAR | 6 | 6 |
| MA2112 | EMPPROJACT | 1 | 3 | ACTNO | SMALLINT | 6 | 2 |
| 60 | EMPPROJACT | 1 | 4 | EMPTIME | DECIMAL | 7 | 4 |
| 1.00 | EMPPROJACT | 1 | 5 | EMSTDATE | DATE | 10 | 10 |
| 2002-01-01 | EMPPROJACT | 1 | 6 | EMENDATE | DATE | 10 | 10 |
| 2002-07-15 | | EMPPROJACT | 2 | 1 | EMPNO | CHAR | 6 |
| 6 | 000150 | EMPPROJACT | 2 | 2 | PROJNO | CHAR | 6 |
| 6 | MA2112 | EMPPROJACT | 2 | 3 | ACTNO | SMALLINT | 6 |
| 3 | 180 | EMPPROJACT | 2 | 4 | EMPTIME | DECIMAL | 7 |
| 4 | 1.00 | EMPPROJACT | 2 | 5 | EMSTDATE | DATE | 10 |
| 10 | 2002-07-15 | EMPPROJACT | 2 | 6 | EMENDATE | DATE | 10 |

We are obviously on a roll, so now we will write the pseudo-query that we began this chapter with (see Transpose Function). We will fetch every row/column instance in all matching tables where any qualifying column in the row is a particular value.

## 22.11. Query Logic

- Define the search parameters.
- Get the list of matching tables and columns to search.

- Recursively work through the list of columns to search (for each table), building a search query with multiple EQUAL predicates – one per searchable column.
- Run the generated queries (i.e. the final line of generated query for each table).
- Select the output.

Now for the query:

*Select rows in any table – then transpose*

```
WITH search_values (search_type,search_length,search_value) AS
(VALUES ('CHARACTER',6,'000150'))
, list_columns AS
(SELECT val.search_value
      , tab.tabschema
      , tab.tabname
      , col.colname
      , ROW_NUMBER() OVER(PARTITION BY val.search_value
                                     , tab.tabschema
                                     , tab.tabname
                                     ORDER BY col.colname ASC) AS col_a
      , ROW_NUMBER() OVER(PARTITION BY val.search_value
                                     , tab.tabschema
                                     , tab.tabname
                                     ORDER BY col.colname DESC) AS col_d
 FROM search_values val
    , syscat.tables tab
    , syscat.columns col
 WHERE tab.tabschema = USER
 AND tab.type = 'T'
 AND tab.tabschema = col.tabschema
 AND tab.tabname = col.tabname
 AND col.typename = val.search_type
 AND col.length = val.search_length)
, make_queries (search_value, tabschema, tabname, colname, col_a, col_d, sql_text) AS
(SELECT tb1.*
      , VARCHAR(' SELECT *' ||
                ' FROM ' || tabname ||
                ' WHERE ' || colname || ' = ''' || search_value || ''''
          , 4000)
 FROM list_columns tb1
 WHERE col_a = 1
   UNION ALL
 SELECT tb2.*
      , mqy.sql_text || ' OR ' || tb2.colname || ' = ''' || tb2.search_value || ''''
 FROM list_columns tb2
    , make_queries mqy
 WHERE tb2.search_value = mqy.search_value
 AND tb2.tabschema = mqy.tabschema
 AND tb2.tabname = mqy.tabname
 AND tb2.col_a = mqy.col_a + 1)
```

```
, run_queries AS
(SELECT qqq.*
      , ttt.*
 FROM make_queries qqq
    , TABLE(tab_Transpose_4K(sql_text)) AS ttt
 WHERE col_d = 1)
SELECT SUBSTR(tabname,1,11)        AS tab_name
     , SMALLINT(row_number)        AS row#
     , col_num                     AS col#
     , CHAR(col_name,13)           AS col_name
     , CHAR(col_type,10)           AS col_type
     , col_length                  AS col_len
     , SMALLINT(LENGTH(col_value)) AS val_len
     , SUBSTR(col_value,1,20)      AS col_value
FROM run_queries
ORDER BY 1,2,3;
```

Below is the answer (with a few values truncated to fit):

| TAB_NAME | ROW# | COL# | COL_NAME | COL_TYPE | COL_LEN | VAL_LEN | COL_VALUE |
|---|---|---|---|---|---|---|---|
| EMP_PHOTO | 1 | 1 | EMPNO | CHAR | 6 | 6 | 000150 |
| EMP_PHOTO | 1 | 2 | PHOTO_FORMAT | VARCHAR | 10 | 6 | bitmap |
| EMP_PHOTO | 1 | 3 | PICTURE | BLOB | 204800 | - | - |
| EMP_PHOTO | 1 | 4 | EMP_ROWID | CHAR | 40 | 40 | |
| EMP_PHOTO | 2 | 1 | EMPNO | CHAR | 6 | 6 | 000150 |
| EMP_PHOTO | 2 | 2 | PHOTO_FORMAT | VARCHAR | 10 | 3 | gif |
| EMP_PHOTO | 2 | 3 | PICTURE | BLOB | 204800 | - | - |
| EMP_PHOTO | 2 | 4 | EMP_ROWID | CHAR | 40 | 40 | |
| EMP_RESUME | 1 | 1 | EMPNO | CHAR | 6 | 6 | 000150 |
| EMP_RESUME | 1 | 2 | RESUME_FORMAT | VARCHAR | 10 | 5 | ascii |
| EMP_RESUME | 1 | 3 | RESUME | CLOB | 5120 | - | - |

| TAB_NAME | ROW# | COL# | COL_NAME | COL_TYPE | COL_LEN | VAL_LEN | COL_VALUE |
|---|---|---|---|---|---|---|---|
| EMP_RESUME | 1 | 4 | EMP_ROWID | CHAR | 40 | 40 | |
| EMP_RESUME | 2 | 1 | EMPNO | CHAR | 6 | 6 | 000150 |
| EMP_RESUME | 2 | 2 | RESUME_FORMAT | VARCHAR | 10 | 4 | html |
| EMP_RESUME | 2 | 3 | RESUME | CLOB | 5120 | - | - |
| EMP_RESUME | 2 | 4 | EMP_ROWID | CHAR | 40 | 40 | |
| EMPLOYEE | 1 | 1 | EMPNO | CHAR | 6 | 6 | 000150 |
| EMPLOYEE | 1 | 2 | FIRSTNME | VARCHAR | 12 | 5 | BRUCE |
| EMPLOYEE | 1 | 3 | MIDINIT | CHAR | 1 | 1 | |
| EMPLOYEE | 1 | 4 | LASTNAME | VARCHAR | 15 | 7 | ADAMSON |
| EMPLOYEE | 1 | 5 | WORKDEPT | CHAR | 3 | 3 | D11 |
| EMPLOYEE | 1 | 6 | PHONENO | CHAR | 4 | 4 | 4510 |
| EMPLOYEE | 1 | 7 | HIREDATE | DATE | 10 | 10 | 2002-02-12 |
| EMPLOYEE | 1 | 8 | JOB | CHAR | 8 | 8 | DESIGNER |
| EMPLOYEE | 1 | 9 | EDLEVEL | SMALLINT | 6 | 2 | 16 |
| EMPLOYEE | 1 | 10 | SEX | CHAR | 1 | 1 | M |
| EMPLOYEE | 1 | 11 | BIRTHDATE | DATE | 10 | 10 | 1977-05-17 |
| EMPLOYEE | 1 | 12 | SALARY | DECIMAL | 11 | 8 | 55280.00 |
| EMPLOYEE | 1 | 13 | BONUS | DECIMAL | 11 | 6 | 500.00 |
| EMPLOYEE | 1 | 14 | COMM | DECIMAL | 11 | 7 | 2022.00 |
| EMPPROJACT | 1 | 1 | EMPNO | CHAR | 6 | 6 | 000150 |
| EMPPROJACT | 1 | 2 | PROJNO | CHAR | 6 | 6 | MA2112 |
| EMPPROJACT | 1 | 3 | ACTNO | SMALLINT | 6 | 2 | 60 |
| EMPPROJACT | 1 | 4 | EMPTIME | DECIMAL | 7 | 4 | 1.00 |

| TAB_NAME | ROW# | COL# | COL_NAME | COL_TYPE | COL_LEN | VAL_LEN | COL_VALUE |
|---|---|---|---|---|---|---|---|
| EMPPROJACT | 1 | 5 | EMSTDATE | DATE | 10 | 10 | 2002-01-01 |
| EMPPROJACT | 1 | 6 | EMENDATE | DATE | 10 | 10 | 2002-07-15 |
| EMPPROJACT | 2 | 1 | EMPNO | CHAR | 6 | 6 | 000150 |
| EMPPROJACT | 2 | 2 | PROJNO | CHAR | 6 | 6 | MA2112 |
|  | EMPPROJACT | 2 | 3 | ACTNO | SMALLINT | 6 | 3 |
| 180 | EMPPROJACT | 2 | 4 | EMPTIME | DECIMAL | 7 | 4 |
| 1.00 | EMPPROJACT | 2 | 5 | EMSTDATE | DATE | 10 | 10 |
| 2002-07-15 | EMPPROJACT | 2 | 6 | EMENDATE | DATE | 10 | 10 |
| 2003-02-01 | PROJECT | 1 | 1 | PROJNO | CHAR | 6 | 6 |
| MA2112 | PROJECT | 1 | 2 | PROJNAME | VARCHAR | 24 | 16 |
| W L ROBOT | PROJECT | 1 | 3 | DEPTNO | CHAR | 3 | 3 |
| D11 | PROJECT | 1 | 4 | RESPEMP | CHAR | 6 | 6 |
| 000150 | PROJECT | 1 | 5 | PRSTAFF | DECIMAL | 7 | 4 |
| 3.00 | PROJECT | 1 | 6 | PRSTDATE | DATE | 10 | 10 |
| 2002-01-01 | PROJECT | 1 | 7 | PRENDATE | DATE | 10 | 10 |
| 1982-12-01 | PROJECT | 1 | 8 | MAJPROJ | CHAR | 6 | 6 |

Below are the queries that were generated and run to get the above answer:

*Queries generated above*

```
SELECT * FROM ACT WHERE ACTKWD = '000150'
SELECT * FROM DEPARTMENT WHERE MGRNO = '000150'
SELECT * FROM EMP_PHOTO WHERE EMPNO = '000150'
SELECT * FROM EMP_RESUME WHERE EMPNO = '000150'
SELECT * FROM EMPLOYEE WHERE EMPNO = '000150'
SELECT * FROM EXPLAIN_OPERATOR WHERE OPERATOR_TYPE = '000150'
SELECT * FROM PROJACT WHERE PROJNO = '000150'
SELECT * FROM EMPPROJACT WHERE EMPNO = '000150' OR PROJNO = '000150'
SELECT * FROM PROJECT WHERE MAJPROJ = '000150' OR PROJNO = '000150' OR
    RESPEMP = '000150'
```

### 22.11.1. Function Definition

The Db2 user-defined tabular function that does the transposing is defined thus:

*Create transpose function*

```
CREATE FUNCTION tab_Transpose (VARCHAR(4000))
RETURNS TABLE (row_number INTEGER
             , num_cols SMALLINT
             , col_num SMALLINT
             , col_name VARCHAR(128)
             , col_type VARCHAR(128)
             , col_length INTEGER
             , col_value VARCHAR(254))
LANGUAGE JAVA
EXTERNAL NAME 'Graeme2!tab_Transpose'
PARAMETER STYLE Db2GENERAL
NO EXTERNAL ACTION
NOT DETERMINISTIC
DISALLOW PARALLEL
READS SQL DATA
FINAL CALL
FENCED;
```

#### Java Code

*CREATE FUNCTION java code*

```
import java.lang.*;
import COM.ibm.db2.app.*;
import java.sql.*;
import java.math.*;
import java.io.*;
public class Graeme2 extends UDF {
    Connection con;
    Statement stmt;
    ResultSet rs;
    ResultSetMetaData rsmtadta;
    int rowNum;
    int i;
    int outLength;
    short colNum;
    int colCount;
    String[] colName = new String[1100];
    String[] colType = new String[1100];
    int[] colSize = new int[1100];
    public void writeRow() throws Exception {
        set(2, rowNum);
        set(3, (short) colCount);
        set(4, colNum);
        set(5, colName[colNum]);
```

```
        set(6, colType[colNum]);
        set(7, colSize[colNum]);
        if (colType[colNum].equals("XML") ||
            colType[colNum].equals("BLOB") ||
            colType[colNum].equals("CLOB") ||
            colType[colNum].equals("DBLOB") ||
            colType[colNum].equals("GRAPHIC") ||
            colType[colNum].equals("VARGRAPHIC") ||
            colSize[colNum] > outLength) {
            // DON'T DISPLAY THIS VALUE
            return;
        }
        else if (rs.getString(colNum) != null) {
            // DISPLAY THIS COLUMN VALUE
            set(8, rs.getString(colNum));
        }
    }

    public void tab_Transpose(String inStmt
                            , int rowNumber
                            , short numColumns
                            , short outColNumber
                            , String outColName
                            , String outColtype
                            , int outColSize
                            , String outColValue) throws Exception {
        switch (getCallType()) {
            case SQLUDF_TF_FIRST:
                break;
            case SQLUDF_TF_OPEN:
                try {
                    con = DriverManager.getConnection("jdbc:default:connection");
                    stmt = con.createStatement();
                    rs = stmt.executeQuery(inStmt);
                    // GET COLUMN NAMES
                    rsmtadta = rs.getMetaData();
                    colCount = rsmtadta.getColumnCount();
                    for (i=1; i <= colCount; i++) {
                        colName[i] = rsmtadta.getColumnName(i);
                        colType[i] = rsmtadta.getColumnTypeName(i);
                        colSize[i] = rsmtadta.getColumnDisplaySize(i);
                    }
                    rowNum = 1;
                    colNum = 1;
                    outLength = 254;
                }
                catch(SQLException sqle) {
                    setSQLstate("38999");
                    setSQLmessage("SQLCODE = " + sqle.getSQLState());
                    return;
                }
```

```
                break;
            case SQLUDF_TF_FETCH:
                if (colNum == 1 && rs.next() == true) {
                    writeRow();
                    colNum++;
                    if (colNum > colCount) {
                        colNum = 1;
                        rowNum++;
                    }
                }
                else if (colNum > 1 && colNum <= colCount) {
                    writeRow();
                    colNum++;
                    if (colNum > colCount) {
                        colNum = 1;
                        rowNum++;
                    }
                }
                else {
                    setSQLstate ("02000");
                }
                break;
            case SQLUDF_TF_CLOSE:
                rs.close();
                stmt.close();
                con.close();
                break;
            case SQLUDF_TF_FINAL:
                break;
            }
        }
    }
```

## 22.12. Java Logic

OPEN (run once):

- Establish connection.
- Prepare the SQL statement (i.e. input string).
- Execute the SQL statement (i.e. open cursor).
- Get meta-data for each column returned by query.
- Set row-number and column-number variables to one.
- Set the maximum output length accepted to 254.

FETCH (run for each row/column instance):

- If row exists and column-number is 1, fetch row.

- For value is not null and of valid Db2 type, return row.
- Increment row-number and column-number variables.

CLOSE (run once):

- Close the cursor.
- Return.

## 22.13. Update Real Data using Meta-Data

Db2 does not allow one to do DML or DDL using a scalar function, but one can do something similar by calling a table function. Thus if the table function defined below is joined to in a query, the following happens:

- User query joins to table function - sends DML or DDL statement to be executed.
- Table function calls stored procedure - sends statement to be executed.
- Stored procedure executes statement.
- Stored procedure returns SQLCODE of statement to the table function.
- Table function joins back to the user query a single-row table with two columns: The SQLCODE and the original input statement.

Now for the code:

*Define function and stored-procedure*

```
CREATE PROCEDURE execute_immediate (IN in_stmt VARCHAR(1000)
                                  , OUT out_sqlcode INTEGER)
LANGUAGE SQL
MODIFIES SQL DATA
BEGIN
  DECLARE sqlcode INTEGER;
  DECLARE EXIT HANDLER FOR sqlexception
    SET out_sqlcode = sqlcode;
  EXECUTE IMMEDIATE in_stmt;
  SET out_sqlcode = sqlcode;
  RETURN;
END!

CREATE FUNCTION execute_immediate (in_stmt VARCHAR(1000))
RETURNS TABLE (sqltext VARCHAR(1000)
             , sqlcode INTEGER)
LANGUAGE SQL
MODIFIES SQL DATA
BEGIN ATOMIC
  DECLARE out_sqlcode INTEGER;
  CALL execute_immediate(in_stmt, out_sqlcode);
  RETURN VALUES (in_stmt, out_sqlcode);
END!
```

This example uses an "!" as the stmt delimiter.

This code is extremely dangerous! Use with care. As we shall see, it is very easy for the above code to do some quite unexpected.

### 22.13.1. Usage Examples

The following query gets a list of materialized query tables for a given table-schema that need to be refreshed, and then refreshes the table:

*Refresh matching tables*

```
WITH temp1 AS
(SELECT tabschema
      , tabname
 FROM syscat.tables
 WHERE tabschema = 'FRED'
 AND type = 'S'
 AND status = 'C'
 AND tabname LIKE '%DEPT%')
SELECT CHAR(tab.tabname, 20) AS tabname
     , stm.sqlcode AS sqlcode
     , CHAR(stm.sqltext,100) AS sqltext
FROM temp1 AS tab
   , TABLE(execute_immediate(
         'REFRESH TABLE ' ||RTRIM(tab.tabschema) || '.' || tab.tabname))AS stm
ORDER BY tab.tabname
WITH UR;
```

I had two matching tables that needed to be refreshed, so I got the following answer:

| TABNAME | SQLCODE | SQLTEXT |
|---|---|---|
| STAFF_DEPT1 | 0 | REFRESH TABLE FRED.STAFF_DEPT1 |
| STAFF_DEPT2 | 0 | REFRESH TABLE FRED.STAFF_DEPT2 |

Observe above that the set of matching tables to be refreshed was defined in a common-tableexpression, and then joined to the table function. It is very important that one always code thus, because in an ordinary join it is possible for the table function to be called before all of the predicates have been applied. To illustrate this concept, the next query is supposed to make a copy of two matching tables. The answer indicates that it did just this. But what it actually did was make copies of many more tables - because the table function was called before all of the predicates on SYSCAT.TABLES were applied. The other tables that were created don't show up in the query output, because they were filtered out later in the query processing:

*Create copies of tables - wrong*

```
SELECT CHAR(tab.tabname, 20) AS tabname
     , stm.sqlcode AS sqlcode
     , CHAR(stm.sqltext,100) AS sqltext
FROM syscat.tables AS tab
   , TABLE(execute_immediate(
             ' CREATE TABLE ' || RTRIM(tab.tabschema) || '.' || tab.tabname || '_C1'
||
             ' LIKE ' || RTRIM(tab.tabschema) || '.' || tab.tabname))AS stm
WHERE tab.tabschema = USER
AND tab.tabname LIKE 'S%'
ORDER BY tab.tabname
FOR FETCH ONLY
WITH UR;
```

*ANSWER*

| TABNAME | SQLCODE | SQLTEXT |
|---|---|---|
| SALES | 0 | CREATE TABLE FRED.SALES_C1 LIKE FRED.SALES |
| STAFF | 0 | CREATE TABLE FRED.STAFF_C1 LIKE FRED.STAFF |

The above is bad enough, but I once managed to do much worse. In a variation of the above code, the query created a copy, of a copy, of a copy, etc. The table function kept finding the table just created, and making a copy of it - until the TABNAME reached the length limit. The correct way to create a copy of a set of tables is shown below. In this query, the list of tables to be copied is identified in a common table expression before the table function is called:

*Create copies of tables - right*

```
WITH temp1 AS
(SELECT tabschema
      , tabname
 FROM syscat.tables
 WHERE tabschema = USER
 AND tabname LIKE 'S%')
SELECT CHAR(tab.tabname, 20) AS tabname
     , stm.sqlcode AS sqlcode
     , CHAR(stm.sqltext,100) AS sqltext
FROM temp1 tab
   , TABLE(execute_immediate(
             ' CREATE TABLE ' ||RTRIM(tab.tabschema) || '.' || tab.tabname || '_C1' ||
             ' LIKE ' || RTRIM(tab.tabschema) || '.' || tab.tabname))AS stm
ORDER BY tab.tabname
FOR FETCH ONLY
WITH UR;
```

*ANSWER*

| TABNAME | SQLCODE | SQLTEXT |
|---|---|---|
| SALES | 0 | CREATE TABLE FRED.SALES_C1 LIKE FRED.SALES |
| STAFF | 0 | CREATE TABLE FRED.STAFF_C1 LIKE FRED.STAFF |

The next example is similar to the previous, except that it creates a copy, and then populates the new table with the contents of the original table:

*Create copies of tables, then populate*

```
WITH temp0 AS
(SELECT RTRIM(tabschema) AS schema
      , tabname AS old_tabname
      , tabname || '_C2' AS new_tabname
 FROM syscat.tables
 WHERE tabschema = USER
 AND tabname LIKE 'S%')
, temp1 AS
(SELECT tab.*
      , stm.sqlcode AS sqlcode1
      , CHAR(stm.sqltext,200) AS sqltext1
 FROM temp0 AS tab
    , TABLE(execute_immediate(
                ' CREATE TABLE ' || schema || '.' || new_tabname ||
                ' LIKE ' || schema || '.' || old_tabname))AS stm)
, temp2 AS
(SELECT tab.*
      , stm.sqlcode AS sqlcode2
      , CHAR(stm.sqltext,200) AS sqltext2
 FROM temp1 AS tab
 , TABLE(execute_immediate(
           ' INSERT INTO ' || schema || '.' || new_tabname ||
           ' SELECT * FROM ' || schema || '.' || old_tabname))AS stm)
SELECT CHAR(old_tabname,20) AS tabname
     , sqlcode1
     , sqlcode2
FROM temp2
ORDER BY old_tabname
FOR FETCH ONLY
WITH UR;
```

*ANSWER*

| TABNAME | SQLCODE1 | SQLCODE2 |
|---|---|---|
| SALES | 0 | 0 |

| TABNAME | SQLCODE1 | SQLCODE2 |
|---|---|---|
| STAFF | 0 | 0 |

### 22.13.2. Query Processing Sequence

In order to explain the above, we need to understand in what sequence the various parts of a query are executed in order to avoid semantic ambiguity:

*Query Processing Sequence*

- FROM clause
- JOIN ON clause
- WHERE clause
- GROUP BY and aggregate
- HAVING clause
- SELECT list
- ORDER BY clause
- FETCH FIRST

Observe above that the FROM clause is resolved before any WHERE predicates are applied. This is why the query in Create copies of tables - wrong did the wrong thing.

# Chapter 23. Fun with SQL

In this chapter will shall cover some of the fun things that one can and, perhaps, should not do, using Db2 SQL. Read on at your own risk.

## 23.1. Creating Sample Data

If every application worked exactly as intended from the first, we would never have any need for test databases. Unfortunately, one often needs to builds test systems in order to both tune the application SQL, and to do capacity planning. In this section we shall illustrate how very large volumes of extremely complex test data can be created using relatively simple SQL statements.

Good Sample Data is

- Reproducible.
- Easy to make.
- Similar to Production:
  - Same data volumes (if needed).
  - Same data distribution characteristics.

### 23.1.1. Data Generation

Create the set of integers between zero and one hundred. In this statement we shall use recursive coding to expand a single value into many more.

*Use recursion to get list of 100 numbers*

```
WITH temp1 (col1) AS
(VALUES 0
   UNION ALL
 SELECT col1 + 1
 FROM temp1
 WHERE col1 + 1 < 100)
SELECT *
FROM temp1;
```

*ANSWER*

| COL1 |
|---|
| 1 |
| 2 |
| 3 |
| etc |

Instead of coding a recursion join every time, we use the table function described on Generating

Numbers to create the required rows. Assuming that the function exists, one would write the following:

*Use user-defined-function to get list of 100 numbers*

```
SELECT *
FROM TABLE(NumList(100)) AS xxx;
```

### 23.1.2. Make Reproducible Random Data

So far, all we have done is create sets of fixed values. These are usually not suitable for testing purposes because they are too consistent. To mess things up a bit we need to use the RAND function, which generates random numbers in the range of zero to one inclusive. In the next example we will get a (reproducible) list of five random numeric values:

*Use RAND to create pseudo-random numbers*

```
WITH temp1 (s1, r1) AS
(VALUES (0, RAND(1))
   UNION ALL
 SELECT s1+1
     , RAND()
 FROM temp1
 WHERE s1+1 < 5)
SELECT SMALLINT(s1)    AS seq#
     , DECIMAL(r1,5,3) AS ran1
FROM temp1;
```

*ANSWER*

| SEQ# | RAN1 |
|---|---|
| 0 | 0.001 |
| 1 | 0.563 |
| 2 | 0.193 |
| 3 | 0.808 |
| 4 | 0.585 |

The initial invocation of the RAND function above is seeded with the value 1. Subsequent invocations of the same function (in the recursive part of the statement) use the initial value to generate a reproducible set of pseudo-random numbers.

### 23.1.3. Using the GENERATE_UNIQUE function

With a bit of data manipulation, the GENERATE_UNIQUE function can be used (instead of the RAND function) to make suitably random test data. The are advantages and disadvantages to using both functions:

- The GENERATE_UNIQUE function makes data that is always unique. The RAND function only outputs one of 32,000 distinct values.
- The RAND function can make reproducible random data, while the GENERATE_UNIQUE function can not.

See the description of the GENERATE_UNIQUE function (see GENERATE_UNIQUE) for an example of how to use it to make random data.

### 23.1.4. Make Random Data - Different Ranges

There are several ways to mess around with the output from the RAND function: We can use simple arithmetic to alter the range of numbers generated (e.g. convert from 0 to 10 to 0 to 10,000). We can alter the format (e.g. from FLOAT to DECIMAL). Lastly, we can make fewer, or more, distinct random values (e.g. from 32K distinct values down to just 10). All of this is done below:

*Make differing ranges of random numbers*

```
WITH temp1 (s1, r1) AS
(VALUES (0
       , RAND(2))
   UNION ALL
 SELECT s1+1
      , RAND()
 FROM temp1
 WHERE s1+1 < 5)
SELECT SMALLINT(s1) AS seq#
     , SMALLINT(r1*10000) AS ran2
     , DECIMAL(r1,6,4) AS ran1
     , SMALLINT(r1*10) AS ran3
FROM temp1;
```

*ANSWER*

| SEQ# | RAN2 | RAN1 | RAN3 |
|---|---|---|---|
| 0 | 13 | 0.0013 | 0 |
| 1 | 8916 | 0.8916 | 8 |
| 2 | 7384 | 0.7384 | 7 |
| 3 | 5430 | 0.5430 | 5 |
| 4 | 8998 | 0.8998 | 8 |

### 23.1.5. Make Random Data - Varying Distribution

In the real world, there is a tendency for certain data values to show up much more frequently than others. Likewise, separate fields in a table usually have independent semi-random data distribution patterns. In the next statement we create three independently random fields. The first has the usual 32K distinct values evenly distributed in the range of zero to one. The second and third have

random numbers that are skewed towards the low end of the range, and have many more distinct values:

*Create RAND data with different distributions*

```
WITH temp1 (s1) AS
(VALUES (0)
   UNION ALL
 SELECT s1 + 1
 FROM temp1
 WHERE s1 + 1 < 5)
SELECT SMALLINT(s1)                            AS s#
     , INTEGER((RAND(1)) * 1E6)                AS ran1
     , INTEGER((RAND() * RAND()) * 1E6)        AS ran2
     , INTEGER((RAND() * RAND()* RAND()) * 1E6) AS ran3
FROM temp1;
```

*ANSWER*

| S# | RAN1 | RAN2 | RAN3 |
|---|---|---|---|
| 0 | 1251 | 365370 | 114753 |
| 1 | 350291 | 280730 | 88106 |
| 2 | 710501 | 149549 | 550422 |
| 3 | 147312 | 33311 | 2339 |
| 4 | 8911 | 556 | 73091 |

### 23.1.6. Make Random Data - Different Flavours

The RAND function generates random numbers. To get random character data one has to convert the RAND output into a character. There are several ways to do this. The first method shown below uses the CHR function to convert a number in the range: 65 to 90 into the ASCII equivalent: "A" to "Z". The second method uses the CHAR function to translate a number into the character equivalent.

*Converting RAND output from number to character*

```
WITH temp1 (s1, r1) AS
(VALUES (0
        , RAND(2))
   UNION ALL
 SELECT s1+1
      , RAND()
 FROM temp1
 WHERE s1+1 < 5)
SELECT SMALLINT(s1)              AS seq#
     , SMALLINT(r1*26+65)        AS ran2
     , CHR(SMALLINT(r1*26+65))   AS ran3
     , CHAR(SMALLINT(r1*26)+65) AS ran4
FROM temp1;
```

*ANSWER*

| SEQ# | RAN2 | RAN3 | RAN4 |
|---|---|---|---|
| 0 | 65 | A | 65 |
| 1 | 88 | X | 88 |
| 2 | 84 | T | 84 |
| 3 | 79 | O | 79 |
| 4 | 88 | X | 88 |

### 23.1.7. Make Test Table & Data

So far, all we have done in this chapter is use SQL to select sets of rows. Now we shall create a Production-like table for performance testing purposes. We will then insert 10,000 rows of suitably lifelike test data into the table. The DDL, with constraints and index definitions, follows. The important things to note are:

- The EMP# and the SOCSEC# must both be unique.
- The JOB_FTN, FST_NAME, and LST_NAME fields must all be non-blank.
- The SOCSEC# must have a special format.
- The DATE_BN must be greater than 1900.

Several other fields must be within certain numeric ranges.

*Production-like test table DDL*

```
CREATE TABLE personnel
( emp#     INTEGER       NOT NULL
, socsec#  CHAR(11)      NOT NULL
, job_ftn  CHAR(4)       NOT NULL
, dept     SMALLINT      NOT NULL
, salary   DECIMAL(7, 2) NOT NULL
, date_bn  DATE          NOT NULL WITH DEFAULT
, fst_name VARCHAR(20)
, lst_name VARCHAR(20)
, CONSTRAINT pex1 PRIMARY KEY (emp#)
, CONSTRAINT pe01 CHECK (emp# > 0)
, CONSTRAINT pe02 CHECK (LOCATE(' ', socsec#) = 0)
, CONSTRAINT pe03 CHECK (LOCATE('-', socsec#,1) = 4)
, CONSTRAINT pe04 CHECK (LOCATE('-', socsec#,5) = 7)
, CONSTRAINT pe05 CHECK (job_ftn <> '')
, CONSTRAINT pe06 CHECK (dept BETWEEN 1 AND 99)
, CONSTRAINT pe07 CHECK (salary BETWEEN 0 AND 99999)
, CONSTRAINT pe08 CHECK (fst_name <> '')
, CONSTRAINT pe09 CHECK (lst_name <> '')
, CONSTRAINT pe10 CHECK (date_bn >= '1900-01-01' ));

CREATE UNIQUE INDEX PEX2 ON PERSONNEL (SOCSEC#);
CREATE UNIQUE INDEX PEX3 ON PERSONNEL (DEPT, EMP#);
```

Now we shall populate the table. The SQL shall be described in detail latter. For the moment, note the four RAND fields. These contain, independently generated, random numbers which are used to populate the other data fields.

*Production-like test table INSERT*

```
INSERT INTO personnel
WITH temp1 (s1, r1, r2, r3, r4) AS
(VALUES (0
       , RAND(2)
       , RAND() + (RAND() /1E5)
       , RAND() * RAND()
       , RAND() * RAND() * RAND())
   UNION ALL
 SELECT s1 + 1
      , RAND()
      , RAND() + (RAND() / 1E5)
      , RAND() * RAND()
      , RAND() * RAND() * RAND()
 FROM temp1
 WHERE s1 < 10000)
SELECT 100000 + s1
     , SUBSTR(DIGITS(INT(r2*988+10)), 8) || '-' ||
       SUBSTR(DIGITS(INT(r1*88+10)),9) || '-' ||
       TRANSLATE(SUBSTR(DIGITS(s1), 7), '9873450126', '0123456789')
     , CASE
         WHEN INT(r4*9) > 7 THEN 'MGR'
         WHEN INT(r4*9) > 5 THEN 'SUPR'
         WHEN INT(r4*9) > 3 THEN 'PGMR'
         WHEN INT(R4*9) > 1 THEN 'SEC'
         ELSE 'WKR'
       END
     , INT(r3*98+1)
     , DECIMAL(r4*99999, 7, 2)
     , DATE('1930-01-01') + INT(50-(r4*50)) YEARS + INT(r4*11) MONTHS + INT(r4*27)
DAYS
     , CHR(INT(r1*26+65))|| CHR(INT(r2*26+97))|| CHR(INT(r3*26+97)) ||
       CHR(INT(r4*26+97))|| CHR(INT(r3*10+97))|| CHR(INT(r3*11+97))
     , CHR(INT(r2*26+65))|| TRANSLATE(CHAR(INT(r2*1E7)), 'aaeeiibmty', '0123456789')
FROM temp1;
```

Some sample data follows:

| EMP# | SOCSEC# | JOB_ | DEPT | SALARY | DATE_BN | F_NME | L_NME |
|---|---|---|---|---|---|---|---|
| 100000 | 484-10-9999 | WKR | 47 | 13.63 | 1979-01-01 | Ammaef | Mimytmbi |
| 100001 | 449-38-9998 | SEC | 53 | 35758.87 | 1962-04-10 | Ilojff | Liiiemea |
| 100002 | 979-90-9997 | WKR | 1 | 8155.23 | 1975-01-03 | Xzacaa | Zytaebma |
| 100003 | 580-50-9993 | WKR | 31 | 16643.50 | 1971-02-05 | Lpiedd | Pimmeeat |

| EMP# | SOCSEC# | JOB_ | DEPT | SALARY | DATE_BN | F_NME | L_NME |
|---|---|---|---|---|---|---|---|
| 100004 | 264-87-9994 | WKR | 21 | 962.87 | 1979-01-01 | Wgfacc | Geimteei |
| 100005 | 661-84-9995 | WKR | 19 | 4648.38 | 1977-01-02 | Wrebbc | Rbiybeet |
| 100006 | 554-53-9990 | WKR | 8 | 375.42 | 1979-01-01 | Mobaaa | Oiiaiaia |
| 100007 | 482-23-9991 | SEC | 36 | 23170.09 | 1968-03-07 | Emjgdd | Mimtmamb |
| 100008 | 536-41-9992 | WKR | 6 | 10514.11 | 1974-02-03 | Jnbcaa | Nieebayt |

In order to illustrate some of the tricks that one can use when creating such data, each field above was calculated using a different schema:

- The EMP# is a simple ascending number.
- The SOCSEC# field presented three problems: It had to be unique, it had to be random with respect to the current employee number, and it is a character field with special layout constraints (see the DDL on Production-like test table DDL).
- To make it random, the first five digits were defined using two of the temporary random number fields. To try and ensure that it was unique, the last four digits contain part of the employee number with some digit-flipping done to hide things. Also, the first random number used is the one with lots of unique values. The special formatting that this field required is addressed by making everything in pieces and then concatenating.
- The JOB FUNCTION is determined using the fourth (highly skewed) random number. This ensures that we get many more workers than managers.
- The DEPT is derived from another, somewhat skewed, random number with a range of values from one to ninety nine.
- The SALARY is derived using the same, highly skewed, random number that was used for the job function calculation. This ensures that theses two fields have related values.
- The BIRTH DATE is a random date value somewhere between 1930 and 1981.
- The FIRST NAME is derived using seven independent invocation of the CHR function, each of which is going to give a somewhat different result.
- The LAST NAME is (mostly) made by using the TRANSLATE function to convert a large random number into a corresponding character value. The output is skewed towards some of the vowels and the lower-range characters during the translation.

### 23.1.8. Time-Series Processing

The following table holds data for a typical time-series application. Observe is that each row has both a beginning and ending date, and that there are three cases where there is a gap between the end-date of one row and the begin-date of the next (with the same key).

*Sample Table DDL - Time Series*

```
CREATE TABLE time_series
( KYY CHAR(03) NOT NULL
, bgn_dt DATE NOT NULL
, end_dt DATE NOT NULL
, CONSTRAINT tsc1 CHECK (kyy <> '')
, CONSTRAINT tsc2 CHECK (bgn_dt <= end_dt));

COMMIT;

INSERT INTO TIME_series values
  ('AAA','1995-10-01','1995-10-04')
, ('AAA','1995-10-06','1995-10-06')
, ('AAA','1995-10-07','1995-10-07')
, ('AAA','1995-10-15','1995-10-19')
, ('BBB','1995-10-01','1995-10-01')
, ('BBB','1995-10-03','1995-10-03');
```

### Find Overlapping Rows

We want to find any cases where the begin-to-end date range of one row overlaps another with the same KYY value. The following diagram illustrates our task. The bold line at the top represents the begin and end date for a row. This row is overlapped (in time) by the six lower rows, but the nature of the overlap differs in each case.

*Overlapping Time-Series rows - Definition*

```
                            time
    >>>-------------------------------------------------------------------->>>
                  <<<------------------------------>>>
                  <<<------------------------------>>>
              <<<--->>>   <<<--->>>        <<<------------>>>
                                                  <<<---------->>>
          <<<-------------------------------------------------------->>>>
```

The general types of overlap are:

- The related row has identical date ranges.
- The related row begins before the start-date and ends after the same.
- The row begins and ends between the start and finish dates.

When writing SQL to check overlapping data ranges, make sure that all possible types of overlap (see diagram above) are tested. Some SQL statements work with some flavors of overlap, but not with others.

The relevant SQL follows. When reading it, think of the "A" table as being the bold line above and "B" table as being the four overlapping rows shown as single lines.

*Find overlapping rows in time-series*

```
SELECT kyy
     , bgn_dt
     , end_dt
FROM time_series a
WHERE EXISTS
    (SELECT *
     FROM time_series b
     WHERE a.kyy = b.kyy
     AND a.bgn_dt <> b.bgn_dt
     AND (a.bgn_dt BETWEEN b.bgn_dt AND b.end_dt
          OR
          b.bgn_dt BETWEEN a.bgn_dt AND a.end_dt)
    )
ORDER BY 1,2;
```

*ANSWER*:

| KEYCOL | BGN_DT | END_DT | BGN_DT | END_DT | DIFF |
|---|---|---|---|---|---|
| AAA | 1995-10-01 | 1995-10-04 | 1995-10-06 | 1995-10-06 | 2 |
| AAA | 1995-10-07 | 1995-10-07 | 1995-10-15 | 1995-10-19 | 8 |
| BBB | 1995-10-01 | 1995-10-01 | 1995-10-03 | 1995-10-03 | 2 |

The first predicate in the above sub-query joins the rows together by matching key value. The second predicate makes sure that one row does not match against itself. The final two predicates look for overlapping date ranges.

The above query relies on the sample table data being valid (as defined by the CHECK constraints in the DDL on Sample Table DDL - Time Series. This means that the END_DT is always greater than or equal to the BGN_DT, and each KYY, BGN_DT combination is unique.

**Find**

We want to find all those cases in the TIME_SERIES table when the ending of one row is not exactly one day less than the beginning of the next (if there is a next). The following query will answer this question. It consists of both a join and a sub-query. In the join (which is done first), we match each row with every other row that has the same key and a BGN_DT that is more than one day greater than the current END_DT. Next, the sub-query excludes from the result those join-rows where there is an intermediate third row.

*Find gap in Time-Series, SQL*

```
SELECT a.kyy
     , a.bgn_dt
     , a.end_dt
     , b.bgn_dt
     , b.end_dt
     , DAYS(b.bgn_dt) - DAYS(A.end_dt) as diff
FROM time_series a
   , time_series b
WHERE a.kyy = b.kyy
AND a.end_dt < b.bgn_dt - 1 DAY
AND NOT EXISTS
    (SELECT *
     FROM time_series z
     WHERE z.kyy = a.kyy
     AND z.kyy = b.kyy
     AND z.bgn_dt > a.bgn_dt
     AND z.bgn_dt < b.bgn_dt)
ORDER BY 1,2;
```

*TIME_SERIES*

| KYY | BGN_DT | END_DT |
|---|---|---|
| AAA | 1995-10-01 | 1995-10-04 |
| AAA | 1995-10-06 | 1995-10-06 |
| AAA | 1995-10-07 | 1995-10-07 |
| AAA | 1995-10-15 | 1995-10-19 |
| BBB | 1995-10-01 | 1995-10-01 |
| BBB | 1995-10-03 | 1995-10-03 |

*ANSWER*

| KEYCOL | BGN_DT | END_DT | BGN_DT | END_DT | DIFF |
|---|---|---|---|---|---|
| AAA | 1995-10-01 | 1995-10-04 | 1995-10-06 | 1995-10-06 | 2 |
| AAA | 1995-10-07 | 1995-10-07 | 1995-10-15 | 1995-10-19 | 8 |
| BBB | 1995-10-01 | 1995-10-01 | 1995-10-03 | 1995-10-03 | 2 |

If there are many rows per key value, the above SQL will be very inefficient. This is because the join (done first) does a form of Cartesian Product (by key value) making an internal result table that can be very large. The sub-query then cuts this temporary table down to size by removing results-rows that have other intermediate rows.

Instead of looking at those rows that encompass a gap in the data, we may want to look at the actual

gap itself. To this end, the following SQL differs from the prior in that the SELECT list has been modified to get the start, end, and duration, of each gap.

*Find gap in Time-Series*

```
SELECT a.kyy AS kyy
     , a.end_dt + 1 DAY AS bgn_gap
     , b.bgn_dt - 1 DAY AS end_gap
     , (DAYS(b.bgn_dt) - DAYS(a.end_dt) - 1) AS sz
FROM time_series a
   , time_series b
WHERE a.kyy = b.kyy
AND a.end_dt < b.bgn_dt - 1 DAY
AND NOT EXISTS
    (SELECT *
     FROM time_series z
     WHERE z.kyy = a.kyy
     AND z.kyy = b.kyy
     AND z.bgn_dt > a.bgn_dt
     AND z.bgn_dt < b.bgn_dt)
ORDER BY 1,2;
```

*TIME_SERIES*

| KYY | BGN_DT | END_DT |
|---|---|---|
| AAA | 1995-10-01 | 1995-10-04 |
| AAA | 1995-10-06 | 1995-10-06 |
| AAA | 1995-10-07 | 1995-10-07 |
| AAA | 1995-10-15 | 1995-10-19 |
| BBB | 1995-10-01 | 1995-10-01 |
| BBB | 1995-10-03 | 1995-10-03 |

*ANSWER*

| KYY | BGN_GAP | END_GAP | SZ |
|---|---|---|---|
| AAA | 1995-10-05 | 1995-10-05 | 1 |
| AAA | 1995-10-08 | 1995-10-14 | 7 |
| BBB | 1995-10-02 | 1995-10-02 | 1 |

### Show Each Day in Gap

Imagine that we wanted to see each individual day in a gap. The following statement does this by taking the result obtained above and passing it into a recursive SQL statement which then generates additional rows - one for each day in the gap after the first.

*Show each day in Time-Series gap*

```
WITH temp (kyy, gap_dt, gsize) AS
(SELECT a.kyy
      , a.end_dt + 1 DAY
      , (DAYS(b.bgn_dt) DAYS(a.end_dt) - 1)
 FROM time_series a
    , time_series b
 WHERE a.kyy = b.kyy
 AND a.end_dt < b.bgn_dt - 1 DAY
 AND NOT EXISTS
     (SELECT *
      FROM time_series z
      WHERE z.kyy = a.kyy
      AND z.kyy = b.kyy
      AND z.bgn_dt > a.bgn_dt
      AND z.bgn_dt < b.bgn_dt)
   UNION ALL
 SELECT kyy
      , gap_dt + 1 DAY
      , gsize - 1
 FROM temp WHERE gsize > 1)
SELECT *
FROM temp
ORDER BY 1, 2;
```

*ANSWER*

| KEYCOL | GAP_DT | GSIZE |
|---|---|---|
| AAA | 1995-10-05 | 1 |
| AAA | 1995-10-08 | 7 |
| AAA | 1995-10-09 | 6 |
| AAA | 1995-10-10 | 5 |
| AAA | 1995-10-11 | 4 |
| AAA | 1995-10-12 | 3 |
| AAA | 1995-10-13 | 2 |
| AAA | 1995-10-14 | 1 |
| BBB | 1995-10-02 | 1 |

## 23.2. Other Fun Things

### 23.2.1. Randomly Sample Data

One can use the TABLESAMPLE schema to randomly sample rows for subsequent analysis.

### Notes

- The table-name must refer to a real table. This can include a declared global temporary table, or a materialized query table. It cannot be a nested table expression.
- The sampling is an addition to any predicates specified in the where clause. Under the covers, sampling occurs before any other query processing, such as applying predicates or doing a join.
- The BERNOULLI option checks each row individually.
- The SYSTEM option lets Db2 find the most efficient way to sample the data. This may mean that all rows on each page that qualifies are included. For small tables, this method often results in a misleading percentage of rows selected.
- The "percent" number must be equal to or less than 100, and greater than zero. It determines what percentage of the rows processed are returns.
- The REPEATABLE option and number is used if one wants to get the same result every time the query is run (assuming no data changes). Without this option, each run will be both random and different.

### Examples

Sample 5% of the rows in the staff table. Get the same result each time:

*Sample rows in STAFF table*

```
SELECT *
FROM staff TABLESAMPLE BERNOULLI(5) REPEATABLE(1234)
ORDER BY id;
```

Sample 18% of the rows in the employee table and 25% of the rows in the employee-activity table, then join the two tables together. Because each table is sampled independently, the fraction of rows that join will be much less either sampling rate:

*Sample rows in two tables*

```
SELECT *
FROM employee ee TABLESAMPLE BERNOULLI(18)
   , emp_act  ea TABLESAMPLE BERNOULLI(25)
WHERE ee.empno = ea.empno
ORDER BY ee.empno;
```

Sample a declared global temporary table, and also apply other predicates:

*Sample Views used in Join Examples*

```
DECLARE GLOBAL TEMPORARY TABLE session.nyc_staff
LIKE staff;

SELECT *
FROM session.nyc_staff TABLESAMPLE SYSTEM(34.55)
WHERE id < 100
AND salary > 100
ORDER BY id;
```

## 23.3. Convert Character to Numeric

The DOUBLE, DECIMAL, INTEGER, SMALLINT, and BIGINT functions call all be used to convert a character field into its numeric equivalent:

*Convert Character to Numeric - SQL*

```
WITH temp1 (c1) AS
(VALUES '123 ', ' 345 ', ' 567')
SELECT c1
     , DOUBLE(c1)    AS dbl
     , DECIMAL(c1,3) AS dec
     , SMALLINT(c1)  AS sml
     , INTEGER(c1)   AS int
FROM temp1;
```

*ANSWER (numbers shortened)*

| C1 | DBL | DEC | SML | INT |
|---|---|---|---|---|
| 123 | +1.2300E+2 | 123. | 123 | 123 |
| 345 | +3.4500E+2 | 345. | 345 | 345 |
| 567 | +5.6700E+2 | 567. | 567 | 567 |

Not all numeric functions support all character representations of a number. The following table illustrates what's allowed and what's not: .Acceptable conversion values

| INPUT STRING | COMPATIBLE FUNCTIONS |
|---|---|
| " 1234" | DOUBLE, DECIMAL, INTEGER, SMALLINT, BIGINT |
| " 12.4" | DOUBLE, DECIMAL |
| " 12E4" | DOUBLE |

### 23.3.1. Checking the Input

There are several ways to check that the input character string is a valid representation of a number - before doing the conversion. One simple solution involves converting all digits to blank, then removing the blanks. If the result is not a zero length string, then the input must have had a character other than a digit:

*Checking for non-digits*

```
WITH temp1 (c1) AS
(VALUES ' 123', '456 ', ' 1 2', ' 33%', NULL)
SELECT c1
     , TRANSLATE(c1, '          ', '1234567890') AS c2
     , LENGTH(LTRIM(TRANSLATE(c1,'          ', '1234567890'))) AS c3
FROM temp1;
```

*ANSWER*

```
 C1   C2  C3
---- ---- --
 123       0
456        0
 1 2       0
 33%    %  1
-    -     -
```

One can also write a user-defined scalar function to check for non-numeric input, which is what is done below. This function returns "Y" if the following is true:

- The input is not null.
- There are no non-numeric characters in the input.
- The only blanks in the input are to the left of the digits.
- There is only one "+" or "-" sign, and it is next to the left-side blanks, if any.
- There is at least one digit in the input.

Now for the code:

This example uses an "!" as the stmt delimiter

*Check Numeric function*

```
--#SET DELIMITER !

CREATE FUNCTION isnumeric(instr VARCHAR(40))
RETURNS CHAR(1)
BEGIN ATOMIC
  DECLARE is_number CHAR(1) DEFAULT 'Y';
```

```
DECLARE bgn_blank CHAR(1) DEFAULT 'Y';
DECLARE found_num CHAR(1) DEFAULT 'N';
DECLARE found_pos CHAR(1) DEFAULT 'N';
DECLARE found_neg CHAR(1) DEFAULT 'N';
DECLARE found_dot CHAR(1) DEFAULT 'N';
DECLARE ctr SMALLINT DEFAULT 1;
IF instr IS NULL THEN
  RETURN NULL;
END IF;
wloop:
WHILE ctr <= LENGTH(instr) AND is_number = 'Y'
DO
  ---------------------------
  --- ERROR CHECKS        ---
  ---------------------------
  IF SUBSTR(instr, ctr, 1) NOT IN (' ', '.', '+', '-', '0', '1', '2'
                                  ,'3', '4', '5', '6', '7', '8', '9') THEN
    SET is_number = 'N';
    ITERATE wloop;
  END IF;
  IF SUBSTR(instr, ctr, 1) = ' ' AND bgn_blank = 'N' THEN
    SET is_number = 'N';
    ITERATE wloop;
  END IF;
  IF SUBSTR(instr, ctr, 1) = '.' AND found_dot = 'Y' THEN
    SET is_number = 'N';
    ITERATE wloop;
  END IF;
  IF SUBSTR(instr,ctr,1) = '+' AND (found_neg = 'Y' OR bgn_blank = 'N') THEN
    SET is_number = 'N';
    ITERATE wloop;
  END IF;
  IF SUBSTR(instr,ctr,1) = '-' AND (found_neg = 'Y' OR bgn_blank = 'N') THEN
    SET is_number = 'N';
    ITERATE wloop;
  END IF;
  ---------------------------
  --- MAINTAIN FLAGS & CTR ---
  ---------------------------
  IF SUBSTR(instr,ctr,1) IN ('0', '1', '2', '3', '4'
                            ,'5', '6', '7', '8', '9') THEN
    SET found_num = 'Y';
  END IF;
  IF SUBSTR(instr,ctr,1) = '.' THEN
    SET found_dot = 'Y';
  END IF;
  IF SUBSTR(instr,ctr,1) = '+' THEN
    SET found_pos = 'Y';
  END IF;
  IF SUBSTR(instr,ctr,1) = '-' THEN
    SET found_neg = 'Y';
```

```
    END IF;
    IF SUBSTR(instr,ctr,1) <> ' ' THEN
      SET bgn_blank = 'N';
    END IF;
    SET ctr = ctr + 1;
  END WHILE wloop;
  IF found_num = 'N' THEN
    SET is_number = 'N';
  END IF;
  RETURN is_number;
END!

WITH TEMP1 (C1) AS
(VALUES '    123'
      , '+123.45'
      , '456    '
      , ' 10 2  '
      , '   -.23'
      , '++12356'
      , '.012349'
      , '    33%'
      , '       '
      , NULL)
SELECT C1            AS C1
     , isnumeric(C1) AS C2
     , CASE
         WHEN isnumeric(C1) = 'Y' THEN DECIMAL(C1, 10, 6)
         ELSE NULL
       END           AS C3
FROM TEMP1!
```

*ANSWER*

| C1 | C2 | C3 |
|---|---|---|
| 123 | Y | 123.00000 |
| +123.45 | Y | 123.45000 |
| 456 | N | - |
| 10 2 | N | - |
| .23 | Y | -0.23000 |
| ++12356 | N | - |
| .012349 | Y | 0.01234 |
| 33% | N | - |
| | N | - |
| - | - | - |

See Check Data Value Type for a much simpler function that is similar to the above.

## 23.4. Convert Number to Character

The CHAR and DIGITS functions can be used to convert a Db2 numeric field to a character representation of the same, but as the following example demonstrates, both functions return problematic output:

*CHAR and DIGITS function usage*

```
SELECT d_sal
     , CHAR(d_sal) AS d_chr
     , DIGITS(d_sal) AS d_dgt
     , i_sal
     , CHAR(i_sal) AS i_chr
     , DIGITS(i_sal) AS i_dgt
FROM (SELECT DEC(salary - 11000, 6, 2) AS d_sal
           , SMALLINT(salary - 11000) AS i_sal
      FROM staff
      WHERE salary > 10000
      AND salary < 12200) AS xxx
ORDER BY d_sal;
```

*ANSWER*

| D_SAL | D_CHR | D_DGT | I_SAL | I_CHR | I_DGT |
|---|---|---|---|---|---|
| 494.10 | -0494.10 | 049410 | -494 | -494 | 00494 |
| -12.00 | -0012.00 | 001200 | -12 | -12 | 00012 |
| 508.60 | 0508.60 | 050860 | 508 | 508 | 00508 |
| 1009.75 | 1009.75 | 100975 | 1009 | 1009 | 01009 |

The DIGITS function discards both the sign indicator and the decimal point, while the CHAR function output is (annoyingly) left-justified, and (for decimal data) has leading zeros. We can do better.

Below are three user-defined functions that convert integer data from numeric to character, displaying the output right-justified, and with a sign indicator if negative. There is one function for each flavor of integer that is supported in Db2:

*User-defined functions - convert integer to character*

```
CREATE FUNCTION char_right(inval SMALLINT)
RETURNS CHAR(06)
RETURN RIGHT(CHAR('',06) CONCAT RTRIM(CHAR(inval)),06);

CREATE FUNCTION char_right(inval INTEGER)
RETURNS CHAR(11)
RETURN RIGHT(CHAR('',11) CONCAT RTRIM(CHAR(inval)),11);

CREATE FUNCTION char_right(inval BIGINT)
RETURNS CHAR(20)
RETURN RIGHT(CHAR('',20) CONCAT RTRIM(CHAR(inval)),20);
```

Each of the above functions works the same way (working from right to left):

- First, convert the input number to character using the CHAR function.
- Next, use the RTRIM function to remove the right-most blanks.
- Then, concatenate a set number of blanks to the left of the value. The number of blanks appended depends upon the input type, which is why there are three separate functions.
- Finally, use the RIGHT function to get the right-most "n" characters, where "n" is the maximum number of digits (plus the sign indicator) supported by the input type.

The next example uses the first of the above functions:

*Convert SMALLINT to CHAR*

```
SELECT i_sal
     , char_right(i_sal) AS i_chr
FROM (SELECT SMALLINT(salary - 11000) AS i_sal
      FROM staff
      WHERE salary > 10000
      AND salary < 12200) AS xxx
ORDER BY i_sal;
```

*ANSWER*

| I_SAL | I_CHR |
|---|---|
| 494 | -494 |
| -12 | -12 |
| 508 | 508 |
| 1009 | 1009 |

# 23.5. Decimal Input

Creating a similar function to handle decimal input is a little more tricky. One problem is that the CHAR function adds leading zeros to decimal data, which we don't want. A more serious problem is that there are many sizes and scales of decimal data, but we can only create one function (with a given name) for a particular input data type. Decimal values can range in both length and scale from 1 to 31 digits. This makes it impossible to define a single function to convert any possible decimal value to character with possibly running out of digits, or losing some precision.

The fact that one can only have one user-defined function, with a given name, per Db2 data type, presents a problem for all variable-length data types - notably character, varchar, and decimal. For character and varchar data, one can address the problem, to some extent, by using maximum length input and output fields. But decimal data has both a scale and a length, so there is no way to make an all-purpose decimal function.

Despite the above, below is a function that converts decimal data to character. It compromises by assuming an input of type decimal(22,2), which should handle most monetary values:

*User-defined function - convert decimal to character*

```
CREATE FUNCTION char_right(inval DECIMAL(20,2))
RETURNS CHAR(22)
RETURN RIGHT(CHAR('', 19)                                    CONCAT
             REPLACE(SUBSTR(CHAR(inval * 1), 1, 1), '0', '') CONCAT
             STRIP(CHAR(ABS(BIGINT(inval))))                 CONCAT
             '.'                                             CONCAT
             SUBSTR(DIGITS(inval), 19, 2), 22);
```

The function works as follows:

- The input value is converted to CHAR and the first byte obtained. This will be a minus sign if the number is negative, else blank.
- The non-fractional part of the number is converted to BIGINT then to CHAR.
- A period (dot) is included.
- The fractional digits (converted to character using the DIGITS function) are appended to the back of the output.
- All of the above is concatenation together, along with some leading blanks. Finally, the 22 right-most characters are returned.

Below is the function in action:

*Convert DECIMAL to CHAR*

```
WITH temp1 (num, tst) AS
(VALUES (1, DEC(0.01, 20, 2))
   UNION ALL
 SELECT num + 1
      , tst * -3.21
 FROM temp1
 WHERE num < 8)
SELECT num
     , tst
     , char_right(tst) AS tchar
FROM temp1;
```

*ANSWER*

| NUM | TST | TCHAR |
|---|---|---|
| 1 | 0.01 | 0.01 |
| 2 | -0.03 | -0.03 |
| 3 | 0.09 | 0.09 |
| 4 | -0.28 | -0.28 |
| 5 | 0.89 | 0.89 |
| 6 | -2.85 | -2.85 |
| 7 | 9.14 | 9.14 |
| 8 | -29.33 | -29.33 |

Floating point data can be processed using the above function, as long as it is first converted to decimal using the standard DECIMAL function.

### 23.5.1. Adding Commas

The next function converts decimal input to character, with embedded comas. It first coverts the value to character - as per the above function. It then steps though the output string, three bytes at a time, from right to left, checking to see if the next-left character is a number. If it is, it insert a comma, else it adds a blank byte to the front of the string:

*User-defined function - convert decimal to character - with commas*

```
CREATE FUNCTION comma_right(inval DECIMAL(20, 2))
RETURNS CHAR(27)
LANGUAGE SQL
DETERMINISTIC
NO EXTERNAL ACTION
BEGIN ATOMIC
  DECLARE i INTEGER DEFAULT 17;
  DECLARE abs_inval BIGINT;
  DECLARE out_value CHAR(27);
  SET abs_inval = ABS(BIGINT(inval));
  SET out_value = RIGHT(CHAR('', 19)             CONCAT
                  RTRIM(CHAR(BIGINT(inval))),19) CONCAT
                  '.'                            CONCAT
                  SUBSTR(DIGITS(inval),19,2);
  WHILE i > 2 DO
    IF SUBSTR(out_value, i-1, 1) BETWEEN '0' AND '9' THEN
      SET out_value = SUBSTR(out_value,1,i-1) CONCAT
                      ','                     CONCAT
                      SUBSTR(out_value,i);
    ELSE
      SET out_value = ' ' CONCAT out_value;
    END IF;
    SET i = i - 3;
  END WHILE;
  RETURN out_value;
END
```

Below is an example of the above function in use:

*Convert DECIMAL to CHAR with commas*

```
WITH temp1 (num) AS
(VALUES (DEC(+1,20,2))
      , (DEC(-1,20,2))
   UNION ALL
 SELECT num * 987654.12
 FROM temp1
 WHERE ABS(num) < 1E10)
, temp2 (num) AS
(SELECT num - 1
 FROM temp1)
SELECT num              AS input
     , comma_right(num) AS output
FROM temp2
ORDER BY num;
```

*ANSWER*

| INPUT | OUTPUT |
|---|---|
| -975460660753.97 | -975,460,660,753.97 |
| -987655.12 | -987,655.12 |
| -2.00 | -2.00 |
| 0.00 | 0.00 |
| 987653.12 | 987,653.12 |
| 975460660751.97 | 975,460,660,751.97 |

### 23.5.2. Convert Timestamp to Numeric

There is absolutely no sane reason why anyone would want to convert a date, time, or timestamp value directly to a number. The only correct way to manipulate such data is to use the provided date/time functions. But having said that, here is how one does it:

*Convert Timestamp to number*

```
WITH tab1(ts1) AS
(VALUES CAST('1998-11-22-03.44.55.123456' AS TIMESTAMP))
SELECT ts1                          --==> 1998-11-22-03.44.55.123456
     , HEX(ts1)                     --==> 19981122034455123456
     , DEC(HEX(ts1), 20)            --==> 19981122034455123456.
     , FLOAT(DEC(HEX(ts1), 20))     --==> 1.99811220344551e+019
     , REAL (DEC(HEX(ts1), 20))     --==> 1.998112e+019
FROM tab1;
```

### 23.5.3. Selective Column Output

There is no way in static SQL to vary the number of columns returned by a select statement. In order to change the number of columns you have to write a new SQL statement and then rebind. But one can use CASE logic to control whether or not a column returns any data. Imagine that you are forced to use static SQL. Furthermore, imagine that you do not always want to retrieve the data from all columns, and that you also do not want to transmit data over the network that you do not need. For character columns, we can address this problem by retrieving the data only if it is wanted, and otherwise returning to a zero-length string. To illustrate, here is an ordinary SQL statement:

*Sample query with no column control*

```
SELECT empno
     , firstnme
     , lastname
     , job
FROM employee
WHERE empno < '000100'
ORDER BY empno;
```

Here is the same SQL statement with each character column being checked against a hostvariable. If the host-variable is 1, the data is returned, otherwise a zero-length string:

*Sample query with column control*

```
SELECT empno
     , CASE :host-var-1
         WHEN 1 THEN firstnme
         ELSE ''
       END AS firstnme
     , CASE :host-var-2
         WHEN 1 THEN lastname
         ELSE ''
       END AS lastname
     , CASE :host-var-3
         WHEN 1 THEN VARCHAR(job)
         ELSE ''
       END AS job
FROM employee
WHERE empno < '000100'
ORDER BY empno;
```

### 23.5.4. Making Charts Using SQL

Imagine that one had a string of numeric values that one wants to display as a line-bar chart. With a little coding, this is easy to do in SQL:

*Make chart using SQL*

```
SELECT id
     , salary
     , INT(salary / 1500) AS len
     , REPEAT('*', INT(salary / 1500)) AS salary_chart
FROM staff
WHERE id > 120
AND   id < 190
ORDER BY id;
```

*ANSWER*

| ID | SALARY | LEN | SALARY_CHART |
|---|---|---|---|
| 130 | 10505.90 | 7 | * |
| 140 | 21150.00 | 14 | ** |
| 150 | 19456.50 | 12 | ** |
| 160 | 22959.20 | 15 | * |
| 170 | 12258.50 | 8 | ** |

| ID | SALARY | LEN | SALARY_CHART |
|---|---|---|---|
| 180 | 12009.75 | 8 | ** |

To create the above graph we first converted the column of interest to an integer field of a manageable length, and then used this value to repeat a single "*" character a set number of times. One problem with the above query is that we won't know how long the chart will be until we run the statement. This may cause problems if we guess wrongly and we are tight for space. The next query addresses this issue by creating a chart of known length. It does it by dividing the row value by the maximum value for the selected rows (all divided by 20). The result is used to repeat the "*" character "n" times:

*Make chart of fixed length*

```
SELECT dept
     , id
     , salary
     , VARCHAR(REPEAT('*', INT(salary / (MAX(salary) OVER() / 20))), 20) AS chart
FROM staff
WHERE dept <= 15
AND id >= 100
ORDER BY 1,2;
```

ANSWER

| DEPT | ID | SALARY | CHART |
|---|---|---|---|
| 10 | 160 | 82959.20 | ** |
| 10 | 210 | 90010.00 | |
| 10 | 240 | 79260.25 | * |
| 10 | 260 | 81234.00 | ** |
| 15 | 110 | 42508.20 | *** |
| 15 | 170 | 42258.50 | *** |

The above code can be enhanced to have two charts in the same column. To illustrate, the next query expresses the salary as a chart, but separately by department. This can be useful to do when the two departments have very different values and one wants to analyze the data in each department independently:

*Make two fixed length charts in the same column*

```
SELECT dept
     , id
     , salary
     , VARCHAR(REPEAT('*',
                      INT(salary / (MAX(salary)
                                    OVER(PARTITION BY dept) / 20)
                          )
                      ), 20) AS chart
FROM staff
WHERE dept <= 15
AND id >= 100
ORDER BY 1,2;
```

*ANSWER*

| DEPT | ID | SALARY | CHART |
|---|---|---|---|
| 10 | 160 | 82959.20 | ** |
| 10 | 210 | 90010.00 | |
| 10 | 240 | 79260.25 | * |
| 10 | 260 | 81234.00 | ** |
| 15 | 110 | 42508.20 | |
| 15 | 170 | 42258.50 | *** |

### 23.5.5. Multiple Counts in One Pass

Suppose we have a STATS table that has a SEX field with just two values, 'F' (for female) and 'M' (for male). To get a count of the rows by sex we can write the following:

*Use GROUP BY to get counts*

```
SELECT sex
     , COUNT(*) AS num
FROM stats
GROUP BY sex
ORDER BY sex;
```

*ANSWER*

| SEX | NUM |
|---|---|
| F | 595 |
| M | 405 |

Imagine now that we wanted to get a count of the different sexes on the same line of output. One,

not very efficient, way to get this answer is shown below. It involves scanning the data table twice (once for males, and once for females) then joining the result.

*Use Common Table Expression to get counts*

```
WITH f (f) AS
(SELECT COUNT(*) FROM stats WHERE sex = 'F')
, m (m) AS
(SELECT COUNT(*) FROM stats WHERE sex = 'M')
SELECT f, m
FROM f
   , m;
```

It would be more efficient if we answered the question with a single scan of the data table. This we can do using a CASE statement and a SUM function:

*Use CASE and SUM to get counts*

```
SELECT SUM(CASE sex WHEN 'F' THEN 1 ELSE 0 END) AS female
     , SUM(CASE sex WHEN 'M' THEN 1 ELSE 0 END) AS male
FROM stats;
```

We can now go one step further and also count something else as we pass down the data. In the following example we get the count of all the rows at the same time as we get the individual sex counts.

*Use CASE and SUM to get counts*

```
SELECT COUNT(*)                                 AS total
     , SUM(CASE sex WHEN 'F' THEN 1 ELSE 0 END) AS female
     , SUM(CASE sex WHEN 'M' THEN 1 ELSE 0 END) AS male
FROM stats;
```

### 23.5.6. Find Missing Rows in Series / Count all Values

One often has a sequence of values (e.g. invoice numbers) from which one needs both found and not-found rows. This cannot be done using a simple SELECT statement because some of rows being selected may not actually exist. For example, the following query lists the number of staff that have worked for the firm for "n" years, but it misses those years during which no staff joined:

*Count staff joined per year*

```
SELECT years
     , COUNT(*) AS #staff
FROM staff
WHERE UCASE(name) LIKE '%E%'
AND years <= 5
GROUP BY years;
```

*ANSWER*

| YEARS | #STAFF |
|---|---|
| 1 | 1 |
| 4 | 2 |
| 5 | 3 |

The simplest way to address this problem is to create a complete set of target values, then do an outer join to the data table. This is what the following example does:

*Count staff joined per year, all years*

```
WITH list_years (year#) AS
(VALUES (0), (1), (2), (3), (4), (5))
SELECT year#              AS years
     , COALESCE(#stff, 0) AS #staff
FROM list_years
LEFT OUTER JOIN
    (SELECT years
          , COUNT(*) AS #stff
     FROM staff
     WHERE UCASE(name) LIKE '%E%'
     AND years <= 5
     GROUP BY years) AS xxx
ON year# = years
ORDER BY 1;
```

*ANSWER*

| YEARS | #STAFF |
|---|---|
| 0 | 0 |
| 1 | 1 |
| 2 | 0 |
| 3 | 0 |
| 4 | 2 |
| 5 | 3 |

The use of the VALUES syntax to create the set of target rows, as shown above, gets to be tedious if the number of values to be made is large. To address this issue, the following example uses recursion to make the set of target values:

*Count staff joined per year, all years*

```
WITH list_years (year#) AS
(VALUES SMALLINT(0)
   UNION ALL
 SELECT year# + 1
 FROM list_years
 WHERE year# < 5)
SELECT year#              AS years
     , COALESCE(#stff, 0) AS #staff
FROM list_years
LEFT OUTER JOIN
    (SELECT years
          , COUNT(*) AS #stff
     FROM staff
     WHERE UCASE(name) LIKE '%E%'
     AND years <= 5
     GROUP BY years) AS xxx
ON year# = years
ORDER BY 1;
```

*ANSWER*

| YEARS | #STAFF |
|---|---|
| 0 | 0 |
| 1 | 1 |
| 2 | 0 |
| 3 | 0 |
| 4 | 2 |
| 5 | 3 |

If one turns the final outer join into a (negative) sub-query, one can use the same general logic to list those years when no staff joined:

*List years when no staff joined*

```
WITH list_years (year#) AS
(VALUES SMALLINT(0)
   UNION ALL
 SELECT year# + 1
 FROM list_years
 WHERE year# < 5)
SELECT year#
FROM list_years y
WHERE NOT EXISTS
    (SELECT *
     FROM staff s
     WHERE UCASE(s.name) LIKE '%E%'
     AND s.years = y.year#)
ORDER BY 1;
```

*ANSWER*

| YEAR# |
|---|
| 0 |
| 2 |
| 3 |

### 23.5.7. Multiple Counts from the Same Row

Imagine that we want to select from the EMPLOYEE table the following counts presented in a tabular list with one line per item. In each case, if nothing matches we want to get a zero:

- Those with a salary greater than $20,000
- Those whose first name begins 'ABC%'
- Those who are male.
- Employees per department.
- A count of all rows.

Note that a given row in the EMPLOYEE table may match more than one of the above criteria. If this were not the case, a simple nested table expression could be used. Instead we will do the following:

*Multiple counts in one pass*

```
WITH category (cat, subcat, dept) AS
(VALUES ('1ST', 'ROWS IN TABLE ', '')
      , ('2ND', 'SALARY > $20K ', '')
      , ('3RD', 'NAME LIKE ABC%', '')
      , ('4TH', 'NUMBER MALES ', '')
   UNION
 SELECT '5TH'
      , deptname
      , deptno
 FROM department)
SELECT xxx.cat        AS "category"
     , xxx.subcat     AS "subcategory/dept"
     , SUM(xxx.found) AS "#rows"
FROM (SELECT cat.cat
           , cat.subcat
           , CASE
               WHEN emp.empno IS NULL THEN 0
               ELSE 1
             END AS found
      FROM category cat
      LEFT OUTER JOIN employee emp
      ON cat.subcat = 'ROWS IN TABLE'
      OR (cat.subcat = 'NUMBER MALES'
          AND
          emp.sex = 'M')
      OR (cat.subcat = 'SALARY > $20K'
          AND
          emp.salary > 20000)
      OR (cat.subcat = 'NAME LIKE ABC%'
          AND
          emp.firstnme LIKE 'ABC%')
      OR (cat.dept <> ''
          AND
          cat.dept = emp.workdept)
         ) AS xxx
GROUP BY xxx.cat
       , xxx.subcat
ORDER BY 1,2;
```

In the above query, a temporary table is defined and then populated with all of the summation types. This table is then joined (using a left outer join) to the EMPLOYEE table. Any matches (i.e. where EMPNO is not null) are given a FOUND value of 1. The output of the join is then feed into a GROUP BY to get the required counts.

| CATEGORY | SUBCATEGORY/DEPT | #ROWS |
|---|---|---|
| 1ST | ROWS IN TABLE | 32 |
| 2ND | SALARY > $20K | 25 |

| CATEGORY | SUBCATEGORY/DEPT | #ROWS |
|---|---|---|
| 3RD | NAME LIKE ABC% | 0 |
| 4TH | NUMBER MALES | 19 |
| 5TH | ADMINISTRATION SYSTEMS | 6 |
| 5TH | DEVELOPMENT CENTER | 0 |
| 5TH | INFORMATION CENTER | 3 |
| 5TH | MANUFACTURING SYSTEMS | 9 |
| 5TH | OPERATIONS | 5 |
| 5TH | PLANNING | 1 |
| 5TH | SOFTWARE SUPPORT | 4 |
| 5TH | SPIFFY COMPUTER SERVICE DIV. | 3 |
| 5TH | SUPPORT SERVICES | 1 |

## 23.6. Normalize Denormalized Data

Imagine that one has a string of text that one wants to break up into individual words. As long as the word delimiter is fairly basic (e.g. a blank space), one can use recursive SQL to do this task. One recursively divides the text into two parts (working from left to right). The first part is the word found, and the second part is the remainder of the text:

*Break text into words - SQL*

```
WITH temp1 (id, data) AS
(VALUES (01, 'SOME TEXT TO PARSE.')
      , (02, 'MORE SAMPLE TEXT.')
      , (03, 'ONE-WORD.')
      , (04,''))
, temp2 (id, word#, word, data_left) AS
(SELECT id
      , SMALLINT(1)
      , SUBSTR(data, 1, CASE LOCATE(' ', data)
                          WHEN 0 THEN LENGTH(data)
                          ELSE LOCATE(' ', data)
                          END
              )
      , LTRIM(SUBSTR(data, CASE LOCATE(' ', data)
                             WHEN 0 THEN LENGTH(data) + 1
                             ELSE LOCATE(' ', data)
                            END
                    )
             )
 FROM temp1
 WHERE data <> ''
   UNION ALL
 SELECT id
      , word# + 1
      , SUBSTR(data_left, 1, CASE LOCATE(' ', data_left)
                               WHEN 0 THEN LENGTH(data_left)
                               ELSE LOCATE(' ', data_left)
                              END
              )
      , LTRIM(SUBSTR(data_left, CASE LOCATE(' ', data_left)
                                  WHEN 0 THEN LENGTH(data_left) + 1
                                  ELSE LOCATE(' ', data_left)
                                 END
                    )
             )
 FROM temp2
 WHERE data_left <> '')
SELECT *
FROM temp2
ORDER BY 1,2;
```

The SUBSTR function is used above to extract both the next word in the string, and the remainder of the text. If there is a blank byte in the string, the SUBSTR stops (or begins, when getting the remainder) at it. If not, it goes to (or begins at) the end of the string. CASE logic is used to decide what to do. .Break text into words

| ID | WORD# | WORD | DATA_LEFT |
|---|---|---|---|
| 1 | 1 | SOME | TEXT TO PARSE. |
| 1 | 2 | TEXT | TO PARSE. |
| 1 | 3 | TO | PARSE. |
| 1 | 4 | PARSE. | |
| 2 | 1 | MORE | SAMPLE TEXT. |
| 2 | 2 | SAMPLE | TEXT. |
| 2 | 3 | TEXT. | |
| 3 | 1 | ONE-WORD. | |

## 23.7. Denormalize Normalized Data

The SUM function can be used to accumulate numeric values. To accumulate character values (i.e. to string the individual values from multiple lines into a single long value) is a little harder, but it can also be done. The following example uses the XMLAGG column function to aggregate multiple values into one. The processing goes as follows:

- The XMLTEXT scalar function converts each character value into XML. A space is put at the end of the each name, so there is a gap before the next.
- The XMLAGG column function aggregates the individual XML values in name order.
- The XMLSERIALIZE scalar function converts the aggregated XML value into a CLOB.
- The SUBSTR scalar function converts the CLOB to a CHAR.

Now for the code:

*Denormalize Normalized Data*

```
SELECT dept
     , SMALLINT(COUNT(*)) AS #w
     , MAX(name) AS max_name
     , SUBSTR(
         XMLSERIALIZE(
           XMLAGG(
             XMLTEXT(name || ' ')
           ORDER BY name) AS CLOB(1M))
         , 1, 50) AS all_names
FROM staff
GROUP BY dept
ORDER BY dept;
```

Here is the answer:

| DEPT | W# | MAX_NAME | ALL_NAMES |
|---|---|---|---|
| 10 | 4 | Molinare | Daniels Jones Lu Molinare |
| 15 | 4 | Rothman | Hanes Kermisch Ngan Rothman |
| 20 | 4 | Sneider | James Pernal Sanders Sneider |
| 38 | 5 | Quigley | Abrahams Marenghi Naughton O'Brien Quigley |
| 42 | 4 | Yamaguchi | Koonitz Plotz Scoutten Yamaguchi |
| 51 | 5 | Williams | Fraye Lundquist Smith Wheeler Williams |
| 66 | 5 | Wilson | Burke Gonzales Graham Lea Wilson |
| 84 | 4 | Quill | Davis Edwards Gafney Quill |

The next example uses recursion to do exactly the same thing. It begins by getting the minimum name in each department. It then recursively gets the next to lowest name, then the next, and so on. As the query progresses, it maintains a count of names added, stores the current name in the temporary NAME field, and appends the same to the end of the ALL_NAMES field. Once all of the names have been processed, the final SELECT eliminates from the answer-set all rows, except the last for each department:

*Denormalize Normalized Data*

```
WITH temp1 (dept,w#,name,all_names) AS
(SELECT dept
      , SMALLINT(1)
      , MIN(name)
      , VARCHAR(MIN(name), 50)
 FROM staff a
 GROUP BY dept
   UNION ALL
 SELECT a.dept
      , SMALLINT(b.w#+1)
      , a.name
      , b.all_names || ' ' || a.name
 FROM staff a
    , temp1 b
 WHERE a.dept = b.dept
 AND a.name > b.name
 AND a.name =
      (SELECT MIN(c.name)
       FROM staff c
       WHERE c.dept = b.dept
       AND c.name > b.name)
 )
SELECT dept
     , w#
     , name AS max_name
     , all_names
FROM temp1 d
WHERE w# = (SELECT MAX(w#)
            FROM temp1 e
            WHERE d.dept = e.dept)
ORDER BY dept;
```

If there are no suitable indexes, the above query may be horribly inefficient. If this is the case, one can create a user-defined function to string together the names in a department:

This example uses an "!" as the stmt delimiter

*Creating a function to denormalize names*

```
CREATE FUNCTION list_names(indept SMALLINT)
RETURNS VARCHAR(50)
BEGIN ATOMIC
  DECLARE outstr VARCHAR(50) DEFAULT '';
  FOR list_names AS
    SELECT name
    FROM staff
    WHERE dept = indept
    ORDER BY name
  DO
    SET outstr = outstr || name || ' ';
  END FOR;
  SET outstr = rtrim(outstr);
  RETURN outstr;
END!

SELECT dept          AS DEPT
     , SMALLINT(cnt) AS W#
     , mxx AS MAX_NAME
     , list_names(dept) AS ALL_NAMES
FROM (SELECT dept
           , COUNT(*) as cnt
           , MAX(name) AS mxx
      FROM staff
      GROUP BY dept) as ddd
ORDER BY dept!
```

Even the above might have unsatisfactory performance - if there is no index on department. If adding an index to the STAFF table is not an option, it might be faster to insert all of the rows into a declared temporary table, and then add an index to that.

# 23.8. Transpose Numeric Data

In this section we will turn rows of numeric data into columns. This cannot be done directly in SQL because the language does not support queries where the output columns are unknown at query start. We will get around this limitation by sending the transposed output to a suitably long VARCHAR field. Imagine that we want to group the data in the STAFF sample table by DEPT and JOB to get the SUM salary for each instance, but not in the usual sense with one output row per DEPT and JOB value. Instead, we want to generate one row per DEPT, with a set of "columns" (in a VARCHAR field) that hold the SUM salary values for each JOB in the department. We will also put column titles on the first line of output. To make the following query simpler, three simple scalar functions will be used to convert data from one type to another:

- Convert decimal data to character - similar to the one on User-defined functions - convert integer to character.
- Convert smallint data to character - same as the one on User-defined functions - convert integer

to character.

- Right justify and add leading blanks to character data.

Now for the functions:

*Data Transformation Functions*

```
CREATE FUNCTION num_to_char(inval SMALLINT)
RETURNS CHAR(06)
RETURN RIGHT(CHAR('',06) CONCAT RTRIM(CHAR(inval)), 06);

CREATE FUNCTION num_to_char(inval DECIMAL(9, 2))
RETURNS CHAR(10)
RETURN RIGHT(CHAR('', 7)               CONCAT
       RTRIM(CHAR(BIGINT(inval))), 7)  CONCAT
       '.'                             CONCAT
       SUBSTR(DIGITS(inval), 8, 2);

CREATE FUNCTION right_justify(inval CHAR(5))
RETURNS CHAR(10)
RETURN RIGHT(CHAR('', 10) || RTRIM(inval), 10);
```

The query consists of lots of little steps that are best explained by describing each temporary table built:

- **DATA_INPUT**: This table holds the set of matching rows in the STAFF table, grouped by DEPT and JOB as per a typical query (see Transpose Numeric Data for the contents). This is the only time that we touch the original STAFF table. All subsequent queries directly or indirectly reference this table.
- **JOBS_LIST**: The list of distinct jobs in all matching rows. Each job is assigned two rownumbers, one ascending, and one descending.
- **DEPT_LIST**: The list of distinct departments in all matching rows.
- **DEPT_JOB_LIST**: The list of all matching department/job combinations. We need this table because not all departments have all jobs.
- **DATA_ALL_JOBS**: The DEPT_JOB_LIST table joined to the original DATA_INPUT table using a left outer join, so we now have one row with a sum-salary value for every JOB and DEPT instance.
- **DATA_TRANSFORM**: Recursively go through the DATA_ALL_JOBS table (for each department), adding the a character representation of the current sum-salary value to the back of a VARCHAR column.
- **DATA_LAST_ROW**: For each department, get the row with the highest ascending JOB# value. This row has the concatenated string of sum-salary values.

At this point we are done, except that we don't have any column headings in our output. The rest of the query gets these.

- **JOBS_TRANSFORM**: Recursively go through the list of distinct jobs, building a VARCHAR string of JOB names. The job names are right justified - to match the sumsalary values, and have the

same output length.

- **JOBS_LAST_ROW**: Get the one row with the lowest descending job number. This row has the complete string of concatenated job names.
- **DATA_AND_JOBS**: Use a UNION ALL to vertically combine the JOBS_LAST_ROW and DATA_LAST_ROW tables. The result is a new table with both column titles and sum-salary values.

Finally, we select the list of column names and sum-salary values. The output is ordered so that the column names are on the first line fetched.

Now for the query:

*Transform numeric data*

```
WITH data_input AS
(SELECT dept
      , job
      , SUM(salary) AS sum_sal
 FROM staff
 WHERE id < 200
 AND name <> 'Sue'
 AND salary > 10000
 GROUP BY dept
        , job)
, jobs_list AS
(SELECT job
      , ROW_NUMBER() OVER(ORDER BY job ASC) AS job#A
      , ROW_NUMBER() OVER(ORDER BY job DESC) AS job#D
 FROM data_input
 GROUP BY job)
, dept_list AS
(SELECT dept
 FROM data_input
 GROUP BY dept)
, dept_jobs_list AS
(SELECT dpt.dept
      , job.job
      , job.job#A
      , job.job#D
 FROM jobs_list job
 FULL OUTER JOIN dept_list dpt
 ON 1 = 1)
, data_all_jobs AS
(SELECT djb.dept
      , djb.job
      , djb.job#A
      , djb.job#D
      , COALESCE(dat.sum_sal, 0) AS sum_sal
 FROM dept_jobs_list djb
 LEFT OUTER JOIN data_input dat
```

```
 ON djb.dept = dat.dept
 AND djb.job = dat.job)
, data_transform (dept, job#A, job#D, outvalue) AS
(SELECT dept
      , job#A
      , job#D
      , VARCHAR(num_to_char(sum_sal), 250)
 FROM data_all_jobs
 WHERE job#A = 1
   UNION ALL
 SELECT dat.dept
      , dat.job#A
      , dat.job#D
      , trn.outvalue || ',' || num_to_char(dat.sum_sal)
 FROM data_transform trn
    , data_all_jobs dat
 WHERE trn.dept = dat.dept
 AND trn.job#A = dat.job#A - 1)
, data_last_row AS
(SELECT dept
      , num_to_char(dept) AS dept_char
     , outvalue
 FROM data_transform
 WHERE job#D = 1)
, jobs_transform (job#A, job#D, outvalue) AS
(SELECT job#A
      , job#D
      , VARCHAR(right_justify(job), 250)
 FROM jobs_list
 WHERE job#A = 1
   UNION ALL
 SELECT job.job#A
      , job.job#D
      , trn.outvalue || ',' || right_justify(job.job)
 FROM jobs_transform trn
    , jobs_list job
 WHERE trn.job#A = job.job#A - 1)
, jobs_last_row AS
(SELECT 0      AS dept
     , ' DEPT' AS dept_char
     , outvalue
 FROM jobs_transform
 WHERE job#D = 1)
, data_and_jobs AS
(SELECT ept
      , ept_char
      , outvalue
 FROM jobs_last_row
   UNION ALL
 SELECT dept
      , dept_char
```

```
      , outvalue
 FROM data_last_row)
SELECT dept_char || ',' || outvalue AS output
FROM data_and_jobs
ORDER BY dept;
```

For comparison, below are the contents of the first temporary table, and the final output:

*Contents of first temporary table and final output*

**DATA_INPUT**

| DEPT | JOB | SUM_SAL |
|---|---|---|
| 10 | Mgr | 22959.20 |
| 15 | Clerk | 24766.70 |
| 15 | Mgr | 20659.80 |
| 15 | Sales | 16502.83 |
| 20 | Clerk | 27757.35 |
| 20 | Mgr | 18357.50 |
| 20 | Sales | 78171.25 |
| 38 | Clerk | 24964.50 |
| 38 | Mgr | 77506.75 |
| 38 | Sales | 34814.30 |
| 42 | Clerk | 10505.90 |
| 42 | Mgr | 18352.80 |
| 42 | Sales | 18001.75 |
| 51 | Mgr | 21150.00 |
| 51 | Sales | 19456.50 |

**OUTPUT**

| DEPT | Clerk | Mgr | Sales |
|---|---|---|---|
| 10 | 0.00 | 22959.20 | 0.00 |
| 15 | 24766.70 | 20659.80 | 16502.83 |
| 20 | 27757.35 | 18357.50 | 78171.25 |
| 38 | 24964.50 | 77506.75 | 34814.30 |
| 42 | 10505.90 | 18352.80 | 18001.75 |
| 51 | 0.00 | 21150.00 | 19456.50 |

# 23.9. Reversing Field Contents

Db2 lacks a simple function for reversing the contents of a data field. Fortunately, we can create a function to do it ourselves.

## 23.9.1. Input vs. Output

Before we do any data reversing, we have to define what the reversed output should look like relative to a given input value. For example, if we have a four-digit numeric field, the reverse of the number 123 could be 321, or it could be 3210. The latter value implies that the input has a leading zero. It also assumes that we really are working with a four digit field. Likewise, the reverse of the number 123.45 might be 54.321, or 543.21. Another interesting problem involves reversing negative numbers. If the value "-123" is a string, then the reverse is probably "321-". If it is a number, then the desired reverse is more likely to be "-321". Trailing blanks in character strings are a similar problem. Obviously, the reverse of "ABC" is "CBA", but what is the reverse of "ABC "? There is no general technical answer to any of these questions. The correct answer depends upon the business needs of the application. Below is a user defined function that can reverse the contents of a character field:

This example uses an "!" as the stmt delimiter

*Reversing character field*

```
--#SET DELIMITER !

CREATE FUNCTION reverse(instr VARCHAR(50))
RETURNS VARCHAR(50)
BEGIN ATOMIC
  DECLARE outstr VARCHAR(50) DEFAULT '';
  DECLARE curbyte SMALLINT DEFAULT 0;
  SET curbyte = LENGTH(RTRIM(instr));
  WHILE curbyte >= 1 DO
    SET outstr = outstr || SUBSTR(instr,curbyte, 1);
    SET curbyte = curbyte - 1;
  END WHILE;
  RETURN outstr;
END!

SELECT id AS ID
     , name AS NAME1
     , reverse(name) AS NAME2
FROM staff
WHERE id < 40
ORDER BY id!
```

*ANSWER*

| ID | NAME1 | NAME2 |
|---|---|---|
| 10 | Sanders | srednaS |
| 20 | Pernal | lanreP |
| 30 | Marenghi | ihgneraM |

The same function can be used to reverse numeric values, as long as they are positive:

*Reversing numeric field*

```
SELECT id                                AS ID
     , salary                            AS SALARY1
     , DEC(reverse(CHAR(salary)), 7, 4) AS SALARY2
FROM staff
WHERE id < 40
ORDER BY id;
```

*ANSWER*

| ID | SALARY1 | SALARY2 |
|---|---|---|
| 10 | 18357.50 | 5.7538 |
| 20 | 78171.25 | 52.1718 |
| 30 | 77506.75 | 57.6057 |

Simple CASE logic can be used to deal with negative values (i.e. to move the sign to the front of the string, before converting back to numeric), if they exist.

## 23.10. Fibonacci Series

A Fibonacci Series is a series of numbers where each value is the sum of the previous two. Regardless of the two initial (seed) values, if run for long enough, the division of any two adjacent numbers will give the value 0.618 or inversely 1.618. The following user defined function generates a Fibonacci series using three input values:

- First seed value.
- Second seed value.
- Number values to generate in series.

Observe that that the function code contains a check to stop series generation if there is not enough space in the output field for more numbers:

This example uses an "!" as the stmt delimiter

*Fibonacci Series function*

```
--#SET DELIMITER !

CREATE FUNCTION Fibonacci (inval1 INTEGER
                         , inval2 INTEGER
                         , loopno INTEGER)
RETURNS VARCHAR(500)
BEGIN ATOMIC
  DECLARE loopctr INTEGER DEFAULT 0;
  DECLARE tempval1 BIGINT;
  DECLARE tempval2 BIGINT;
  DECLARE tempval3 BIGINT;
  DECLARE outvalue VARCHAR(500);
  SET tempval1 = inval1;
  SET tempval2 = inval2;
  SET outvalue = RTRIM(LTRIM(CHAR(tempval1))) || ', ' ||
                 RTRIM(LTRIM(CHAR(tempval2)));
  calc:
  WHILE loopctr < loopno DO
    SET tempval3 = tempval1 + tempval2;
    SET tempval1 = tempval2;
    SET tempval2 = tempval3;
    SET outvalue = outvalue || ', ' || RTRIM(LTRIM(CHAR(tempval3)));
    SET loopctr = loopctr + 1;
    IF LENGTH(outvalue) > 480 THEN
      SET outvalue = outvalue || ' etc...';
      LEAVE calc;
    END IF;
  END WHILE;
  RETURN outvalue;
END!
```

The following query references the function:

*Fibonacci Series generation*

```
WITH temp1 (v1, v2, lp) AS
(VALUES (00, 01, 11)
      , (12, 61, 10)
      , (02, 05, 09)
      , (01, -1, 08))
SELECT t1.*
     , Fibonacci(v1, v2, lp) AS sequence
FROM temp1 t1;
```

*ANSWER*

| V1 | V2 | LP | SEQUENCE |
|---|---|---|---|
| 0 | 1 | 11 | 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144 |
| 12 | 61 | 10 | 12, 61, 73, 134, 207, 341, 548, 889, 1437, 2326, 3763, 6089 |
| 2 | 5 | 9 | 2, 5, 7, 12, 19, 31, 50, 81, 131, 212, 343 |
| 1 | -1 | 8 | 1, -1, 0, -1, -1, -2, -3, -5, -8, -13 |

The above example generates the complete series of values. If needed, the code could easily be simplified to simply return only the last value in the series. Likewise, a recursive join can be used to create a set of rows that are a Fibonacci series.

## 23.11. Business Day Calculation

The following function will calculate the number of business days (i.e. Monday to Friday) between to two dates:

This example uses an "!" as the stmt delimiter.

*Calculate number of business days between two dates*

```
CREATE FUNCTION business_days (lo_date DATE, hi_date DATE)
RETURNS INTEGER
BEGIN ATOMIC
  DECLARE bus_days INTEGER DEFAULT 0;
  DECLARE cur_date DATE;
  SET cur_date = lo_date;
  WHILE cur_date < hi_date DO
    IF DAYOFWEEK(cur_date) IN (2,3,4,5,6) THEN
      SET bus_days = bus_days + 1;
    END IF;
    SET cur_date = cur_date + 1 DAY;
  END WHILE;
  RETURN bus_days;
END!
```

Below is an example of the function in use:

*Use business-day function*

```
WITH temp1 (ld, hd) AS
(VALUES (DATE('2006-01-10'), DATE('2007-01-01'))
      , (DATE('2007-01-01'), DATE('2007-01-01'))
      , (DATE('2007-02-10'), DATE('2007-01-01')))
SELECT t1.*
     , DAYS(hd) - DAYS(ld)   AS diff
     , business_days(ld, hd) AS bdays
FROM temp1 t1;
```

*ANSWER*

| LD | HD | DIFF | BDAYS |
|---|---|---|---|
| 2006-01-10 | 2007-01-01 | 356 | 254 |
| 2007-01-01 | 2007-01-01 | 0 | 0 |
| 2007-02-10 | 2007-01-01 | -40 | 0 |

# 23.12. Query Runs for "n" Seconds

Imagine that one wanted some query to take exactly four seconds to run. The following query does just this - by looping (using recursion) until such time as the current system timestamp is four seconds greater than the system timestamp obtained at the beginning of the query:

*Run query for four seconds*

```
WITH temp1 (num,ts1,ts2) AS
(VALUES (INT(1)
       , TIMESTAMP(GENERATE_UNIQUE())
       , TIMESTAMP(GENERATE_UNIQUE()))
   UNION ALL
 SELECT num + 1
      , ts1
      , TIMESTAMP(GENERATE_UNIQUE())
 FROM temp1
 WHERE TIMESTAMPDIFF(2, CHAR(ts2 - ts1)) < 4)
SELECT MAX(num) AS #loops
     , MIN(ts2) AS bgn_timestamp
     , MAX(ts2) AS end_timestamp
FROM temp1;
```

*ANSWER*

| #LOOPS | BGN_TIMESTAMP | END_TIMESTAMP |
|---|---|---|
| 58327 | 2001-08-09-22.58.12.754579 | 2001-08-09-22.58.16.754634 |

Observe that the CURRENT TIMESTAMP special register is not used above. It is not appropriate for

this situation, because it always returns the same value for each invocation within a single query.

## 23.13. Function to Pause for "n" Seconds

We can take the above query and convert it into a user-defined function that will loop for "n" seconds, where "n" is the value passed to the function. However, there are several caveats:

- Looping in SQL is a "really stupid" way to hang around for a couple of seconds. A far better solution would be to call a stored procedure written in an external language that has a true pause command.
- The number of times that the function is invoked may differ, depending on the access path used to run the query.
- The recursive looping is going to result in the calling query getting a warning message.

Now for the code:

*Function that pauses for "n" seconds*

```
CREATE FUNCTION pause(inval INT)
RETURNS INTEGER
NOT DETERMINISTIC
EXTERNAL ACTION
RETURN
WITH ttt (num, strt, stop) AS
       (VALUES (1
              , TIMESTAMP(GENERATE_UNIQUE())
              , TIMESTAMP(GENERATE_UNIQUE()))
          UNION ALL
        SELECT num + 1
             , strt
             , TIMESTAMP(GENERATE_UNIQUE())
        FROM ttt
        WHERE TIMESTAMPDIFF(2, CHAR(stop - strt)) < inval)
SELECT MAX(num)
FROM ttt;
```

Below is a query that calls the above function:

*Query that uses pause function*

```
SELECT id
     , SUBSTR(CHAR(TIMESTAMP(GENERATE_UNIQUE())),18) AS ss_mmmmmm
     , pause(id / 10) AS #loops
     , SUBSTR(CHAR(TIMESTAMP(GENERATE_UNIQUE())),18) AS ss_mmmmmm
FROM staff
WHERE id < 31;
```

*ANSWER*

| ID | SS_MMMMMM | #LOOPS | SS_MMMMMM |
|---|---|---|---|
| 10 | 50.068593 | 76386 | 50.068587 |
| 20 | 52.068744 | 144089 | 52.068737 |
| 30 | 55.068930 | 206101 | 55.068923 |

# 23.14. Sort Character Field Contents

The following user-defined scalar function will sort the contents of a character field in either ascending or descending order. There are two input parameters:

- The input string: As written, the input can be up to 20 bytes long. To sort longer fields, change the input, output, and OUT-VAL (variable) lengths as desired.
- The sort order (i.e. 'A' or 'D').

The function uses a very simple, and not very efficient, bubble-sort. In other words, the input string is scanned from left to right, comparing two adjacent characters at a time. If they are not in sequence, they are swapped - and flag indicating this is set on. The scans are repeated until all of the characters in the string are in order:

*Define sort-char function*

```
--#SET DELIMITER !

CREATE FUNCTION sort_char(in_val VARCHAR(20), sort_dir VARCHAR(1))
RETURNS VARCHAR(20)
BEGIN ATOMIC
  DECLARE cur_pos SMALLINT;
  DECLARE do_sort CHAR(1);
  DECLARE out_val VARCHAR(20);
  IF UCASE(sort_dir) NOT IN ('A','D') THEN
    SIGNAL SQLSTATE '75001'
    SET MESSAGE_TEXT = 'Sort order not ''A'' or ''D''';
  END IF;
  SET out_val = in_val;
  SET do_sort = 'Y';
  WHILE do_sort = 'Y' DO
    SET do_sort = 'N';
    SET cur_pos = 1;
    WHILE cur_pos < length(in_val) DO
      IF (UCASE(sort_dir) = 'A'
          AND SUBSTR(out_val, cur_pos+1, 1) < SUBSTR(out_val, cur_pos, 1)
         ) OR
         (UCASE(sort_dir) = 'D'
          AND SUBSTR(out_val, cur_pos+1, 1) > SUBSTR(out_val, cur_pos, 1)) THEN
        SET do_sort = 'Y';
        SET out_val = CASE
        WHEN cur_pos = 1
          THEN ''
          ELSE SUBSTR(out_val, 1, cur_pos-1)
        END CONCAT SUBSTR(out_val, cur_pos+1, 1)
            CONCAT SUBSTR(out_val, cur_pos , 1)
            CONCAT CASE WHEN cur_pos = length(in_val) - 1
                     THEN ''
                     ELSE SUBSTR(out_val,cur_pos+2)
                   END;
      END IF;
      SET cur_pos = cur_pos + 1;
    END WHILE;
  END WHILE;
  RETURN out_val;
END!
```

Here is the function in action:

*Use sort-char function*

```
WITH word1 (w#, word_val) AS
(VALUES(1, '12345678')
     , (2, 'ABCDEFG')
     , (3, 'AaBbCc')
     , (4,'abccb')
     , (5,'''%#.')
     , (6,'bB')
     , (7,'a')
     , (8,''))
SELECT w#
     , word_val
     , sort_char(word_val, 'a') sa
     , sort_char(word_val, 'D') sd
FROM word1
ORDER BY w#;
```

*ANSWER*

| W# | WORD_VAL | SA | SD |
|---|---|---|---|
| 1 | 12345678 | 12345678 | 87654321 |
| 2 | ABCDEFG | ABCDEFG | GFEDCBA |
| 3 | AaBbCc | aAbBcC | CcBbAa |
| 4 | abccb | abbcc | ccbba |
| 5 | '%#. | .'#% | %#'. |
| 6 | bB | bB | Bb |
| 7 | a | a | a |
| 8 | | | |

# 23.15. Calculating the Median

The median is defined at that value in a series of values where half of the values are higher to it and the other half are lower. The median is a useful number to get when the data has a few very extreme values that skew the average. If there are an odd number of values in the list, then the median value is the one in the middle (e.g. if 7 values, the median value is #4). If there is an even number of matching values, there are two formulas that one can use:

- The most commonly used definition is that the median equals the sum of the two middle values, divided by two.
- A less often used definition is that the median is the smaller of the two middle values.

Db2 does not come with a function for calculating the median, but it can be obtained using the ROW_NUMBER function. This function is used to assign a row number to every matching row, and then one searches for the row with the middle row number.

### 23.15.1. Using Formula #1

Below is some sample code that gets the median SALARY, by JOB, for some set of rows in the STAFF table. Two JOB values are referenced - one with seven matching rows, and one with four. The query logic goes as follows:

- Get the matching set of rows from the STAFF table, and give each row a row-number, within each JOB value.
- Using the set of rows retrieved above, get the maximum row-number, per JOB value, then add 1.0, then divide by 2, then add or subtract 0.6. This will give one two values that encompass a single row-number, if an odd number of rows match, and two row-numbers, if an even number of rows match.
- Finally, join the one row per JOB obtained in step 2 above to the set of rows retrieved in step 1 - by common JOB value, and where the row-number is within the high/low range. The average salary of whatever is retrieved is the median.

Now for the code:

*Calculating the median*

```
WITH numbered_rows AS
(SELECT s.*
      , ROW_NUMBER() OVER(PARTITION BY job
                          ORDER BY salary, id) AS row#
 FROM staff s
 WHERE comm > 0
 AND name LIKE '%e%')
, median_row_num AS
(SELECT job
    , (MAX(row# + 1.0) / 2) - 0.5 AS med_lo
    , (MAX(row# + 1.0) / 2) + 0.5 AS med_hi
 FROM numbered_rows
 GROUP BY job)
SELECT nn.job
     , DEC(AVG(nn.salary), 7, 2) AS med_sal
FROM numbered_rows nn
   , median_row_num mr
WHERE nn.job = mr.job
AND nn.row# BETWEEN mr.med_lo AND mr.med_hi
GROUP BY nn.job
ORDER BY nn.job;
```

*ANSWER*

| JOB | MED_SAL |
|---|---|
| Clerk | 13030.50 |
| Sales | 17432.10 |

To get consistent results when using the ROW_NUMBER function, one must ensure that the ORDER BY column list encompasses the unique key of the table. Otherwise the row-number values will be assigned randomly - if there are multiple rows with the same value. In this particular case, the ID has been included in the ORDER BY list, to address duplicate SALARY values.

The next example is the essentially the same as the prior, but there is additional code that gets the average SALARY, and a count of the number of matching rows per JOB value. Observe that all this extra code went in the second step:

*Get median plus average*

```
WITH numbered_rows AS
(SELECT s.*
      , ROW_NUMBER() OVER(PARTITION BY job
                          ORDER BY salary, id) AS row#
 FROM staff s
 WHERE comm > 0
 AND name LIKE '%e%')
, median_row_num AS
(SELECT job
     , (MAX(row# + 1.0) / 2) - 0.5 AS med_lo
     , (MAX(row# + 1.0) / 2) + 0.5 AS med_hi
     , DEC(AVG(salary),7,2)        AS avg_sal
     , COUNT(*)                    AS #rows
 FROM numbered_rows
 GROUP BY job)
SELECT nn.job
     , DEC(AVG(nn.salary),7,2) AS med_sal
     , MAX(mr.avg_sal)         AS avg_sal
     , MAX(mr.#rows)           AS #r
FROM numbered_rows  nn
   , median_row_num mr
WHERE nn.job = mr.job
AND nn.row# BETWEEN mr.med_lo AND mr.med_hi
GROUP BY nn.job
ORDER BY nn.job;
```

*ANSWER*

| JOB | MED_SAL | AVG_SAL | #R |
|---|---|---|---|
| Clerk | 13030.50 | 12857.56 | 7 |
| Sales | 17432.10 | 17460.93 | 4 |

## 23.15.2. Using Formula #2

Once again, the following sample code gets the median SALARY, by JOB, for some set of rows in the STAFF table. Two JOB values are referenced - one with seven matching rows, the other with four. In

this case, when there is an even number of matching rows, the smaller of the two middle values is chosen. The logic goes as follows:

- Get the matching set of rows from the STAFF table, and give each row a row-number, within each JOB value.
- Using the set of rows retrieved above, get the maximum row-number per JOB, then add 1, then divide by 2. This will get the row-number for the row with the median value.
- Finally, join the one row per JOB obtained in step 2 above to the set of rows retrieved in step 1 - by common JOB and row-number value.

*Calculating the median*

```
WITH numbered_rows AS
(SELECT s.*
      , ROW_NUMBER() OVER(PARTITION BY job
                          ORDER BY salary, id) AS row#
 FROM staff s
 WHERE comm > 0
 AND name LIKE '%e%')
, median_row_num AS
(SELECT job
      , MAX(row# + 1) / 2 AS med_row#
 FROM numbered_rows GROUP BY job)
SELECT nn.job
     , nn.salary AS med_sal
FROM numbered_rows nn
   , median_row_num mr
WHERE nn.job = mr.job
AND nn.row# = mr.med_row#
ORDER BY nn.job;
```

*ANSWER*

| JOB | MED_SAL |
|---|---|
| Clerk | 13030.50 |
| Sales | 16858.20 |

The next query is the same as the prior, but it uses a sub-query, instead of creating and then joining to a second temporary table:

*Calculating the median*

```
WITH numbered_rows AS
(SELECT s.*
      , ROW_NUMBER() OVER(PARTITION BY job
                          ORDER BY salary, id) AS row#
 FROM staff s
 WHERE comm > 0
 AND name LIKE '%e%')
SELECT job
     , salary AS med_sal
FROM numbered_rows
WHERE (job, row#) IN
    (SELECT job
          , MAX(row# + 1) / 2
     FROM numbered_rows
     GROUP BY job)
ORDER BY job;
```

*ANSWER*

| JOB | MED_SAL |
|---|---|
| Clerk | 13030.50 |
| Sales | 16858.20 |

The next query lists every matching row in the STAFF table (per JOB), and on each line of output, shows the median salary:

*List matching rows and median*

```
WITH numbered_rows AS
(SELECT s.*
      , ROW_NUMBER() OVER(PARTITION BY job
                          ORDER BY salary, id) AS row#
 FROM staff s
 WHERE comm > 0
 AND name LIKE '%e%')
SELECT r1.*
    , (SELECT r2.salary
       FROM numbered_rows r2
       WHERE r2.job = r1.job
       AND r2.row# = (SELECT MAX(r3.row# + 1) / 2
                      FROM numbered_rows r3
                      WHERE r2.job = r3.job
                     )
      ) AS med_sal
FROM numbered_rows r1
ORDER BY job
       , salary;
```

# 23.16. Converting HEX Data to Number

The following function accepts as input a hexadecimal representation of an integer value, and returns a BIGINT number. It works for any integer type:

*Function to convert HEX value to integer*

```
CREATE FUNCTION hex_to_int(input_val VARCHAR(16))
RETURNS BIGINT
BEGIN ATOMIC
  DECLARE parse_val VARCHAR(16) DEFAULT '';
  DECLARE sign_val BIGINT DEFAULT 1;
  DECLARE out_val BIGINT DEFAULT 0;
  DECLARE cur_exp BIGINT DEFAULT 1;
  DECLARE input_len SMALLINT DEFAULT 0;
  DECLARE cur_byte SMALLINT DEFAULT 1;
  IF LENGTH(input_val) NOT IN (4,8,16) THEN
    SIGNAL SQLSTATE VALUE '70001'
    SET MESSAGE_TEXT = 'Length wrong';
  END IF;
  SET input_len = LENGTH(input_val);
  WHILE cur_byte <= input_len DO
    SET parse_val = parse_val                          ||
                    SUBSTR(input_val,cur_byte + 1,1) ||
                    SUBSTR(input_val,cur_byte + 0,1);
    SET cur_byte = cur_byte + 2;
  END WHILE;
  IF SUBSTR(parse_val,input_len,1) BETWEEN '8' AND 'F' THEN
```

```
    SET sign_val = -1;
    SET out_val = -1;
    SET parse_val = TRANSLATE(parse_val, '0123456789ABCDEF', 'FEDCBA9876543210');
  END IF;
  SET cur_byte = 1;
  WHILE cur_byte <= input_len DO
    SET out_val = out_val + (cur_exp *
                             sign_val *
                             CASE SUBSTR(parse_val, cur_byte, 1)
                               WHEN '0' THEN 00
                               WHEN '1' THEN 01
                               WHEN '2' THEN 02
                               WHEN '3' THEN 03
                               WHEN '4' THEN 04
                               WHEN '5' THEN 05
                               WHEN '6' THEN 06
                               WHEN '7' THEN 07
                               WHEN '8' THEN 08
                               WHEN '9' THEN 09
                               WHEN 'A' THEN 10
                               WHEN 'B' THEN 11
                               WHEN 'C' THEN 12
                               WHEN 'D' THEN 13
                               WHEN 'E' THEN 14
                               WHEN 'F' THEN 15
                             END);
    IF cur_byte < input_len THEN
      SET cur_exp = cur_exp * 16;
    END IF;
    SET cur_byte = cur_byte + 1;
  END WHILE;
  RETURN out_val;
END
```

### 23.16.1. Function Logic

The function does the following:

- Check that the input value is the correct length for an integer value. If not, flag an error.
- Transpose every second byte in the input value. This is done because the HEX representation of an integer does not show the data as it really is.
- Check the high-order bit of what is now the last byte. If it is a "1", the value is a negative number, so the processing will be slightly different.
- Starting with the first byte in the (transposed) input, covert each byte to an integer value using CASE logic. Multiply each digit obtained by the (next) power of sixteen.
- Return the final result.

### 23.16.2. Usage Examples

*Using trigger to convert data*

```
WITH temp1 (num) AS
(VALUES (SMALLINT(+0))
      , (SMALLINT(+1))
      , (SMALLINT(-1))
      , (SMALLINT(+32767))
      , (SMALLINT(-32768)))
SELECT num
     , HEX(num)             AS hex
     , hex_to_int(HEX(num)) AS h2i
FROM temp1;
```

*ANSWER*

| NUM | HEX | H2I |
|---|---|---|
| 0 | 0000 | 0 |
| 1 | 0100 | 1 |
| -1 | FFFF | -1 |
| 32767 | FF7F | 32767 |
| -32768 | 0080 | -32768 |

*Using trigger to convert data*

```
...
WITH temp1 (num) AS
(VALUES (INTEGER(0))
   UNION ALL
 SELECT (num + 1) * 7
 FROM temp1
 WHERE num < 1E6)
, temp2 (sgn) AS
(VALUES (+1)
      , (-13))
, temp3 (num) AS
(SELECT DISTINCT num * sgn
 FROM temp1
    , temp2)
SELECT num
     , HEX(num)             AS hex
     , hex_to_int(HEX(num)) AS h2i
FROM temp3
ORDER BY num;
....
```

*ANSWER*

| NUM | HEX | H2I |
|---|---|---|
| 87432800 | A0E1C9FA | -87432800 |
| -12490387 | 6D6941FF | -12490387 |
| -1784328 | F8C5E4FF | -1784328 |
| -254891 | 551CFCFF | -254891 |
| -36400 | D071FFFF | -36400 |
| -5187 | BDEBFFFF | -5187 |
| -728 | 28FDFFFF | -728 |
| -91 | A5FFFFFF | -91 |
| 0 | 00000000 | 0 |
| 7 | 07000000 | 7 |
| 56 | 38000000 | 56 |
| 399 | 8F010000 | 399 |
| 2800 | F00A0000 | 2800 |
| 19607 | 974C0000 | 19607 |
| 137256 | 28180200 | 137256 |
| 960799 | 1FA90E00 | 960799 |
| 6725600 | E09F6600 | 6725600 |

**Usage Notes**

- The above function won't work on the mainframe because the internal representation of an integer value is different (see below). The modifications required to make it work are minor.
- The above function won't work on the HEX representation of packed-decimal or floatingpoint data.
- One could have three different flavors of the above function - one for each type of integer value. The input value length would determine the output type.

# 23.17. Endianness

Most computers use one of two internal formats to store binary data. In big-endian, which is used on z/OS, the internal representation equals the HEX value. So the four-byte integer value 1,234,567,890 is stored as "49 96 02 D2". In little-endian, which is used on all Intel chips, the bytes are reversed, so the above value is stored internally as "D2 02 96 49". This is why the above function transposed every two-byte block before converting the HEX value to numeric.

# Chapter 24. Quirks in SQL

One might have noticed by now that not all SQL statements are easy to comprehend. Unfortunately, the situation is perhaps a little worse than you think. In this section we will discuss some SQL statements that are correct, but which act just a little funny.

## 24.1. Trouble with Timestamps

When does one timestamp not equal another with the same value? The answer is, when one value uses a 24 hour notation to represent midnight and the other does not. To illustrate, the following two timestamp values represent the same point in time, but not according to Db2:

*Timestamp comparison - Incorrect*

```
WITH temp1 (c1, t1, t2) AS
(VALUES('A', TIMESTAMP('1996-05-01-24.00.00.000000'), TIMESTAMP('1996-05-02-
00.00.00.000000')))
SELECT c1
FROM temp1
WHERE t1 = t2;
```

*ANSWER*: No rows

To make Db2 think that both timestamps are actually equal (which they are), all we have to do is fiddle around with them a bit:

*Timestamp comparison - Correct*

```
WITH temp1 (c1,t1,t2)
AS (VALUES('A', TIMESTAMP('1996-05-01-24.00.00.000000'), TIMESTAMP('1996-05-02-
00.00.00.000000')))
SELECT c1
FROM temp1
WHERE t1 + 0 MICROSECOND = t2 + 0 MICROSECOND;
```

*ANSWER*

| C1 |
| --- |
| A |

Be aware that, as with everything else in this section, what is shown above is not a bug. It is the way that it is because it makes perfect sense, even if it is not intuitive.

## 24.2. Using 24 Hour Notation

One might have to use the 24-hour notation, if one needs to record (in Db2) external actions that happen just before midnight - with the correct date value. To illustrate, imagine that we have the

following table, which records supermarket sales:

*Sample Table*

```
CREATE TABLE supermarket_sales
( sales_ts TIMESTAMP NOT NULL
, sales_val DECIMAL(8, 2) NOT NULL
, PRIMARY KEY(sales_ts));
```

In this application, anything that happens before midnight, no matter how close, is deemed to have happened on the specified day. So if a transaction comes in with a timestamp value that is a tiny fraction of a microsecond before midnight, we should record it thus:

*Insert row*

```
INSERT INTO supermarket_sales
VALUES ('2003-08-01-24.00.00.000000', 123.45);
```

Now, if we want to select all of the rows that are for a given day, we can write this:

*Select rows for given date*

```
SELECT *
FROM supermarket_sales
WHERE DATE(sales_ts) = '2003-08-01'
ORDER BY sales_ts;
```

Or this:

*Select rows for given date*

```
SELECT *
FROM supermarket_sales
WHERE sales_ts BETWEEN '2003-08-01-00.00.00' AND '2003-08-01-24.00.00'
ORDER BY sales_ts;
```

Db2 will never internally generate a timestamp value that uses the 24 hour notation. But it is provided so that you can use it yourself, if you need to.

## 24.3. No Rows Match

How many rows are returned by a query when no rows match the provided predicates? The answer is that sometimes you get none, and sometimes you get one:

*Query with no matching rows (1 of 8)*

```
SELECT creator
FROM sysibm.systables
WHERE creator = 'ZZZ';
```

*ANSWER*: no row

*Query with no matching rows (2 of 8)*

```
SELECT MAX(creator)
FROM sysibm.systables
WHERE creator = 'ZZZ';
```

*ANSWER*: Null

*Query with no matching rows (3 of 8)*

```
SELECT MAX(creator)
FROM sysibm.systables
WHERE creator = 'ZZZ'
HAVING MAX(creator) IS NOT NULL;
```

*ANSWER*: no row

*Query with no matching rows (4 of 8)*

```
SELECT MAX(creator)
FROM sysibm.systables
WHERE creator = 'ZZZ'
HAVING MAX(creator) = 'ZZZ';
```

*ANSWER*: no row

*Query with no matching rows (5 of 8)*

```
SELECT MAX(creator)
FROM sysibm.systables
WHERE creator = 'ZZZ'
GROUP BY creator;
```

*ANSWER*: no row

*Query with no matching rows (6 of 8)*

```
SELECT creator
FROM sysibm.systables
WHERE creator = 'ZZZ'
GROUP BY creator;
```

*ANSWER*: no row

*Query with no matching rows (7 of 8)*

```
SELECT COUNT(*)
FROM sysibm.systables
WHERE creator = 'ZZZ'
GROUP BY creator;
```

*ANSWER*: no row

*Query with no matching rows (8 of 8)*

```
SELECT COUNT(*)
FROM sysibm.systables
WHERE creator = 'ZZZ';
```

*ANSWER*: 0

There is a pattern to the above, and it goes thus:

- When there is no column function (e.g. MAX, COUNT) in the SELECT then, if there are no matching rows, no row is returned.
- If there is a column function in the SELECT, but nothing else, then the query will always return a row - with zero if the function is a COUNT, and null if it is something else.
- If there is a column function in the SELECT, and also a HAVING phrase in the query, a row will only be returned if the HAVING predicate is true.
- If there is a column function in the SELECT, and also a GROUP BY phrase in the query, a row will only be returned if there was one that matched.

Imagine that one wants to retrieve a list of names from the STAFF table, but when no names match, one wants to get a row/column with the phrase "NO NAMES", rather than zero rows. The next query does this by first generating a "not found" row using the SYSDUMMY1 table, and then left-outer-joining to the set of matching rows in the STAFF table. The COALESCE function will return the STAFF data, if there is any, else the not-found data:

*Always get a row, example 1 of 2*

```
SELECT COALESCE(name,noname)  AS nme
     , COALESCE(salary,nosal) AS sal
FROM (SELECT 'NO NAME' AS noname
           , 0 AS nosal
      FROM sysibm.sysdummy1) AS nnn
LEFT OUTER JOIN
    (SELECT *
     FROM staff
     WHERE id < 5) AS xxx
ON 1 = 1
ORDER BY name;
```

*ANSWER*

| NME | SAL |
|---|---|
| NO NAME | 0.00 |

The next query is logically the same as the prior, but it uses the WITH phrase to generate the "not found" row in the SQL statement:

*Always get a row, example 2 of 2*

```
WITH nnn (noname, nosal) AS
(VALUES ('NO NAME', 0))
SELECT COALESCE(name, noname) AS nme
     , COALESCE(salary,nosal) AS sal
FROM nnn
LEFT OUTER JOIN
    (SELECT *
     FROM staff
     WHERE id < 5) AS xxx
ON 1 = 1
ORDER BY NAME;
```

*ANSWER*

| NME | SAL |
|---|---|
| NO NAME | 0.00 |

## 24.4. Dumb Date Usage

Imagine that you have some character value that you convert to a Db2 date. The correct way to do it is given below:

*Convert value to Db2 date, right*

```
SELECT DATE('2001-09-22')
FROM sysibm.sysdummy1;
```

*ANSWER*: 2001-09-22

What happens if you accidentally leave out the quotes in the DATE function? The function still works, but the result is not correct:

*Convert value to Db2 date, wrong*

```
SELECT DATE(2001-09-22)
FROM sysibm.sysdummy1;
```

*ANSWER*: 0006-05-24

Why the 2,000 year difference in the above results? When the DATE function gets a character string as input, it assumes that it is valid character representation of a Db2 date, and converts it accordingly. By contrast, when the input is numeric, the function assumes that it represents the number of days minus one from the start of the current era (i.e. 0001-01-01). In the above query the input was 2001-09-22, which equals (2001-9)-22, which equals 1970 days.

## 24.5. RAND in Predicate

The following query was written with intentions of getting a single random row out of the matching set in the STAFF table. Unfortunately, it returned two rows:

*Get random rows - Incorrect*

```
SELECT id
     , name
FROM staff
WHERE id <= 100
AND id = (INT(RAND()* 10) * 10) + 10
ORDER BY id;
```

*ANSWER*

| ID | NAME |
|---|---|
| 30 | Marenghi |
| 60 | Quigley |

The above SQL returned more than one row because the RAND function was reevaluated for each matching row. Thus the RAND predicate was being dynamically altered as rows were being fetched. To illustrate what is going on above, consider the following query. The results of the RAND function are displayed in the output. Observe that there are multiple rows where the function output

(suitably massaged) matched the ID value. In theory, anywhere between zero and all rows could match:

*Get random rows - Explanation*

```
WITH temp AS
(SELECT id
      , name
      , (INT(RAND(0)* 10) * 10) + 10 AS ran
 FROM staff
 WHERE id <= 100)
SELECT t.*
     , CASE id
         WHEN ran THEN 'Y'
         ELSE ' '
       END AS eql
FROM temp t
ORDER BY id;
```

*ANSWER*

| ID | NAME | RAN | EQL |
|---|---|---|---|
| 10 | Sanders | 10 | Y |
| 20 | Pernal | 30 | |
| 30 | Marenghi | 70 | |
| 40 | O'Brien | 10 | |
| 50 | Hanes | 30 | |
| 60 | Quigley | 40 | |
| 70 | Rothman | 30 | |
| 80 | James | 100 | |
| 90 | Koonitz | 40 | |
| 100 | Plotz | 100 | Y |

To randomly select some fraction of the rows in a table efficiently and consistently, use the TABLESAMPLE feature. See Randomly Sample Data for more details.

## 24.6. Getting "n" Random Rows

There are several ways to always get exactly "n" random rows from a set of matching rows. In the following example, three rows are required:

*Get random rows - Non-distinct*

```
WITH staff_numbered AS
(SELECT s.*
      , ROW_NUMBER() OVER() AS row#
 FROM staff s
 WHERE id <= 100)
, count_rows AS
(SELECT MAX(row#) AS #rows
 FROM staff_numbered)
, random_values (RAN#) AS
(VALUES (RAND())
      , (RAND())
      , (RAND()))
, rows_t0_get AS
(SELECT INT(ran# * #rows) + 1 AS get_row
 FROM random_values
    , count_rows)
SELECT id
     , name
FROM staff_numbered
   , rows_t0_get
WHERE row# = get_row
ORDER BY id;
```

*ANSWER*

| ID | NAME |
|---|---|
| 10 | Sanders |
| 20 | Pernal |
| 90 | Koonitz |

The above query works as follows:

- First, the matching rows in the STAFF table are assigned a row number.
- Second, a count of the total number of matching rows is obtained.
- Third, a temporary table with three random values is generated.
- Fourth, the three random values are joined to the row-count value, resulting in three new row-number values (of type integer) within the correct range.
- Finally, the three row-number values are joined to the original temporary table.

There are some problems with the above query:

- If more than a small number of random rows are required, the random values cannot be defined using the VALUES phrase. Some recursive code can do the job.
- In the extremely unlikely event that the RAND function returns the value "one", no row will

match. CASE logic can be used to address this issue.

- Ignoring the problem just mentioned, the above query will always return three rows, but the rows may not be different rows. Depending on what the three RAND calls generate, the query may even return just one row - repeated three times.

In contrast to the above query, the following will always return three different random rows:

*Get random rows - Distinct*

```
SELECT id
     , name
FROM (SELECT s2.*
           , ROW_NUMBER() OVER(ORDER BY r1) AS r2
      FROM (SELECT s1.*
                 , RAND() AS r1
            FROM staff s1
            WHERE id <= 100) AS s2
     ) as s3
WHERE r2 <= 3
ORDER BY id;
```

*ANSWER*

| ID | NAME |
|---|---|
| 10 | Sanders |
| 40 | O'Brien |
| 60 | Quigley |

In this query, the matching rows are first numbered in random order, and then the three rows with the lowest row number are selected.

# 24.7. Summary of Issues

The lesson to be learnt here is that one must consider exactly how random one wants to be when one goes searching for a set of random rows:

- Does one want the number of rows returned to be also somewhat random?
- Does one want exactly "n" rows, but it is OK to get the same row twice?
- Does one want exactly "n" distinct (i.e. different) random rows?

# 24.8. Date/Time Manipulation

I once had a table that contained two fields - the timestamp when an event began, and the elapsed time of the event. To get the end-time of the event, I added the elapsed time to the begin-timestamp - as in the following SQL:

*Date/Time manipulation - wrong*

```
WITH temp1 (bgn_tstamp, elp_sec) AS
(VALUES (TIMESTAMP('2001-01-15-01.02.03.000000'), 1.234)
      , (TIMESTAMP('2001-01-15-01.02.03.123456'), 1.234))
SELECT bgn_tstamp
     , elp_sec
     , bgn_tstamp + elp_sec SECONDS AS end_tstamp
FROM temp1;
```

*ANSWER*

| BGN_TSTAMP | ELP_SEC | END_TSTAMP |
|---|---|---|
| 2001-01-15-01.02.03.000000 | 1.234 | 2001-01-15-01.02.04.000000 |
| 2001-01-15-01.02.03.123456 | 1.234 | 2001-01-15-01.02.04.123456 |

As you can see, my end-time is incorrect. In particular, the factional part of the elapsed time has not been used in the addition. I subsequently found out that Db2 never uses the fractional part of a number in date/time calculations. So to get the right answer I multiplied my elapsed time by one million and added microseconds:

*Date/Time manipulation - right*

```
WITH temp1 (bgn_tstamp, elp_sec) AS
(VALUES (TIMESTAMP('2001-01-15-01.02.03.000000'), 1.234)
      , (TIMESTAMP('2001-01-15-01.02.03.123456'), 1.234))
SELECT bgn_tstamp
     , elp_sec
     , bgn_tstamp + (elp_sec * 1E6) MICROSECONDS AS end_tstamp
FROM temp1;
```

*ANSWER*

| BGN_TSTAMP | ELP_SEC | END_TSTAMP |
|---|---|---|
| 2001-01-15-01.02.03.000000 | 1.234 | 2001-01-15-01.02.04.234000 |
| 2001-01-15-01.02.03.123456 | 1.234 | 2001-01-15-01.02.04.357456 |

Db2 doesn't use the fractional part of a number in date/time calculations because such a value often makes no sense. For example, 3.3 months or 2.2 years are meaningless values - given that neither a month nor a year has a fixed length.

### 24.8.1. The Solution

When one has a fractional date/time value (e.g. 5.1 days, 4.2 hours, or 3.1 seconds) that is for a period of fixed length that one wants to use in a date/time calculation, one has to convert the value into some whole number of a more precise time period. For example:

- 5.1 days times 86,400 returns the equivalent number of seconds.
- 6.2 seconds times 1,000,000 returns the equivalent number of microseconds.

## 24.9. Use of  on VARCHAR

Sometimes one value can be EQUAL to another, but is not LIKE the same. To illustrate, the following SQL refers to two fields of interest, one CHAR, and the other VARCHAR. Observe below that both rows in these two fields are seemingly equal:

*Use LIKE on CHAR field*

```
WITH temp1 (c0, c1, v1) AS
(VALUES('A', CHAR(' ', 1), VARCHAR(' ', 1))
     , ('B', CHAR(' ', 1), VARCHAR('' , 1)))
SELECT c0
FROM temp1
WHERE c1 = v1
AND c1 LIKE ' ';
```

*ANSWER*

| C0 |
| --- |
| A |
| B |

Look what happens when we change the final predicate from matching on C1 to V1. Now only one row matches our search criteria.

*Use LIKE on VARCHAR field*

```
WITH temp1 (c0, c1, v1) AS
(VALUES ('A',CHAR(' ', 1),VARCHAR(' ', 1))
      , ('B',CHAR(' ', 1),VARCHAR('' , 1)))
SELECT c0
FROM temp1
WHERE c1 = v1
AND v1 LIKE ' ';
```

*ANSWER*

| C0 |
| --- |
| A |

To explain, observe that one of the VARCHAR rows above has one blank byte, while the other has no data. When an EQUAL check is done on a VARCHAR field, the value is padded with blanks (if needed) before the match. This is why C1 equals C2 for both rows. However, the LIKE check does not pad VARCHAR fields with blanks. So the LIKE test in the second SQL statement only matched on

one row. The RTRIM function can be used to remove all trailing blanks and so get around this problem:

*Use RTRIM to remove trailing blanks*

```
WITH temp1 (c0,c1,v1) AS
(VALUES ('A',CHAR(' ',1),VARCHAR(' ',1))
      , ('B',CHAR(' ',1),VARCHAR('' ,1)))
SELECT c0
FROM temp1
WHERE c1 = v1
AND RTRIM(v1) LIKE '';
```

*ANSWER*

| C0 |
|---|
| A |
| B |

## 24.10. Comparing Weeks

One often wants to compare what happened in part of one year against the same period in another year. For example, one might compare January sales over a decade period. This may be a perfectly valid thing to do when comparing whole months, but it rarely makes sense when comparing weeks or individual days. The problem with comparing weeks from one year to the next is that the same week (as defined by Db2) rarely encompasses the same set of days. The following query illustrates this point by showing the set of days that make up week 33 over a ten-year period. Observe that some years have almost no overlap with the next:

*Comparing week 33 over 10 years*

```
WITH temp1 (yymmdd) AS
(VALUES DATE('2000-01-01')
   UNION ALL
 SELECT yymmdd + 1 DAY
 FROM temp1
 WHERE yymmdd < '2010-12-31')
SELECT yy                      AS year
     , CHAR(MIN(yymmdd), ISO) AS min_dt
     , CHAR(MAX(yymmdd), ISO) AS max_dt
FROM (SELECT yymmdd
           , YEAR(yymmdd) yy
           , WEEK(yymmdd) wk
      FROM temp1
      WHERE WEEK(yymmdd) = 33) AS xxx
GROUP BY yy
       , wk;
```

*ANSWER*

| YEAR | MIN_DT | MAX_DT |
|---|---|---|
| 2000 | 2000-08-06 | 2000-08-12 |
| 2001 | 2001-08-12 | 2001-08-18 |
| 2002 | 2002-08-11 | 2002-08-17 |
| 2003 | 2003-08-10 | 2003-08-16 |
| 2004 | 2004-08-08 | 2004-08-14 |
| 2005 | 2005-08-07 | 2005-08-13 |
| 2006 | 2006-08-13 | 2006-08-19 |
| 2007 | 2007-08-12 | 2007-08-18 |
| 2008 | 2008-08-10 | 2008-08-16 |
| 2009 | 2009-08-09 | 2009-08-15 |
| 2010 | 2010-08-08 | 2010-08-14 |

## 24.11. Db2 Truncates, not Rounds

When converting from one numeric type to another where there is a loss of precision, Db2 always truncates not rounds. For this reason, the S1 result below is not equal to the S2 result:

*Db2 data truncation*

```
SELECT SUM(INTEGER(salary)) AS s1
     , INTEGER(SUM(salary)) AS s2
FROM staff;
```

*ANSWER*

| S1 | S2 |
|---|---|
| 583633 | 583647 |

If one must do scalar conversions before the column function, use the ROUND function to improve the accuracy of the result:

*Db2 data rounding*

```
SELECT SUM(INTEGER(ROUND(salary, -1))) AS s1
     , INTEGER(SUM(salary))            AS s2
FROM staff;
```

*ANSWER*

| S1 | S2 |
|---|---|
| 583640 | 583647 |

## 24.12. CASE Checks in Wrong Sequence

The case WHEN checks are processed in the order that they are found. The first one that matches is the one used. To illustrate, the following statement will always return the value 'FEM' in the SXX field:

*Case WHEN Processing - Incorrect*

```
SELECT lastname
     , sex
     , CASE
         WHEN sex >= 'F' THEN 'FEM'
         WHEN sex >= 'M' THEN 'MAL'
       END AS sxx
FROM employee
WHERE lastname LIKE 'J%'
ORDER BY 1;
```

*ANSWER*

| LASTNAME | SX | SXX |
|---|---|---|
| JEFFERSON | M | FEM |
| JOHNSON | F | FEM |
| JONES | M | FEM |

By contrast, in the next statement, the SXX value will reflect the related SEX value:

*Case WHEN Processing - Correct*

```
SELECT lastname
     , sex
     , CASE
         WHEN sex >= 'M' THEN 'MAL'
         WHEN sex >= 'F' THEN 'FEM'
       END AS sxx
FROM employee
WHERE lastname LIKE 'J%'
ORDER BY 1;
```

*ANSWER*

| LASTNAME | SX | SXX |
|---|---|---|
| JEFFERSON | M | MAL |

| LASTNAME | SX | SXX |
|---|---|---|
| JOHNSON | F | FEM |
| JONES | M | MAL |

## 24.13. Division and Average

The following statement gets two results, which is correct?

*Division and Average*

```
SELECT AVG(salary) / AVG(comm) AS a1
     , AVG(salary / comm)      AS a2
FROM staff;
```

*ANSWER*

| A1 | A2 |
|---|---|
| -32 | 61.98 |

Arguably, either answer could be correct - depending upon what the user wants. In practice, the first answer is almost always what they intended. The second answer is somewhat flawed because it gives no weighting to the absolute size of the values in each row (i.e. a big SALARY divided by a big COMM is the same as a small divided by a small).

## 24.14. Date Output Order

Db2 has a bind option (called DATETIME) that specifies the default output format of datetime data. This bind option has no impact on the sequence with which date-time data is presented. It simply defines the output template used. To illustrate, the plan that was used to run the following SQL defaults to the USA date-time-format bind option. Observe that the month is the first field printed, but the rows are sequenced by year:

*DATE output in year, month, day order*

```
SELECT hiredate
FROM employee
WHERE hiredate < '1960-01-01'
ORDER BY 1;
```

*ANSWER*

| HIREDATE |
|---|
| 1947-05-05 |
| 1949-08-17 |

| HIREDATE |
| --- |
| 1958-05-16 |

When the CHAR function is used to convert the date-time value into a character value, the sort order is now a function of the display sequence, not the internal date-time order:

*DATE output in month, day, year order*

```
SELECT CHAR(hiredate, USA)
FROM employee
WHERE hiredate < '1960-01-01'
ORDER BY 1;
```

*ANSWER*

| HIREDATE |
| --- |
| 05/05/1947 |
| 05/16/1958 |
| 08/17/1949 |

In general, always bind plans so that date-time values are displayed in the preferred format. Using the CHAR function to change the format can be unwise.

## 24.15. Ambiguous Cursors

The following pseudo-code will fetch all of the rows in the STAFF table (which has ID's ranging from 10 to 350) and, then while still fetching, insert new rows into the same STAFF table that are the same as those already there, but with ID's that are 500 larger.

*Ambiguous Cursor*

```
EXEC-SQL
  DECLARE fred CURSOR FOR
    SELECT *
    FROM staff
    WHERE id < 1000
    ORDER BY id;
END-EXEC;
EXEC-SQL
  OPEN fred
END-EXEC;
DO UNTIL SQLCODE = 100;
  EXEC-SQL
    FETCH fred INTO :HOST-VARS
  END-EXEC;
  IF SQLCODE <> 100 THEN DO;
    SET HOST-VAR.ID = HOST-VAR.ID + 500;
    EXEC-SQL
      INSERT INTO staff VALUES (:HOST-VARS)
    END-EXEC;
  END-DO;
END-DO;
EXEC-SQL
  CLOSE fred
END-EXEC;
```

We want to know how many rows will be fetched, and so inserted? The answer is that it depends upon the indexes available. If there is an index on ID, and the cursor uses that index for the ORDER BY, there will 70 rows fetched and inserted. If the ORDER BY is done using a row sort (i.e. at OPEN CURSOR time) only 35 rows will be fetched and inserted.

Be aware that Db2, unlike some other database products, does NOT (always) retrieve all of the matching rows at OPEN CURSOR time. Furthermore, understand that this is a good thing for it means that Db2 (usually) does not process any row that you do not need. Db2 is very good at always returning the same answer, regardless of the access path used. It is equally good at giving consistent results when the same logical statement is written in a different manner (e.g. A=B vs. B=A). What it has never done consistently (and never will) is guarantee that concurrent read and write statements (being run by the same user) will always give the same results.

## 24.16. Multiple User Interactions

There was once a mythical company that wrote a query to list all orders in the ORDER table for a particular DATE, with the output sequenced by REGION and STATUS. To make the query fly, there was a secondary index on the DATE, REGION, and STATUS columns, in addition to the primary unique index on the ORDER-NUMBER column:

*Select from ORDER table*

```
SELECT region_code  AS region
     , order_status AS status
     , order_number AS order#
     , order_value  AS value
FROM order_table
WHERE order_date = '2006-03-12'
ORDER BY region_code
       , order_status
WITH CS;
```

When the users ran the above query, they found that some orders were seemingly listed twice:

| REGION | STATUS | ORDER# | VALUE |
|---|---|---|---|
| EAST | PAID | 111 | 4.66 (_) |
| EAST | PAID | 222 | 6.33 |
| EAST | PAID | 333 | 123.45 |
| EAST | SHIPPED | 111 | 4.66 (_) |
| EAST | SHIPPED | 444 | 123.45 |

(*) Same ORDER#

While the above query was running (i.e. traversing the secondary index) another user had come along and updated the STATUS for ORDER# 111 from PAID to SHIPPED, and then committed the change. This update moved the pointer for the row down the secondary index, so that the query subsequently fetched the same row twice.

### 24.16.1. Explanation

In the above query, Db2 is working exactly as intended. Because the result may seem a little odd, a simple example will be used to explain what is going on:

Imagine that one wants to count the number of cars parked on a busy street by walking down the road from one end to the other, counting each parked car as you walk past. By the time you get to the end of the street, you will have a number, but that number will not represent the number of cars parked on the street at any point in time. And if a car that you counted at the start of the street was moved to the end of the street while you were walking, you will have counted that particular car twice. Likewise, a car that was moved from the end of the street to the start of the street while you were walking in the middle of the street would not have been counted by you, even though it never left the street during your walk.

One way to get a true count of cars on the street is to prevent car movement while you do your walk. This can be unpopular, but it works. The same can be done in Db2 by changing the WITH phrase (i.e. isolation level) at the bottom of the above query:

### 24.16.2. WITH RR - Repeatable Read

A query defined with repeatable read can be run multiple times and will always return the same result, with the following qualifications:

- References to special registers, like CURRENT TIMESTAMP, may differ.
- Rows changed by the user will show in the query results.

No row will ever be seen twice with this solution, because once a row is read it cannot be changed. And the query result is a valid representation of the state of the table, or at least of the matching rows, as of when the query finished. In the car-counting analogy described above, this solution is akin to locking down sections of the street as you walk past, regardless of whether there is a car parked there or not. As long as you do not move a car yourself, each traverse of the street will always get the same count, and no car will ever be counted more than once. In many cases, defining a query with repeatable read will block all changes by other users to the target table for the duration. In theory, rows can be changed if they are outside the range of the query predicates, but this is not always true. In the case of the order system described above, it was not possible to use this solution because orders were coming in all the time.

### 24.16.3. WITH RS - Read Stability

A query defined with read-stability can be run multiple times, and each row processed previously will always look the same the next time that the query is run - with the qualifications listed above. But rows can be inserted into the table that match the query predicates. These will show in the next run. No row will ever be inadvertently read twice. In our car-counting analogy, this solution is akin to putting a wheel-lock on each parked car as you walk past. The car can't move, ut new cars can be parked in the street while you are counting. The new cars can also leave subsequently, as you long as you don't lock them in your next walk down the street. No car will ever be counted more than once in a single pass, but nor will your count ever represent the true state of the street. As with repeatable read, defining a query with read stability will often block all updates by other users to the target table for the duration. It is not a great way to win friends.

### 24.16.4. WITH CS - Cursor Stability

A query defined with cursor stability will read every committed matching row, occasionally more than once. If the query is run multiple times, it may get a different result each time. In our car-counting analogy, this solution is akin to putting a wheel-lock on each parked car as you count it, but then removing the lock as soon as you move on to the next car. A car that you are not currently counting can be moved anywhere in the street, including to where you have yet to count. In the latter case, you will count it again. This is what happened during our mythical query of the ORDER table. Queries defined with cursor stability still need to take locks, and thus can be delayed if another user has updated a matching row, but not yet done a commit. In extreme cases, the query may get a timeout or deadlock.

### 24.16.5. WITH UR - Uncommitted Read

A query defined with uncommitted read will read every matching row, including those that have not yet been committed. Rows may occasionally be read more than once. If the query is run

multiple times, it may get a different result each time. In our car-counting analogy, this solution is akin to counting each stationary car as one walks past, regardless of whether or not the car is permanently parked. Queries defined with uncommitted read do not take locks, and thus are not delayed by other users who have changed rows, but not yet committed. But some of the rows read may be subsequently rolled back, and so were never valid. Below is a summary of the above options:

*WITH Option vs. Actions*

```
                     SAME RESULT  FETCH SAME UNCOMMITTED ROWS
CURSOR "WITH" OPTION  IF RUN TWICE ROW > ONCE ROWS SEEN   LOCKED
==================== ============ ========== =========== ========
RR - Repeatable Read  Yes          Never      Never       Many/All
RS - Read Stability   No (inserts) Never      Never       Many/All
CS - Cusor Stability  No (all DML) Maybe      Never       Current
UR - Uncommitted Read No (all DML) Maybe      Yes         None
```

# 24.17. Check for Changes, Using Trigger

The target table can have a column of type timestamp that is set to the current timestamp value (using triggers) every time a row is inserted or updated. The query scanning the table can have a predicate (see below) so it only fetches those rows that were updated before the current timestamp, which is the time when the query was opened:

*Select from ORDER table*

```
SELECT region_code  AS region
     , order_status AS status
      , order_number AS order#
     , order_value  AS value
FROM order_table
WHERE order_date = '2006-03-12'
AND update_ts < CURRENT TIMESTAMP      -- <= New predicate
ORDER BY region_code
       , order_status
WITH CS;
```

This solution is almost certainly going to do the job, but it is not quite perfect. There is a very small chance that one can still fetch the same row twice. To illustrate, imagine the following admittedly very improbable sequence of events:

*Sequence of events required to fetch same row twice*

```
#1 UPDATE statement begins (will run for a long time).
#2 QUERY begins (will also run for a long time).
#3 QUERY fetches target row (via secondary index).
#4 QUERY moves on to the next row, etc...
#5 UPDATE changes target row - moves it down index.
#6 UPDATE statement finishes, and commits.
#7 QUERY fetches target row again (bother).
```

## 24.18. Check for Changes, Using Generated TS

A similar solution that will not suffer from the above problem involves adding a timestamp column to the table that is defined GENERATED ALWAYS. This column will be assigned the latest timestamp value (sort of) every time a row is inserted or updated – on a row-by-row basis. Below is an example of a table with this column type:

*Table with ROW CHANGE TIMESTAMP column*

```
CREATE TABLE order_table
( order#       SMALLINT  NOT NULL
, order_date   DATE      NOT NULL
, order_status CHAR(1)   NOT NULL
, order_value  DEC(7, 2) NOT NULL
, order_rct    TIMESTAMP NOT NULL
               GENERATED ALWAYS FOR EACH ROW ON UPDATE
               AS ROW CHANGE TIMESTAMP
, PRIMARY KEY (order#));
```

A query accessing this table that wants to ensure that it does not select the same row twice will include a predicate to check that the order_rct column value is less than or equal to the current timestamp:

*Select from ORDER table*

```
SELECT region_code AS region
     , order_status AS status
     , order_number AS order#
     , order_value AS value
FROM order_table
WHERE order_date = '2006-03-12'
AND order_rct <= CURRENT TIMESTAMP  --  <= New predicate
ORDER BY region_code
       , order_status
WITH CS;
```

There is just one minor problem with this solution: The generated timestamp value is not always exactly the current timestamp. Sometimes it is every so slightly higher. If this occurs, the above

query will not retrieve the affected rows. This problem only occurs during a multi-row insert or update. The generated timestamp value is always unique. To enforce uniqueness, the first row (in a multi-row insert or update) gets the current timestamp special register value. Subsequent rows get the same value, plus "n" microseconds, where "n" incremented by one for each row changed. To illustrate this problem, consider the following statement, which inserts three rows into the above table, but only returns one row- because only the first row inserted has an order_rct value that is equal to or less than the current timestamp special register:

*SELECT from INSERT*

```
SELECT order#
FROM FINAL TABLE
    (INSERT INTO order_table (order#, order_date, order_status, order_value)
     VALUES (1, '2007-11-22', 'A', 123.45)
          , (2,'2007-11-22','A',123.99)
          , (3,'2007-11-22','A',123.99))
WHERE order_rct <= CURRENT TIMESTAMP;
```

*ANSWER*

| order# |
| --- |
| 1 |

The same problem can occur when a query is run immediately after the above insert (i.e. before a commit is done). Occasionally, but by no means always, this query will be use the same current timestamp special register value as the previous insert. If this happens, only the first row inserted will show. **NOTE**: This problem arises in Db2 running on Windows, which has a somewhat imprecise current timestamp value. It should not occur in environments where Db2 references a system clock with microsecond, or sub-microsecond precision.

# 24.19. Other Solutions - Good and Bad

Below are some alternatives to the above:

- **Lock Table**: If one wanted to see the state of the table as it was at the start of the query, one could use a LOCK TABLE command - in share or exclusive mode. Doing this may not win you many friends with other users.
- **Drop Secondary Indexes**: The problem described above does not occur if one accesses the table using a tablespace scan, or via the primary index. However, if the table is large, secondary indexes will probably be needed to get the job done.
- **Two-part Query**: One can do the query in two parts: First get a list of DISTINCT primary key values, then join back to the original table using the primary unique index to get the rest of the row:

*Two-part query*

```
SELECT region_code  AS region
     , order_status AS status
     , order_number AS order#
     , order_value  AS value
FROM (SELECT DISTINCT order_number AS distinct_order#
      FROM order_table
      WHERE order_date = '2006-03-12' ) AS xxx
      , order_table
WHERE order_number = distinct_order#
ORDER BY region_code
       , order_status
WITH CS;
```

This solution will do the job, but it is probably going to take about twice as long to complete as the original query.

- **Use Versions**: See the chapter titled "Retaining a Record" for a schema that uses lots of complex triggers and views, and that lets one see consistent views of the rows in the table as of any point in time.

# 24.20. What Time is It

The **CURRENT TIMESTAMP** special register returns the current time – in local time. There are two other ways to get the something similar the current timestamp. This section discusses the differences:

- **Current Timestamp Special Register**: As its name implies, this special register returns the current timestamp. The value will be the same for all references within a single SQL statement, and possibly between SQL statements and/or between users.
- **Generate Unique Scalar Function**: With a bit of fudging, this scalar function will return a timestamp value that is unique for every invocation. The value will be close to the current timestamp, but may be a few seconds behind.
- **Generate Always Column Type**: This timestamp value will be unique (within a table) for every row changed. In a multi-row insert or update, the first row changed will get the current timestamp. Subsequent rows get the same value, plus "n" microseconds, where "n" incremented by one for each row changed.

The following table will hold the above three values:

*Create table to hold timestamp values*

```
CREATE TABLE test_table
( test# SMALLINT NOT NULL
, current_ts TIMESTAMP NOT NULL
, generate_u TIMESTAMP NOT NULL
, generate_a TIMESTAMP NOT NULL
             GENERATED ALWAYS FOR EACH ROW ON UPDATE
             AS ROW CHANGE TIMESTAMP);
```

The next statement will insert four rows into the above table:

*Insert four rows*

```
INSERT INTO test_table (test#, current_ts, generate_u)
WITH temp1 (t1) AS
(VALUES (1),(2),(3),(4))
, temp2 (t1, ts1, ts2) AS
(SELECT t1
      , CURRENT TIMESTAMP
      , TIMESTAMP(GENERATE_UNIQUE()) + CURRENT TIMEZONE
 FROM temp1)
SELECT *
FROM temp2;
```

Below are the contents of the table after the above insert. Observe the different values: .Table after insert

| TEST# | CURRENT_TS | GENERATE_U | GENERATE_A |
|---|---|---|---|
| 1 | 2007-11-13-19.12.43.139000 | 2007-11-13-19.12.42.973805 | 2007-11-13-19.12.43.139000 |
| 2 | 2007-11-13-19.12.43.139000 | 2007-11-13-19.12.42.974254 | 2007-11-13-19.12.43.154000 |
| 3 | 2007-11-13-19.12.43.139000 | 2007-11-13-19.12.42.974267 | 2007-11-13-19.12.43.154001 |
| 4 | 2007-11-13-19.12.43.139000 | 2007-11-13-19.12.42.974279 | 2007-11-13-19.12.43.154002 |

# 24.21. Floating Point Numbers

The following SQL repetitively multiplies a floating-point number by ten:

*Multiply floating-point number by ten*

```
WITH temp (f1) AS
(VALUES FLOAT(1.23456789)
   UNION ALL
 SELECT f1 * 10
 FROM temp
 WHERE f1 < 1E18)
SELECT f1           AS float1
     , DEC(f1,31,8) AS decimal1
     , BIGINT(f1)   AS bigint1
FROM temp;
```

After a while, things get interesting:

| FLOAT1 | DECIMAL1 | BIGINT1 |
|---|---|---|
| +1.23456789000000E+000 | 1.23456789 | 1 |
| +1.23456789000000E+001 | 12.34567890 | 12 |
| +1.23456789000000E+002 | 123.45678900 | 123 |
| +1.23456789000000E+003 | 1234.56789000 | 1234 |
| +1.23456789000000E+004 | 12345.67890000 | 12345 |
| +1.23456789000000E+005 | 123456.78900000 | 123456 |
| +1.23456789000000E+006 | 1234567.89000000 | 1234567 |
| +1.23456789000000E+007 | 12345678.90000000 | 12345678 |
| +1.23456789000000E+008 | 123456789.00000000 | 123456788 |
| +1.23456789000000E+009 | 1234567890.00000000 | 1234567889 |
| +1.23456789000000E+010 | 12345678900.00000000 | 12345678899 |
| +1.23456789000000E+011 | 123456789000.00000000 | 123456788999 |
| +1.23456789000000E+012 | 1234567890000.00000000 | 1234567889999 |
| +1.23456789000000E+013 | 12345678900000.00000000 | 12345678899999 |
| +1.23456789000000E+014 | 123456789000000.00000000 | 123456788999999 |
| +1.23456789000000E+015 | 1234567890000000.00000000 | 1234567889999999 |
| +1.23456789000000E+016 | 12345678900000000.00000000 | 12345678899999998 |
| +1.23456789000000E+017 | 123456789000000000.00000000 | 123456788999999984 |
| +1.23456789000000E+018 | 1234567890000000000.00000000 | 1234567889999999744 |

Why do the BIGINT values differ from the original float values? The answer is that they don't, it is the decimal values that differ. Because this is not what you see in front of your eyes, we need to explain. Note that there are no bugs here, everything is working fine. Perhaps the most insidious problem involved with using floating point numbers is that the number you see is not always the number that you have. Db2 stores the value internally in binary format, and when it displays it, it

shows a decimal approximation of the underlying binary value. This can cause you to get very strange results like the following:

*Two numbers that look equal, but aren't equal*

```
WITH temp (f1, f2) AS
(VALUES (FLOAT(1.23456789E1 * 10 * 10 * 10 * 10 * 10 * 10 * 10)
       , FLOAT(1.23456789E8)))
SELECT f1
     , f2
FROM temp
WHERE f1 <> f2;
```

*ANSWER*

| F1 | F2 |
|---|---|
| +1.23456789000000E+008 | +1.23456789000000E+008 |

We can use the HEX function to show that, internally, the two numbers being compared above are not equal:

*Two numbers that look equal, but aren't equal, shown in HEX*

```
WITH temp (f1, f2) AS
(VALUES (FLOAT(1.23456789E1 * 10 * 10 * 10 * 10 * 10 * 10 * 10)
       , FLOAT(1.23456789E8)))
SELECT HEX(f1) AS hex_f1
     , HEX(f2) AS hex_f2
FROM temp
WHERE f1 <> f2;
```

*ANSWER*

| HEX_F1 | HEX_F2 |
|---|---|
| FFFFFF53346F9D41 | 00000054346F9D41 |

Now we can explain what is going on in the recursive code shown at the start of this section. The same value is being displayed using three different methods:

- The floating-point representation (on the left) is really a decimal approximation (done using rounding) of the underlying binary value.
- When the floating-point data was converted to decimal (in the middle), it was rounded using the same method that is used when it is displayed directly.
- When the floating-point data was converted to BIGINT (on the right), no rounding was done because both formats hold binary values.

In any computer-based number system, when you do division, you can get imprecise results due to

rounding. For example, when you divide 1 by 3 you get "one third", which can not be stored accurately in either a decimal or a binary number system. Because they store numbers internally differently, dividing the same number in floating-point vs. decimal can result in different results. Here is an example:

*Comparing float and decimal division*

```
WITH temp1 (dec1, dbl1) AS
(VALUES (DECIMAL(1),DOUBLE(1)))
, temp2 (dec1, dec2, dbl1, dbl2) AS
(SELECT dec1
      , dec1 / 3 AS dec2
      , dbl1
      , dbl1 / 3 AS dbl2
 FROM temp1)
SELECT *
FROM temp2
WHERE dbl2 <> dec2;
```

*ANSWER (1 row returned)*

```
DEC1 = 1.0
DEC2 = 0.33333333333333333
DBL1 = +1.00000000000000E+000
DBL2 = +3.33333333333333E-001
```

When you do multiplication of a fractional floating-point number, you can also encounter rounding differences with respect to decimal. To illustrate this, the following SQL starts with two numbers that are the same, and then keeps multiplying them by ten:

*Comparing float and decimal multiplication*

```
WITH temp (f1, d1) AS
(VALUES (FLOAT(1.23456789)
       , DEC(1.23456789,20,10))
   UNION ALL
 SELECT f1 * 10
      , d1 * 10
 FROM temp
 WHERE f1 < 1E9)
SELECT f1
     , d1
     , CASE
         WHEN d1 = f1 THEN 'SAME'
         ELSE 'DIFF'
       END AS compare
FROM temp;
```

Here is the answer:

| F1 | D1 | COMPARE |
|---|---|---|
| +1.23456789000000E+000 | 1.2345678900 | SAME |
| +1.23456789000000E+001 | 12.3456789000 | SAME |
| +1.23456789000000E+002 | 123.4567890000 | DIFF |
| +1.23456789000000E+003 | 1234.5678900000 | DIFF |
| +1.23456789000000E+004 | 12345.6789000000 | DIFF |
| +1.23456789000000E+005 | 123456.7890000000 | DIFF |
| +1.23456789000000E+006 | 1234567.8900000000 | SAME |
| +1.23456789000000E+007 | 12345678.9000000000 | DIFF |
| +1.23456789000000E+008 | 123456789.0000000000 | DIFF |
| +1.23456789000000E+009 | 1234567890.0000000000 | DIFF |

As we mentioned earlier, both floating-point and decimal fields have trouble accurately storing certain fractional values. For example, neither can store "one third". There are also some numbers that can be stored in decimal, but not in floating-point. One common value is "one tenth", which as the following SQL shows, is approximated in floating-point:

*Internal representation of "one tenth" in floating-point*

```
WITH temp (f1) AS
(VALUES FLOAT(0.1))
SELECT f1
     , HEX(f1) AS hex_f1
FROM temp
WHERE f1 <> 1.0;
```

*ANSWER*

| F1 | HEX_F1 |
|---|---|
| +1.00000000000000E-001 | 9A9999999999B93F |

In conclusion, a floating-point number is, in many ways, only an approximation of a true integer or decimal value. For this reason, this field type should not be used for monetary data, nor for other data where exact precision is required.

# 24.22. DECFLOAT Usage

We can avoid the problems described above if we use a DECFLOAT value. To illustrate, the following query is exactly the same as that shown on Floating Point Numbers, except that base value is now of type DECFLOAT:

*Multiply DECFLOAT number by ten*

```
WITH temp (f1) AS
(VALUES DECFLOAT(1.23456789)
   UNION ALL
 SELECT f1 * 10
 FROM temp
 WHERE f1 < 1E18)
SELECT f1           AS float1
     , DEC(f1,31,8) AS decimal1
     , BIGINT(f1)   AS bigint1
FROM temp;
```

Now we get the result that we expect:

| FLOAT1 | DECIMAL1 | BIGINT1 |
|---|---|---|
| +1.23456789000000E+000 | 1.23456789 | 1 |
| +1.23456789000000E+001 | 12.34567890 | 12 |
| +1.23456789000000E+002 | 123.45678900 | 123 |
| +1.23456789000000E+003 | 1234.56789000 | 1234 |
| +1.23456789000000E+004 | 12345.67890000 | 12345 |
| +1.23456789000000E+005 | 123456.78900000 | 123456 |
| +1.23456789000000E+006 | 1234567.89000000 | 1234567 |
| +1.23456789000000E+007 | 12345678.90000000 | 12345678 |
| +1.23456789000000E+008 | 123456789.00000000 | 123456789 |
| +1.23456789000000E+009 | 1234567890.00000000 | 1234567890 |
| +1.23456789000000E+010 | 12345678900.00000000 | 12345678900 |
| +1.23456789000000E+011 | 123456789000.00000000 | 123456789000 |
| +1.23456789000000E+012 | 1234567890000.00000000 | 1234567890000 |
| +1.23456789000000E+013 | 12345678900000.00000000 | 12345678900000 |
| +1.23456789000000E+014 | 123456789000000.00000000 | 123456789000000 |
| +1.23456789000000E+015 | 1234567890000000.00000000 | 1234567890000000 |
| +1.23456789000000E+016 | 12345678900000000.00000000 | 12345678900000000 |
| +1.23456789000000E+017 | 123456789000000000.00000000 | 123456789000000000 |
| +1.23456789000000E+018 | 1234567890000000000.00000000 | 1234567890000000000 |

# Chapter 25. Time Travel

This chapter gives an overview on the time travel feature in Db2. It is based on an article by Jan-Eike Michels and Matthias Nicola published on October 18, 2012 on developerWorks [https://www.ibm.com/developerworks/data/library/techarticle/dm-1210temporaltablesdb2/index.html].

We don't go that much into detail, just show you how it works. The basic idea is: sometimes you need to keep track on how the data changes over time in the database. While it is possible to track the changes using old database features like triggers (as described in the chapter Retaining a Record), you will get a very robust solution utilizing the native features of the database. The database solution target two different scenarios:

- track all the changes to data (System-period) and
- track the validity of data for some time interval (Application-period).

You can have both at the same time defined for one table (Db2 call this a "Bitemporal" table), i.e., you let the database keep track of all changes and at the same time manage the business validity of the data for you.

To show time travel in action let's take a similar table as we used in chapter Retaining a Record:

*MyCustomer table*

```
CREATE TABLE mycustomer
( cust#     INTEGER NOT NULL
, cust_name CHAR(10)
, cust_mgr  CHAR(10)
, PRIMARY KEY(cust#));
```

If you query this table you will always get one manager (the last one) for the customer. If you need to know who was the manager for this customer im some date in the past, you just cannot do it. If this is a need in our business, we can profit from the time travel feature. To implement it for this table we will need the following steps:

- add new columns for the table (start, end and transaction) with the following characteristics:
  - system_begin TIMESTAMP(12) NOT NULL GENERATED ALWAYS AS ROW BEGIN
  - system_end TIMESTAMP(12) NOT NULL GENERATED ALWAYS AS ROW END
  - trans_start TIMESTAMP(12) GENERATED ALWAYS AS TRANSACTION START ID
- tell the table that this columns will manage the time slices when the data was active in the database:
  - PERIOD SYSTEM_TIME (system_begin, system_end)
- define a new table to save the history data:
  - CREATE TABLE mycustomer_history LIKE customer

- instruct the database management system to activate the time travel feature for this object:
  - ALTER TABLE mycustomer ADD VERSIONING USE HISTORY TABLE mycustomer_history

*DDL to implement the history for mycustomer*

```
ALTER TABLE mycustomer
  ADD COLUMN system_begin  TIMESTAMP(12) NOT NULL GENERATED ALWAYS AS ROW BEGIN
IMPLICITLY HIDDEN
  ADD COLUMN system_end    TIMESTAMP(12) NOT NULL GENERATED ALWAYS AS ROW END
IMPLICITLY HIDDEN
  ADD COLUMN trans_start   TIMESTAMP(12)          GENERATED ALWAYS AS TRANSACTION
START ID
  ADD PERIOD SYSTEM_TIME (system_begin, system_end);

CREATE TABLE mycustomer_history LIKE mycustomer;

ALTER TABLE mycustomer ADD VERSIONING USE HISTORY TABLE mycustomer_history;
```

Now suppose we have the following time line in our company:

*Table 17. Events time line for time travel*

| Date | Event |
|---|---|
| 01.01.2019 | First customer is registered in the database. Name: Customer. Manager: Mgr 1 |
| 01.07.2019 | Second customer is registered in the database. Name: Customer2. Manager: Mgr 1 |
| 01.12.2019 | Third customer is registered in the database. Name: Customer3. Manager: Mgr 2 |
| 12.01.2020 | Fourth customer is registerd in the database. Name: Customer4. Manager: Mgr 3 |
| 15.01.2020 | Fourth customer is deleted from the database. |
| 16.01.2020 | Name of first customer is changed from Customer to Customer1 |
| 17.01.2020 | Manager of second customer is changed from Mgr 1 to Mgr 2 |
| 18.01.2020 | Manager of third customer is changed from Mgr 2 to Mgr 3 |

Now let us select the current data from our table: .Select current data from table

```
SELECT *
  FROM mycustomer;
```

*Table 18. Current data*

| CUST# | CUST_NAME | CUST_MGR |
|---|---|---|
| 1 | Customer1 | Mgr 1 |
| 2 | Customer2 | Mgr 2 |

| CUST# | CUST_NAME | CUST_MGR |
|---|---|---|
| 3 | Customer3 | Mgr 3 |

We don't see Customer4 because she was already deleted from the database. If we want to see the data as it was on 12.1.2020, we can do it!

*Select table data as it was on 12.1.2020*

```
SELECT *
  FROM mycustomer
  FOR SYSTEM_TIME AS OF '2020-01-12'
```

*Table 19. Data on 12.1.2020*

| CUST# | CUST_NAME | CUST_MGR |
|---|---|---|
| 1 | Customer | Mgr 1 |
| 2 | Customer2 | Mgr 1 |
| 3 | Customer3 | Mgr 2 |
| 4 | Customer4 | Mgr 3 |

All the changes were done after this date, so we see changes in every single record! Because we defined the columns SYSTEM_BEGIN and SYSTEM_END as IMPLICITLY HIDDEN, we don't get them in the result set. If we want to see them we need to explicitly code their names in our select statement:

*Select table data as it was on 12.1.2020, show all columns*

```
SELECT cust#, cust_name, cust_mgr, system_begin, system_end
  FROM mycustomer
  FOR SYSTEM_TIME AS OF '2020-01-12'
```

*Table 20. Table data as it was on 12.1.2020, show all columns*

| CUST# | CUST_NAME | CUST_MGR | SYSTEM_BEGIN | SYSTEM_END |
|---|---|---|---|---|
| 1 | Customer | Mgr 1 | 2019-01-01 00:00:00.0 | 2020-01-16 00:00:00.0 |
| 2 | Customer2 | Mgr 1 | 2019-07-01 00:00:00.0 | 2020-01-17 00:00:00.0 |
| 3 | Customer3 | Mgr 2 | 2019-12-01 00:00:00.0 | 2020-01-18 00:00:00.0 |
| 4 | Customer4 | Mgr 3 | 2020-01-12 00:00:00.0 | 2020-01-15 00:00:00.0 |

Because the content of SYSTEM_END is always different from the default value of a "ROW END" column in the base table ('9999-12-30 00:00:00.0'), we know that all data came from the history table!

In the first version of the time travel feature IBM used '9999-12-31 00:00:00.0' as default value for the row end. That worked fine when you didn't work in different time zones! This is the reason why it changed to 9999-12-30: to avoid overflows.

The history table is a normal table and you can query it. If you want to show all the events in the table mycustomer, you could write the following query:

*Select all events (all data) from both tables*

```
select 'myCustomer' as Tablename, cust#, cust_name, cust_mgr, system_begin, system_end
from mycustomer
union all
select 'myCustomer_History', cust#, cust_name, cust_mgr, system_begin, system_end
from mycustomer_history
order by system_begin;
```

*Table 21. All table data, order by start of event*

| TABLENAME | CUST# | CUST_NAME | CUST_MGR | SYSTEM_BEGIN | SYSTEM_END |
|---|---|---|---|---|---|
| myCustomer_History | 1 | Customer | Mgr 1 | 2019-01-01 00:00:00.0 | 2020-01-16 00:00:00.0 |
| myCustomer_History | 2 | Customer2 | Mgr 1 | 2019-07-01 00:00:00.0 | 2020-01-17 00:00:00.0 |
| myCustomer_History | 3 | Customer3 | Mgr 2 | 2019-12-01 00:00:00.0 | 2020-01-18 00:00:00.0 |
| myCustomer_History | 4 | Customer4 | Mgr 3 | 2020-01-12 00:00:00.0 | 2020-01-15 00:00:00.0 |
| myCustomer | 1 | Customer1 | Mgr 1 | 2020-01-16 00:00:00.0 | 9999-12-30 00:00:00.0 |
| myCustomer | 2 | Customer2 | Mgr 2 | 2020-01-17 00:00:00.0 | 9999-12-30 00:00:00.0 |
| myCustomer | 3 | Customer3 | Mgr 3 | 2020-01-18 00:00:00.0 | 9999-12-30 00:00:00.0 |

Now we can answer the question we posted at the beginning: who was the manager Customer2 on Christmas in year 2019?

*Select Manager for a specific date*

```
select cust_name, cust_mgr
  FROM mycustomer
  FOR SYSTEM_TIME AS OF '2019-12-25'
```

*Table 22. Result for a specific date*

| CUST_NAME | CUST_MGR |
|---|---|
| Customer | Mgr 1 |
| Customer2 | Mgr 1 |
| Customer3 | Mgr 2 |

Now if some customer is unsatisfied because she didn’t get a xmas gift, you can blame the right person.

# Chapter 26. Appendix

## 26.1. Db2 Sample Tables

Sample table DDL follows.

### 26.1.1. ACT

*ACT sample table – DDL*

```
CREATE TABLE ACT
( ACTNO   SMALLINT     NOT NULL
, ACTKWD  CHARACTER(6) NOT NULL
, ACTDESC VARCHAR(20)  NOT NULL)
IN USERSPACE1;

ALTER TABLE ACT
  ADD CONSTRAINT PK_ACT PRIMARY KEY (ACTNO);

CREATE UNIQUE INDEX XACT2
ON ACT(ACTNO ASC, ACTKWD ASC)
ALLOW REVERSE SCANS;
```

### 26.1.2. CATALOG

*CATALOG sample table – DDL*

```
CREATE TABLE CATALOG
( NAME VARCHAR(128) NOT NULL
, CATLOG XML)
IN IBMDB2SAMPLEXML;

ALTER TABLE CATALOG
ADD CONSTRAINT PK_CATALOG PRIMARY KEY (NAME);
```

### 26.1.3. CL_SCHED

*CL_SCHED sample table – DDL*

```
CREATE TABLE CL_SCHED
( CLASS_CODE CHARACTER(7)
, DAY        SMALLINT
, STARTING   TIME
, ENDING     TIME)
IN USERSPACE1;
```

### 26.1.4. CUSTOMER

*CUSTOMER sample table – DDL*

```
CREATE TABLE CUSTOMER
( CID BIGINT NOT NULL
, INFO XML
, HISTORY XML)
IN IBMDB2SAMPLEXML;

ALTER TABLE CUSTOMER
ADD CONSTRAINT PK_CUSTOMER PRIMARY KEY(CID);
```

### 26.1.5. DATA_FILE_NAMES

*DATA_FILE_NAMES sample table – DDL*

```
CREATE TABLE DATA_FILE_NAMES
( DATA_FILE_NAME VARCHAR(40)   NOT NULL
, DB2_TABLE_NAME VARCHAR(40)   NOT NULL
, EXPORT_FILE_NAM CHARACTER(8) NOT NULL)
IN IBMDB2SAMPLEREL;
```

### 26.1.6. DEPARTMENT

*DEPARTMENT sample table – DDL*

```
CREATE TABLE DEPARTMENT
( DEPTNO   CHARACTER(3) NOT NULL
, DEPTNAME VARCHAR(36)  NOT NULL
, MGRNO    CHARACTER(6)
, ADMRDEPT CHARACTER(3) NOT NULL
, LOCATION CHARACTER(16))
IN USERSPACE1;

ALTER TABLE DEPARTMENT
ADD CONSTRAINT PK_DEPARTMENT PRIMARY KEY(DEPTNO);

CREATE INDEX XDEPT2
ON DEPARTMENT(MGRNO ASC)
ALLOW REVERSE SCANS;

CREATE INDEX XDEPT3
ON DEPARTMENT(ADMRDEPT ASC)
ALLOW REVERSE SCANS;

CREATE ALIAS DEPT FOR DEPARTMENT;
```

### 26.1.7. EMPLOYEE

*EMPLOYEE sample table – DDL*

```
CREATE TABLE EMPLOYEE
( EMPNO     CHARACTER(6) NOT NULL
, FIRSTNME  VARCHAR(12)  NOT NULL
, MIDINIT   CHARACTER(1)
, LASTNAME  VARCHAR(15)  NOT NULL
, WORKDEPT  CHARACTER(3)
, PHONENO   CHARACTER(4)
, HIREDATE  DATE
, JOB       CHARACTER(8)
, EDLEVEL   SMALLINT     NOT NULL
, SEX       CHARACTER(1)
, BIRTHDATE DATE
, SALARY    DECIMAL(9, 2)
, BONUS     DECIMAL(9, 2)
, COMM      DECIMAL(9, 2))
IN USERSPACE1;

ALTER TABLE EMPLOYEE
ADD CONSTRAINT PK_EMPLOYEE PRIMARY KEY (EMPNO);

CREATE INDEX XEMP2
ON EMPLOYEE(WORKDEPT ASC)
ALLOW REVERSE SCANS;

CREATE ALIAS EMP FOR EMPLOYEE;
```

### 26.1.8. EMPMDC

*EMPMDC sample table – DDL*

```
CREATE TABLE EMPMDC
( EMPNO INTEGER
, DEPT  INTEGER
, DIV   INTEGER)
IN IBMDB2SAMPLEREL;
```

### 26.1.9. EMPPROJACT

*EMPPROJACT sample table – DDL*

```
CREATE TABLE EMPPROJACT
( EMPNO    CHARACTER(6) NOT NULL
, PROJNO   CHARACTER(6) NOT NULL
, ACTNO    SMALLINT     NOT NULL
, EMPTIME  DECIMAL(5, 2)
, EMSTDATE DATE
, EMENDATE DATE)
IN USERSPACE1;

CREATE ALIAS EMP_ACT FOR EMPPROJACT;
CREATE ALIAS EMPACT FOR EMPPROJACT;
```

### 26.1.10. EMP_PHOTO

*EMP_PHOTO sample table – DDL*

```
CREATE TABLE EMP_PHOTO
( EMPNO        CHARACTER(6)  NOT NULL
, PHOTO_FORMAT VARCHAR(10)   NOT NULL
, PICTURE      BLOB(102400)
, EMP_ROWID    CHARACTER(40) NOT NULL)
IN USERSPACE1;

ALTER TABLE EMP_PHOTO
ADD CONSTRAINT PK_EMP_PHOTO PRIMARY KEY(EMPNO, PHOTO_FORMAT);
```

### 26.1.11. EMP_RESUME

*EMP_RESUME sample table – DDL*

```
CREATE TABLE EMP_RESUME
( EMPNO CHARACTER(6) NOT NULL
, RESUME_FORMAT VARCHAR(10) NOT NULL
, RESUME CLOB(5120)
, EMP_ROWID CHARACTER(40) NOT NULL)
IN USERSPACE1;

ALTER TABLE EMP_RESUME
ADD CONSTRAINT PK_EMP_RESUME PRIMARY KEY(EMPNO, RESUME_FORMAT);
```

### 26.1.12. IN_TRAY

*IN_TRAY sample table – DDL*

```
CREATE TABLE IN_TRAY
( RECEIVED TIMESTAMP
, SOURCE CHARACTER(8)
, SUBJECT CHARACTER(64)
, NOTE_TEXT VARCHAR(3000))
IN USERSPACE1;
```

*INVENTORY sample table – DDL*

```
CREATE TABLE INVENTORY
( PID VARCHAR(10) NOT NULL
, QUANTITY INTEGER
, LOCATION VARCHAR(128))
IN IBMDB2SAMPLEXML;

ALTER TABLE INVENTORY
ADD CONSTRAINT PK_INVENTORY PRIMARY KEY(PID);
```

### 26.1.13. ORG

*ORG sample table – DDL*

```
CREATE TABLE ORG
( DEPTNUMB SMALLINT NOT NULL
, DEPTNAME VARCHAR(14)
, MANAGER SMALLINT
, DIVISION VARCHAR(10)
, LOCATION VARCHAR(13))
IN USERSPACE1;
```

### 26.1.14. PRODUCT

*PRODUCT sample table – DDL*

```
CREATE TABLE PRODUCT
( PID VARCHAR(10) NOT NULL
, NAME VARCHAR(128)
, PRICE DECIMAL(30,2)
, PROMOPRICE DECIMAL(30, 2)
, PROMOSTART DATE
, PROMOEND DATE
, DESCRIPTION XML)
IN IBMDB2SAMPLEXML;

ALTER TABLE PRODUCT
ADD CONSTRAINT PK_PRODUCT PRIMARY KEY(PID);
```

### 26.1.15. PRODUCTSUPPLIER

*PRODUCTSUPPLIER sample table – DDL*

```
CREATE TABLE PRODUCTSUPPLIER
( PID VARCHAR(10) NOT NULL
, SID VARCHAR(10) NOT NULL)
IN IBMDB2SAMPLEXML;
```

### 26.1.16. PROJACT

*PROJACT sample table – DDL*

```
CREATE TABLE PROJACT
( PROJNO CHARACTER(6) NOT NULL
, ACTNO SMALLINT NOT NULL
, ACSTAFF DECIMAL(5, 2)
, ACSTDATE DATE NOT NULL
, ACENDATE DATE)
IN USERSPACE1;

ALTER TABLE PROJACT
ADD CONSTRAINT PK_PROJACT PRIMARY KEY(PROJNO, ACTNO, ACSTDATE);
```

### 26.1.17. PROJECT

*PROJECT sample table – DDL*

```
CREATE TABLE PROJECT
( PROJNO CHARACTER(6) NOT NULL
, PROJNAME VARCHAR(24) NOT NULL
, DEPTNO CHARACTER(3) NOT NULL
, RESPEMP CHARACTER(6) NOT NULL
, PRSTAFF DECIMAL(5, 2)
, PRSTDATE DATE
, PRENDATE DATE
, MAJPROJ CHARACTER(6))
IN USERSPACE1;

ALTER TABLE PROJECT
ADD CONSTRAINT PK_PROJECT PRIMARY KEY(PROJNO);

CREATE INDEX XPROJ2
ON PROJECT(RESPEMP ASC)
ALLOW REVERSE SCANS;

CREATE ALIAS PROJ FOR PROJECT;
```

### 26.1.18. PURCHASEORDER

*PURCHASEORDER sample table – DDL*

```
CREATE TABLE PURCHASEORDER
( POID BIGINT NOT NULL
, STATUS VARCHAR(10) NOT NULL
, CUSTID BIGINT
, ORDERDATE DATE
, PORDER XML
, COMMENTS VARCHAR(1000))
IN IBMDB2SAMPLEXML;

ALTER TABLE PURCHASEORDER
ADD CONSTRAINT PK_PURCHASEORDER PRIMARY KEY(POID);
```

### 26.1.19. SALES

*SALES sample table – DDL*

```
CREATE TABLE SALES
( SALES_DATE DATE
, SALES_PERSON VARCHAR(15)
, REGION VARCHAR(15)
, SALES INTEGER)
IN USERSPACE1;
```

### 26.1.20. STAFF

*STAFF sample table – DDL*

```
CREATE TABLE STAFF
( ID SMALLINT NOT NULL
, NAME VARCHAR(9)
, DEPT SMALLINT
, JOB CHARACTER(5)
, YEARS SMALLINT
, SALARY DECIMAL(7, 2)
, COMM DECIMAL(7,2))
IN USERSPACE1;
```

### 26.1.21. SUPPLIERS

*SUPPLIERS sample table – DDL*

```
CREATE TABLE SUPPLIERS
( SID VARCHAR(10) NOT NULL
, ADDR XML)
IN IBMDb2SAMPLEXML;

ALTER TABLE SUPPLIERS
ADD CONSTRAINT PK_PRODUCTSUPPLIER PRIMARY KEY(SID);
```

# Chapter 27. Thank you, Graeme Birchall!

This chapter contains the last introduction, history and comments from Graeme Birchall in his "Db2 SQL Cookbook". It is a kind of tribute to him and his work.

Db2 LUW V9.7 SQL Cookbook Graeme Birchall 16-Aug-2011

## 27.1. Preface Important!

If you didn't get this document directly from my personal website, you may have got an older edition. The book is changed very frequently, so if you want the latest, go to the source. Also, the latest edition is usually the best book to have, as the examples are often much better. This is true even if you are using an older version of Db2. This Cookbook is written for Db2 for LUW (i.e. Linux, Unix, Windows). It is not suitable for Db2 for z/OS unless you are running Db2 8 in new-function-mode, or (even better) Db2 9.

## 27.2. Acknowledgments

I did not come up with all of the ideas presented in this book. Many of the best examples were provided by readers, friends, and/or coworkers too numerous to list. Thanks also to the many people at IBM for their (strictly unofficial) assistance.

## 27.3. Disclaimer & Copyright

DISCLAIMER: This document is a best effort on my part. However, I screw up all the time, so it would be extremely unwise to trust the contents in its entirety. I certainly don't. And if you do something silly based on what I say, life is tough. COPYRIGHT: You can make as many copies of this book as you wish. And I encourage you to give it to others. But you cannot charge for it (other than to recover reproduction costs), nor claim the material as your own, nor replace my name with another. You are also encouraged to use the related class notes for teaching. In this case, you can charge for your time and materials - and your expertise. But you cannot charge any licensing fee, nor claim an exclusive right of use. In other words, you can pretty well do anything you want. And if you find the above too restrictive, just let me know. TRADEMARKS: Lots of words in this document, like "Db2", are registered trademarks of the IBM Corporation. Lots of other words, like "Windows", are registered trademarks of the Microsoft Corporation. Acrobat is a registered trademark of the Adobe Corporation.

## 27.4. Tools Used

This book was written on a Dell PC that came with oodles of RAM. All testing was done in Db2 V9.7 Express-C for Windows. Word for Windows was used to write the document. Adobe Acrobat was used to make the PDF file.

## 27.5. Book Binding

This book looks best when printed on a doubled sided laser printer and then suitably bound. To this

end, I did some experiments a few years ago to figure out how to bind books cheaply using commonly available materials. I came up with what I consider to be a very satisfactory solution that is fully documented on Book Binding.

# 27.6. Author / Book

Author: Email: Graeme_Birchall@verizon.net Web: http://mysite.verizon.net/Graeme_Birchall/ Title: Db2 9.7 SQL Cookbook © Date: 16-Aug-2011

# 27.7. Preface

## 27.7.1. Author

**Notes Book History**

This book originally began a series of notes for my own use. After a while, friends began to ask for copies, and enemies started to steal it, so I decided to tidy everything up and give it away. Over the years, new chapters have been added as Db2 has evolved, and as I have found new ways to solve problems. Hopefully, this process will continue for the foreseeable future.

## 27.7.2. Why Free

This book is free because I want people to use it. The more people that use it, and the more that it helps them, the more inclined I am to keep it up to date. For these reasons, if you find this book to be useful, please share it with others. This book is free, rather than formally published, because I want to deliver the best product that I can. If I had a publisher, I would have the services of an editor and a graphic designer, but I would not be able to get to market so quickly, and when a product changes as quickly as Db2 does, timeliness is important. Also, giving it away means that I am under no pressure to make the book marketable. I simply include whatever I think might be useful.

## 27.7.3. Other Free Documents

The following documents are also available for free from my web site: SAMPLE SQL: The complete text of the SQL statements in this Cookbook is available in an HTML file. Only the first and last few lines of the file have HTML tags, the rest is raw text, so it can easily be cut and paste into other files. CLASS OVERHEADS: Selected SQL examples from this book have been rewritten as class overheads. This enables one to use this material to teach Db2 SQL to others. Use this cookbook as the student notes. OLDER EDITIONS: This book is rewritten, and usually much improved, with each new version of Db2. Some of the older editions are available from my website. The others can be emailed upon request. However, the latest edition is the best, so you should probably use it, regardless of the version of Db2 that you have.

## 27.7.4. Answering Questions

As a rule, I do not answer technical questions because I need to have a life. But I'm interested in hearing about interesting SQL problems, and also about any bugs in this book. However you may

not get a prompt response, or any response. And if you are obviously an idiot, don't be surprised if I point out (for free, remember) that you are an idiot.

### 27.7.5. Software Whines

This book is written using Microsoft Word for Windows. I've been using this software for many years, and it has generally been a bunch of bug-ridden junk. I do confess that it has been mildly more reliable in recent years. However, I could have written more than twice as much that was twice as good in half the time - if it weren't for all of the bugs in Word.

Graeme

### 27.7.6. Graeme Birchall Book Editions Upload Dates

| Date Published (Version) | Content |
|---|---|
| 1996-05-08 | First edition of the Db2 V2.1.1 SQL Cookbook was posted to my web site. This version was in Postscript Print File format. |
| 1998-02-26 | The Db2 V2.1.1 SQL Cookbook was converted to an Adobe Acrobat file and posted to my web site. Some minor cosmetic changes were made. |
| 1998-08-19 | First edition of Db2 UDB V5 SQL Cookbook posted. Every SQL statement was checked for V5, and there were new chapters on OUTER JOIN and GROUP BY. |
| 1998-08-26 | About 20 minor cosmetic defects were corrected in the V5 Cookbook. |
| 1998-09-03 | Another 30 or so minor defects were corrected in the V5 ookbook. |
| 1998-10-24 | The Cookbook was updated for Db2 UDB V5.2. |
| 1998-10-25 | About twenty minor typos and sundry cosmetic defects were fixed. |
| 1998-12-03 | This book was based on the second edition of the V5.2 upgrade. |
| 1999-01-25 | A chapter on Summary Tables (new in the Dec/98 fixpack) was added and all the SQL was checked for changes. |
| 1999-01-28 | Some more SQL was added to the new chapter on Summary Tables. |
| 1999-02-15 | The section of stopping recursive SQL statements was completely rewritten, and a new section was added on denormalizing hierarchical data structures. |
| 1999-02-16 | Minor editorial changes were made. |
| 1999-03-16 | Some bright spark at IBM pointed out that my new and improved section on stopping recursive SQL was all wrong. Damn. I undid everything. |
| 1999-05-12 | Minor editorial changes were made, and one new example (on getting multiple counts from one value) was added. |

| Date Published (Version) | Content |
|---|---|
| 1999-09-16 | Db2 V6.1 edition. All SQL was rechecked, and there were some minor additions - especially to summary tables, plus a chapter on "Db2 Dislikes". |
| 1999-09-23 | Some minor layout changes were made. |
| 1999-10-06 | Some bugs fixed, plus new section on index usage in summary tables. |
| 2000-04-12 | Some typos fixed, and a couple of new SQL tricks were added. |
| 2000-09-19 | Db2 V7.1 edition. All SQL was rechecked. The new areas covered are: OLAP functions (whole chapter), ISO functions, and identity columns. |
| 2000-09-25 | Some minor layout changes were made. |
| 2000-10-26 | More minor layout changes. |
| 2001-01-03 | Minor layout changes (to match class notes). |
| 2001-02-06 | Minor changes, mostly involving the RAND function. |
| 2001-04-11 | Document new features in latest fixpack. Also add a new chapter on Identity Columns and completely rewrite sub-query chapter. |
| 2001-10-24 | Db2 V7.2 fixpack 4 edition. Tested all SQL and added more examples, plus a new section on the aggregation function. |
| 2002-03-11 | Minor changes, mostly to section on precedence rules. |
| 2002-08-20 | Db2 V8.1 (beta) edition. A few new functions are added. New section on temporary tables. Identity Column and Join chapters rewritten. Whine chapter removed. |
| 2003-01-02 | Db2 V8.1 (post-Beta) edition. SQL rechecked. More examples added. |
| 2003-07-11 | New sections added on DML, temporary tables, compound SQL, and user defined functions. Halting recursion section changed to use ser-defined function. |
| 2003-09-04 | New sections on complex joins and history tables. |
| 2003-10-02 | Minor changes. Some more user-defined functions. |
| 2003-11-20 | Added "quick find" chapter. |
| 2003-12-31 | Tidied up the SQL in the Recursion chapter, and added a section on the merge statement. Completely rewrote the chapter on materialized query tables. |
| 2004-02-04 | Added select-from-DML section, and tidied up some code. Also managed to waste three whole days due to bugs in Microsoft Word. |
| 2004-07-23 | Rewrote chapter of identity column and sequences. Made DML separate chapter. Added chapters on protecting data and XML functions. Other minor changes. |

| Date Published (Version) | Content |
|---|---|
| 2004-11-03 | Upgraded to V8.2. Retested all SQL. Documented new SQL features. Some major hacking done on the GROUP BY chapter. |
| 2005-04-15 | Added short section on cursors, and a chapter on using SQL to make SQL. |
| 2005-06-01 | Added a chapter on triggers. |
| 2005-11-11 | Updated MQT table chapter and added bibliography. Other minor changes. |
| 2005-12-01 | Applied fixpack 10. Changed my website name. |
| 2005-12-16 | Added notes on isolation levels, data-type functions, transforming data. |
| 2006-01-26 | Fixed dumb bugs generated by WORD. What stupid software. Also wrote an awesome new section on joining meta-data to real data. |
| 2006-02-17 | Touched up the section on joining meta-data to real data. Other minor fixes. |
| 2006-02-27 | Added precedence rules for SQL statement processing, and a description of a simplified nested table expression. |
| 2006-03-23 | Added better solution to avoid fetching the same row twice. |
| 2006-04-26 | Added trigger that can convert HEX value to number. |
| 2006-09-08 | Upgraded to V9.1. Retested SQL. Removed the XML chapter as it is now obsolete. I'm still cogitating about XQuery. Looks hard. Added some awesome java code. |
| 2006-09-13 | Fixed some minor problems in the initial V9.1 book. |
| 2006-10-17 | Fixed a few cosmetic problems that were bugging me. |
| 2006-11-06 | Found out that IBM had removed the "UDB" from the Db2 product name, so I did the same. It is now just plain "Db2 V9". |
| 2006-11-29 | I goofed. Turns out Db2 is now called "Db2 9". I relabeled accordingly. |
| 2006-12-15 | Improved code to update or delete first "n" rows. |
| 2007-02-22 | Get unique timestamp values during multi-row insert. Other minor changes. |
| 2007-11-20 | Finished the Db2 V9.5 edition. Lots of changes! |
| 2008-09-20 | Fixed some minor problems. |
| 2008-11-28 | Fixed some minor problems. |
| 2009-01-18 | Fixed some minor problems, plus lots of bugs in Microsoft WORD! |
| 2009-03-12 | Converted to a new version of Adobe Acrobat, plus minor fixes. |
| 2010-10-12 | Finished initial V9.7 edition. Only minor changes. More to come. |
| 2010-11-05 | First batch of cute/deranged V9.7 SQL examples added. |

| Date Published (Version) | Content |
|---|---|
| 2010-11-14 | Fixed some minor typos. |
| 2011-01-11 | Added LIKE_COLUMN function. Removed bibliography. |
| 2011-01-14 | Added HASH function. Other minor edits. |
| 2011-08-16 | Fixed some minor problems. |

# 27.8. Book Binding

Below is a quick-and-dirty technique for making a book out of this book. The object of the exercise is to have a manual that will last a long time, and that will also lie flat when opened up. All suggested actions are done at your own risk.

## 27.8.1. Tools Required

- Printer, to print the book.
- KNIFE, to trim the tape used to bind the book.
- BINDER CLIPS, (1" size), to hold the pages together while gluing. To bind larger books, or to do multiple books in one go, use two or more cheap screw clamps.
- CARDBOARD: Two pieces of thick card, to also help hold things together while gluing.

## 27.8.2. Consumables

Ignoring the capital costs mentioned above, the cost of making a bound book should work out to about $4.00 per item, almost all of which is spent on the paper and toner. To bind an already printed copy should cost less than fifty cents.

- PAPER and TONER, to print the book.
- CARD STOCK, for the front and back covers.
- GLUE, to bind the book. Cheap rubber cement will do the job. The glue must come with an applicator brush in the bottle. Sears hardware stores sell a more potent flavor called Duro Contact Cement that is quite a bit better. This is toxic stuff, so be careful.
- CLOTH TAPE, (2" wide) to bind the spine. Pearl tape, available from Pearl stores, is fine. Wider tape will be required if you are not printing double-sided.
- TIME: With practice, this process takes less than five minutes work per book.

## 27.8.3. Before you Start

- Make that sure you have a well-ventilated space before gluing.
- Practice binding on some old scraps of paper.
- Kick all kiddies out off the room.

### 27.8.4. Instructions

- Print the book - double-sided if you can. If you want, print the first and last pages on card stock to make suitable protective covers.
- Jog the pages, so that they are all lined up along the inside spine. Make sure that every page is perfectly aligned, otherwise some pages won't bind. Put a piece of thick cardboard on either side of the set of pages to be bound. These will hold the pages tight during the gluing process.
- Place binder clips on the top and bottom edges of the book (near the spine), to hold everything in place while you glue. One can also put a couple on the outside edge to stop the pages from splaying out in the next step. If the pages tend to spread out in the middle of the spine, put one in the centre of the spine, then work around it when gluing. Make sure there are no gaps between leafs, where the glue might soak in.
- Place the book spine upwards. The objective here is to have a flat surface to apply the glue on. Lean the book against something if it does not stand up freely.
- Put on gobs of glue. Let it soak into the paper for a bit, then put on some more.
- Let the glue dry for at least half an hour. A couple of hours should be plenty.
- Remove the binder clips that are holding the book together. Be careful because the glue does not have much structural strength.
- Separate the cardboard that was put on either side of the book pages. To do this, carefully open the cardboard pages up (as if reading their inside covers), then run the knife down the glue between each board and the rest of the book.
- Lay the book flat with the front side facing up. Be careful here because the rubber cement is not very strong.
- Cut the tape to a length that is a little longer that the height of the book.
- Put the tape on the book, lining it up so that about one quarter of an inch (of the tape width) is on the front side of the book. Press the tape down firmly (on the front side only) so that it is properly attached to the cover. Make sure that a little bit of tape sticks out of both the bottom and top ends of the spine.
- Turn the book over (gently) and, from the rear side, wrap the cloth tape around the spine of the book. Pull the tape around so that it puts the spine under compression.
- Trim excess tape at either end of the spine using a knife or pair of scissors.
- Tap down the tape so that it is firmly attached to the book.
- Let the book dry for a day. Then do the old "hold by a single leaf" test. Pick any page, and gently pull the page up into the air. The book should follow without separating from the page.

### 27.8.5. More Information

The binding technique that I have described above is fast and easy, but rather crude. It would not be suitable if one was printing books for sale. There are plenty of other binding methods that take a little more skill and better gear that can be used to make "store-quality" books. Search the web for more information.

# References


- [sql.reference] Db2 11.1 for Linux, UNIX, and Windows: SQL. IBM Corp. 1994, 2017

# Index

**L**



**M**



**N**



**O**



**P**