DB2 UDB V8.1

SQL Cookbook

Graeme Birchall

4-Feb-2004

Preface

Important!

If you didn't get this document directly from my website, you may have got an older edition. The book gets changed all the time, so if you want the latest, go to the source. Also, the latest edition is usually the best book to have, even if you are using an older version of DB2, as the examples are often much better.

This Cookbook is for DB2 UDB for Windows, UNIX, LINX, OS/2, etc. It is not suitable for DB2 for z/OS or DB2 for AS/400. The SQL in these two products is quite different.

Disclaimer & Copyright

Tools Used

This book was written on a Dell PC that came with oodles or RAM. All testing was done on DB2 V8.1.4. Word for Windows was used to write the document. Adobe Acrobat was used to make the PDF file. As always, the book would have been written in half the time if Word for Windows wasn't such a bunch of bug-ridden junk.

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 page 341.

Author / Book

Author: Graeme Birchall ©
Address: 1 River Court, Apt 1706
Jersey City NJ 07310-2007
Ph/Fax: (201)-963-0071
Email: Graeme_Birchall@compuserve.com
Web: http://ourworld.compuserve.com/homepages/Graeme_Birchall


```
Title:   DB2 UDB V8.1 SQL Cookbook ©
Date:    4-Feb-2004
```

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 I have found new ways to solve problems. Hopefully, this process will continue for the foreseeable future.

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, then 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.

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 are 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.

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 idiot.

Graeme

Book Editions

Upload Dates

- 1996-05-08: First edition of the DB2 V2.1.1 SQL Cookbook was posted to my web site. This version was is 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 Cookbook.
- 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: IBM published two versions of the V5.2 upgrade. The initial edition, which I had used, evidently had a lot of problems. It was replaced within a week with a more complete upgrade. This book was based on the later 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.
- 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, plus there is a new section on temporary tables. The Identity Column and Join chapters were completely rewritten, and the Whine chapter was removed.
- 2003-01-02: DB2 V8.1 (post-Beta) edition. SQL rechecked. More examples added.
- 2003-07-11: New chapters added for temporary tables, compound SQL, and user defined functions. New DML section also added. Halting recursion section changed to use user-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.

Writing Software Whines

This book is written using Microsoft Word for Windows. I've been using this product for approximately ten years, and it has always been a bunch of bug-ridden junk. 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 unnecessary bugs in Word. So if somebody from Microsoft is reading this note, and if they feel committed to delivering decent software, kindly contact me.

Unfortunately, I'm probably going to be stuck with Word for a while. I've spent quite a bit of time looking at the alternatives and they are generally less productive, or have their own set of bugs, or are just wonderful, but cost too much and/or take too long to learn. Also unfortunately, I am now getting to the point where Word is so buggy that it is all but impossible to add new stuff to this document. Damn.

Table of Contents

Quick Find

This brief chapter is for those who want to find how to do something, but are not sure what the task is called. Hopefully, this list will identify the concept.

Index of Concepts

Join Rows

To combine matching rows in multiple tables, use a join (see page 177).

```
EMP_NM          EMP_JB         SELECT   nm.id              ANSWER
+----------+    +--------+             ,nm.name            ================
|ID|NAME    |   |ID|JOB   |            ,jb.job             ID NAME    JOB
|--|------- |   |--|----- |   FROM     emp_nm nm           -- ------- -----
|10|Sanders |   |10|Sales |            ,emp_jb jb          10 Sanders Sales
|20|Pernal  |   |20|Clerk |   WHERE    nm.id = jb.id       20 Pernal  Clerk
|50|Hanes   |   +--------+    ORDER BY 1;
+----------+
```

Figure 1, Join example

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 page 180).

```
EMP_NM          EMP_JB         SELECT   nm.id              ANSWER
+----------+    +--------+             ,nm.name            ================
|ID|NAME    |   |ID|JOB   |            ,jb.job             ID NAME    JOB
|--|------- |   |--|----- |   FROM     emp_nm nm           -- ------- -----
|10|Sanders |   |10|Sales |   LEFT OUTER JOIN              10 Sanders Sales
|20|Pernal  |   |20|Clerk |            emp_jb jb           20 Pernal  Clerk
|50|Hanes   |   +--------+    ON       nm.id = jb.id       50 Hanes   -
+----------+                  ORDER BY nm.id;
```

Figure 2,Left-outer-join example

To get rows from either side of the join, regardless of whether they match (the join) or not, use a full outer join (see page 184).

Null Values - Replace

Use the COALESCE function (see page 106) to replace a null value (e.g. generated in an outer join) with a non-null value.

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 page 199).

```
EMP_NM          EMP_JB         SELECT   *                          ANSWER
+----------+    +--------+     FROM     emp_nm nm                  ========
|ID|NAME    |   |ID|JOB   |   WHERE NOT EXISTS                    ID NAME
|--|------- |   |--|----- |           (SELECT *                    == =====
|10|Sanders |   |10|Sales |            FROM   emp_jb jb            50 Hanes
|20|Pernal  |   |20|Clerk |            WHERE  nm.id = jb.id)
|50|Hanes   |   +--------+    ORDER BY id;
+----------+
```

Figure 3, Sub-query example

Append Rows

To add (append) one set of rows to another set of rows, use a union (see page 213).

```
EMP_NM          EMP_JB          SELECT   *                         ANSWER
+----------+    +--------+      FROM     emp_nm                    ========
|ID|NAME    |   |ID|JOB   |     WHERE    name < 'S'                ID 2
|--|-------|    |--|-----|      UNION                              -- ------
|10|Sanders|    |10|Sales|      SELECT   *                         10 Sales
|20|Pernal |    |20|Clerk|      FROM     emp_jb                    20 Clerk
|50|Hanes  |    +--------+      ORDER BY 1,2;                      20 Pernal
+----------+                                                       50 Hanes
```

Figure 4, Union example

Assign Output Numbers

To assign line numbers to SQL output, use the ROW_NUMBER function (see page 84).

```
EMP_JB        SELECT   id
+--------+            ,job                                        ANSWER
|ID|JOB  |            ,ROW_NUMBER() OVER(ORDER BY job) AS R       =========
|--|-----|    FROM     emp_jb                                     ID JOB   R
|10|Sales|    ORDER BY job;                                       -- ----- -
|20|Clerk|                                                        20 Clerk 1
+--------+                                                        10 Sales 2
```

Figure 5, Assign row-numbers example

Assign Unique Key Numbers

The 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 page 229).

If-Then-Else Logic

To include if-then-else logical constructs in SQL stmts, use the CASE phrase (see page 37).

```
EMP_JB        SELECT   id                                   ANSWER
+--------+            ,job                                  ===============
|ID|JOB  |            ,CASE                                 ID JOB   STATUS
|--|-----|               WHEN job = 'Sales'                 -- ----- ------
|10|Sales|               THEN 'Fire'                        10 Sales Fire
|20|Clerk|               ELSE 'Demote'                      20 Clerk Demote
+--------+             END AS STATUS
              FROM     emp_jb;
```

Figure 6, Case stmt example

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 page 255).

```
FAMILY          WITH temp (persn, lvl) AS                     ANSWER
+-----------+     (SELECT  parnt, 1                           ========
|PARNT|CHILD|      FROM    family                             PERSN LVL
|-----|-----|      WHERE   parnt = 'Dad'                      ----- ---
|GrDad|Dad  |      UNION ALL                                  Dad     1
|Dad  |Dghtr|      SELECT  child, Lvl + 1                     Dghtr   2
|Dghtr|GrSon|      FROM    temp,                              GrSon   3
|Dghtr|GrDtr|              family                             GrDtr   3
+-----------+      WHERE   persn = parnt)
                SELECT *
                FROM   temp;
```

Figure 7, Recursion example

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 page 307).

```
INPUT DATA                 Recursive SQL                   ANSWER
=================          ============>                   ===========
"Some silly text"                                          TEXT  LINE#
                                                           ----- -----
                                                           Some      1
                                                           silly     2
                                                           text      3
```

Figure 8, Convert string to rows

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

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 page 308).

```
INPUT DATA                 Recursive SQL             ANSWER
===========                ============>             =================
TEXT  LINE#                                          "Some silly text"
----- -----
Some      1
silly     2
text      3
```

Figure 9, Convert rows to string

Fetch First "n" Rows

To fetch the first "n" matching rows, use the FETCH FIRST notation (see page 24).

```
EMP_NM         SELECT   *                                        ANSWER
+----------+   FROM     emp_nm                                   =========
|ID|NAME   |   ORDER BY id DESC                                  ID NAME
|--|-------|   FETCH FIRST 2 ROWS ONLY;                          -- ------
|10|Sanders|                                                     50 Hanes
|20|Pernal |                                                     20 Pernal
|50|Hanes  |
+----------+
```

Figure 10, Fetch first "n" rows example

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 page 85).

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 (see page 85).

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         SELECT   *                                       ANSWER
+----------+   FROM     emp_nm                                  ==========
|ID|NAME   |   WHERE    name like 'S%'                          ID NAME
|--|-------|   WITH UR;                                         -- -------
|10|Sanders|                                                    10 Sanders
|20|Pernal |
|50|Hanes  |
+----------+
```

Figure 11, Fetch WITH UR example

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

Summarize Column Contents

Use a column function (see page 67) to summarize the contents of a column.

```
EMP_NM         SELECT   AVG(id)   AS avg            ANSWER
+----------+           ,MAX(name) AS maxn           ================
|ID|NAME   |           ,COUNT(*)  AS #rows          AVG MAXN    #ROWS
|--|-------|   FROM     emp_nm;                     --- ------- -----
|10|Sanders|                                         26 Sanders     3
|20|Pernal |
|50|Hanes  |
+----------+
```

Figure 12, Column Functions example

Subtotals and Grand Totals

To obtain subtotals and grand-totals, use the ROLLUP or CUBE statements (see page 167).

```
SELECT   job                                 ANSWER
        ,dept                                =======================
        ,SUM(salary) AS sum_sal              JOB   DEPT SUM_SAL  #EMP
        ,COUNT(*)    AS #emps                ----- ---- -------- ----
FROM     staff                               Clerk   15 24766.70    2
WHERE    dept   < 30                         Clerk   20 27757.35    2
  AND    salary < 20000                      Clerk    - 52524.05    4
  AND    job    < 'S'                        Mgr     10 19260.25    1
GROUP BY ROLLUP(job, dept)                   Mgr     20 18357.50    1
ORDER BY job                                 Mgr      - 37617.75    2
        ,dept;                               -        - 90141.80    6
```

Figure 13, Subtotal and Grand-total example

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 page 229), 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 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 about the above.

Hide Complex SQL

One can create a view (see page 18) 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.

Summary Table

Some queries that use a GROUP BY can be made to run much faster by defining a summary table (see page 217) 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.

Introduction to SQL

This chapter contains a basic introduction to DB2 UDB 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.

Syntax Diagram Conventions

This book uses railroad diagrams to describe the DB2 UDB SQL statements. The following diagram shows the conventions used.

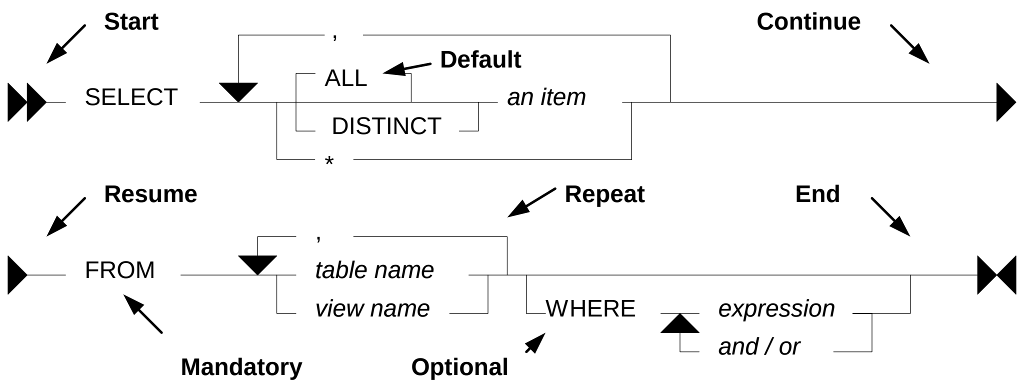

Figure 14, Syntax Diagram Conventions

Rules

- Upper Case text is a SQL keyword.
- Italic text is either a placeholder, or explained elsewhere.
- Backward arrows enable one to repeat parts of the text.
- A branch line going above the main line is the default.
- A branch line going below the main line is an optional item.

Statement Delimiter

DB2 SQL does not come with a designated statement delimiter (terminator), though a semi-colon is often used. A semi-colon cannot be used when writing a compound SQL statement (see page 57) because that character is used to terminate the various sub-components of the statement.

In DB2BATCH one can set the statement delimiter using an intelligent comment:

```
--#SET DELIMITER !

SELECT name FROM staff WHERE id = 10!

--#SET DELIMITER ;

SELECT name FROM staff WHERE id = 20;
```

Figure 15, Set Delimiter example

SQL Components

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.

```
CREATE TABLE employee
(empno      CHARACTER (00006)    NOT NULL
,firstnme   VARCHAR   (00012)    NOT NULL
,midinit    CHARACTER (00001)    NOT NULL
,lastname   VARCHAR   (00015)    NOT NULL
,workdept   CHARACTER (00003)
,phoneno    CHARACTER (00004)
,hiredate   DATE
,job        CHARACTER (00008)
,edlevel    SMALLINT             NOT NULL
,SEX        CHARACTER (00001)
,birthdate  DATE
,salary     DECIMAL   (00009,02)
,bonus      DECIMAL   (00009,02)
,comm       DECIMAL   (00009,02)
 )
 DATA CAPTURE NONE;
```

Figure 16, DB2 sample table - EMPLOYEE

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.

```
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);
```

Figure 17, DB2 sample view - EMPLOYEE_VIEW

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

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

Figure 18, Define a view using a VALUES clause

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

```
SELECT   c1, c2, c3                                  ANSWER
FROM     silly                                       ===========
ORDER BY c1 aSC;                                     C1  C2   C3
                                                     --  ---  --
                                                     11  AAA  22
                                                     12  BBB  33
                                                     13  CCC   -
```

Figure 19, SELECT from a view that has its own data

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.

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

Figure 20, Define a view that creates data on the fly

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.

```
CREATE ALIAS  employee_al1 FOR employee;
COMMIT;

CREATE ALIAS  employee_al2 fOR employee_al1;
COMMIT;

CREATE ALIAS  employee_al3 FOR employee_al2;
COMMIT;
```

Figure 21, Define three aliases, the latter on the earlier

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.

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).
- 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).

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

```
CREATE TABLE sales_record
(sales#              INTEGER             NOT NULL
                     GENERATED ALWAYS AS IDENTITY
                     (START   WITH 1
                     ,INCREMENT BY 1
                     ,NO MAXVALUE
                     ,NO CYCLE)
,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#));
```

Figure 22, Sample table definition

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 page 229 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:

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

Figure 23, Table with default column lengths

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.

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).

One creates a distinct (data) type using the following syntax:

```
▶— CREATE DISTINCT TYPE — type-name — source-type — WITH COMPARISONS —▶
```

Figure 24, Create Distinct Type Syntax

NOTE: The following source types do not support distinct types: LOB, LONG VARCHAR, LONG VARGRAPHIC, and DATALINK.

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. Support for the basic comparison operators (=, <>, <, <=, >, and >=) is also provided.

Below is a typical create and drop statement:

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

Figure 25, Create and drop distinct type

NOTE: 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:

```
CREATE TABLE customer
(id                  INTEGER                NOT NULL
,fname               VARCHAR(00010)         NOT NULL WITH DEFAULT ''
,lname               VARCHAR(00015)         NOT NULL WITH DEFAULT ''
,date_of_birth       DATE
,citizenship         CHAR(03)
,usa_sales           DECIMAL(9,2)
,eur_sales           DECIMAL(9,2)
,sales_office#       SMALLINT
,last_updated        TIMESTAMP
,PRIMARY KEY(id));
```

Figure 26, Sample table, without distinct types

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:

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

Figure 27, Silly query, but works

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

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

Figure 28, Create Distinct Type examples

Now we can define the customer table thus:

```
CREATE TABLE customer
(id                  INTEGER              NOT NULL
,fname               VARCHAR(00010)       NOT NULL WITH DEFAULT ''
,lname               VARCHAR(00015)       NOT NULL WITH DEFAULT ''
,date_of_birth       DATE
,citizenship         CHAR(03)
,usa_sales           USA_DOLLARS
,eur_sales           EUR_DOLLARS
,sales_office#       SMALLINT
,last_updated        TIMESTAMP
,PRIMARY KEY(id));
```

Figure 29, Sample table, with distinct types

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

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

Figure 30, Silly query, now fails

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:

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

Figure 31, Silly query, works again

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, or an alias.
- WITH clause. This clause is optional. Use this phrase to include independent SELECT statements that are subsequently accessed in a final SELECT (see page 246).
- ORDER BY clause. Optionally, order the final output (see page 159).
- FETCH FIRST clause. Optionally, stop the query after "n" rows (see page 24). 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.

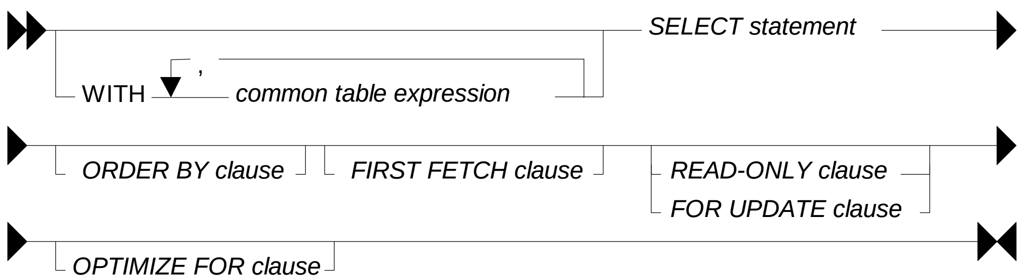

Figure 32, SELECT Statement Syntax (general)

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.

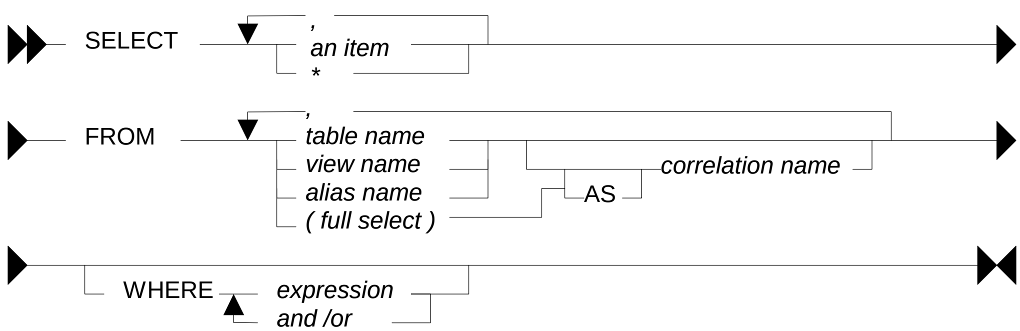

Figure 33, SELECT Statement Syntax

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.

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.

Sample SQL

```
SELECT   deptno                              ANSWER
        ,admrdept                            ===================
        ,'ABC' AS abc                        DEPTNO ADMRDEPT ABC
FROM     department                          ------ -------- ---
WHERE    deptname LIKE '%ING%'               B01    A00      ABC
ORDER BY 1;                                  D11    D01      ABC
```

Figure 34, Sample SELECT statement

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

```
SELECT   *                                                ANSWER (part of)
FROM     department                                       ================
WHERE    deptname LIKE '%ING%'                            DEPTNO etc...
ORDER BY 1;                                               ------ ------>>>
                                                          B01    PLANNING
                                                          D11    MANUFACTU
```

Figure 35, Use "" to select all columns in table*

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   deptno                                    ANSWER (part of)
        ,department.*                              =======================
FROM     department                                DEPTNO DEPTNO etc...
WHERE    deptname LIKE '%ING%'                     ------ ------ ------>>>
ORDER BY 1;                                        B01    B01    PLANNING
                                                   D11    D11    MANUFACTU
```

Figure 36, Select an individual column, and all columns

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

```
SELECT   department.*                                     ANSWER (part of)
        ,department.*                                     ================
FROM     department                                       DEPTNO etc...
WHERE    deptname LIKE '%NING%'                           ------ ------>>>
ORDER BY 1;                                               B01    PLANNING
```

Figure 37, Select all columns twice

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.

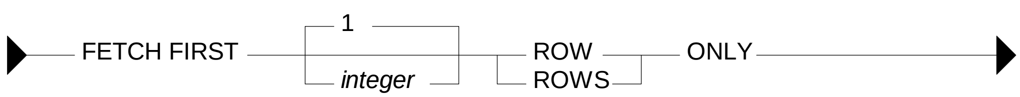

Figure 38, Fetch First clause Syntax

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:

```
SELECT   years                                      ANSWER
        ,name                                       =====================
        ,id                                         YEARS  NAME      ID
FROM     staff                                      ------ --------- ----
FETCH FIRST 3 ROWS ONLY;                                 7 Sanders     10
                                                         8 Pernal      20
                                                         5 Marenghi    30
```

Figure 39, FETCH FIRST without ORDER BY, gets random rows

WARNING: 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:

```
SELECT   years                                        ANSWER
        ,name                                         =====================
        ,id                                           YEARS  NAME      ID
FROM     staff                                        ------ --------- ----
WHERE    years IS NOT NULL                                13 Graham     310
ORDER BY years DESC                                       12 Jones      260
FETCH FIRST 3 ROWS ONLY;                                  10 Hanes       50
```

Figure 40, FETCH FIRST with ORDER BY, gets wrong answer

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):

```
SELECT   years                                        ANSWER
        ,name                                         =====================
        ,id                                           YEARS  NAME      ID
FROM     staff                                        ------ --------- ----
WHERE    years IS NOT NULL                                13 Graham     310
ORDER BY years DESC                                       12 Jones      260
        ,id    DESC                                       10 Quill      290
FETCH FIRST 3 ROWS ONLY;
```

Figure 41, FETCH FIRST with ORDER BY, gets right answer

WARNING: Getting the first "n" rows from a query is actually quite a complicated problem. Refer to page 87 for a more complete discussion.

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:

```
SELECT   a.empno                                          ANSWER
        ,a.lastname                                       =================
FROM     employee  a                                      EMPNO  LASTNAME
        ,(SELECT MAX(empno)AS empno                       ------ ----------
          FROM   employee) AS b                           000340 GOUNOT
WHERE    a.empno = b.empno;
```

Figure 42, Correlation Name usage example

```
SELECT   a.empno                                     ANSWER
        ,a.lastname                                  ======================
        ,b.deptno AS dept                            EMPNO  LASTNAME   DEPT
FROM     employee   a                                ------ ---------- ----
        ,department b                                000090 HENDERSON  E11
WHERE    a.workdept  = b.deptno                      000280 SCHNEIDER  E11
  AND    a.job      <> 'SALESREP'                    000290 PARKER     E11
  AND    b.deptname  = 'OPERATIONS'                  000300 SMITH      E11
  AND    a.sex      IN ('M','F')                     000310 SETRIGHT   E11
  AND    b.location IS NULL
ORDER BY 1;
```

Figure 43, Correlation name usage example

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:

```
SELECT   empno    AS  e_num                          ANSWER
        ,midinit  AS "m int"                         ===================
        ,phoneno  AS "..."                           E_NUM   M INT  ...
FROM     employee                                    ------  -----  ----
WHERE    empno < '000030'                            000010  I      3978
ORDER BY 1;                                          000020  L      3476
```

Figure 44, Renaming fields using AS

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 full-select of nested table expressions, common table expressions, and in a view definition.

```
CREATE view emp2 AS
SELECT empno    AS  e_num
      ,midinit  AS "m int"
      ,phoneno  AS "..."
FROM   employee;                                     ANSWER
                                                     ===================
SELECT *                                             E_NUM   M INT  ...
FROM   emp2                                          ------  -----  ----
WHERE "..." = '3978';                                000010  I      3978
```

Figure 45, View field names defined using AS

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:

```
SELECT   AVG(comm)              AS a1                    ANSWER
        ,SUM(comm) / COUNT(*)   AS a2                    ===============
FROM     staff                                           A1       A2
WHERE    id < 100;                                       -------  ------
                                                         796.025  530.68
```

Figure 46, AVG of data containing null values

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:

```
SELECT   COUNT(*)       AS num                                  ANSWER
        ,MAX(lastname)  AS max                                  ========
FROM     employee                                               NUM  MAX
WHERE    firstnme = 'FRED';                                     ---  ---
                                                                  0  -
```

Figure 47, Getting a NULL value from a field defined NOT NULL

Why Nulls 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.

```
SELECT   AVG(comm)              AS a1                   ANSWER
        ,SUM(comm) / COUNT(*)   AS a2                   ===============
FROM     staff                                          A1       A2
WHERE    id < 100                                       -------  ------
  AND    comm IS NOT NULL;                              796.025  796.02
```

Figure 48, AVG of those rows that are not null

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:

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

Figure 49, Quote usage

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

```
SELECT   id       AS "USER ID"          ANSWER
        ,dept     AS "D#"               ===============================
        ,years    AS "#Y"               USER ID D# #Y 'TXT' "quote" fld
        ,'ABC'    AS "'TXT'"            ------- -- -- ----- -----------
        ,'"'      AS """quote"" fld"         10 20  7 ABC   "
FROM     staff s                             20 20  8 ABC   "
WHERE    id < 40                             30 38  5 ABC   "
ORDER BY "USER ID";
```

Figure 50, Double-quote usage

> NOTE: Nonstandard column names (i.e. with double quotes) cannot be used in tables, but they are permitted in view definitions.

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.

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.

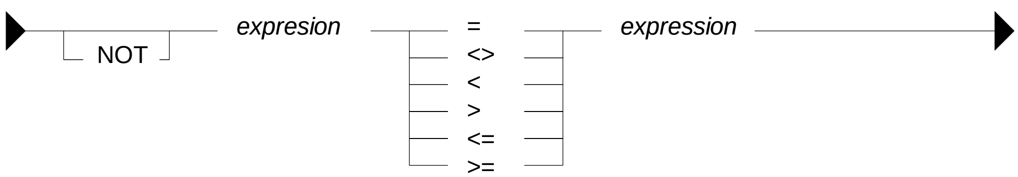

Figure 51, Basic Predicate syntax

```
SELECT     id, job, dept                                      ANSWER
FROM       staff                                              ===============
WHERE      job  =  'Mgr'                                      ID  JOB   DEPT
  AND NOT job  <> 'Mgr'                                       --- ----  ----
  AND NOT job  =  'Sales'                                      10 Mgr     20
  AND      id   <>  100                                        30 Mgr     38
  AND      id   >=    0                                        50 Mgr     15
  AND      id   <=  150                                       140 Mgr     51
  AND NOT dept =    50
ORDER BY  id;
```

Figure 52, Basic Predicate examples

Quantified Predicate

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

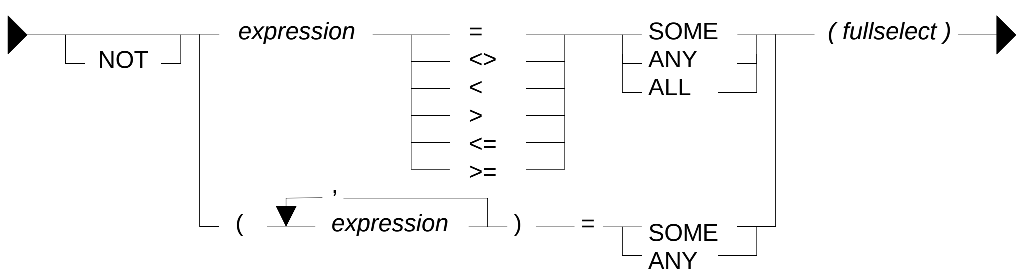

Figure 53, Quantified Predicate syntax,1 of 2

```
SELECT    id, job                                    ANSWER
FROM      staff                                      ========
WHERE     job  = ANY (SELECT job FROM staff)         ID  JOB
  AND     id  <= ALL (SELECT id  FROM staff)         --- ----
ORDER BY id;                                          10 Mgr
```

Figure 54, Quantified Predicate example, two single-value sub-queries

```
SELECT    id, dept, job                               ANSWER
FROM      staff                                       ==============
WHERE     (id,dept) = ANY                             ID  DEPT JOB
          (SELECT dept, id                            --- ---- -----
           FROM   staff)                               20   20 Sales
ORDER BY 1;
```

Figure 55, Quantified Predicate example, multi-value sub-query

See the sub-query chapter on page 199 for more data on this predicate type.

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

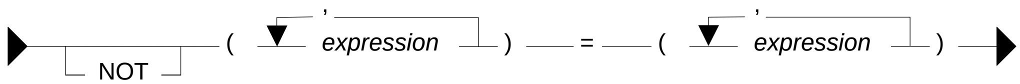

Figure 56, Quantified Predicate syntax, 2 of 2

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

Figure 57, Quantified Predicate example, multi-value check

Below is the same query written the old fashioned way:

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

Figure 58, Same query as prior, using individual predicates

BETWEEN Predicate

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

Figure 59, BETWEEN Predicate syntax

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.

```
SELECT id, job                                              ANSWER
FROM   staff                                                =========
WHERE      id     BETWEEN 10 AND 30                         ID  JOB
  AND      id NOT BETWEEN 30 AND 10                         --- -----
  AND NOT id NOT BETWEEN 10 AND 30                           10 Mgr
ORDER BY  id;                                                20 Sales
                                                             30 Mgr
```

Figure 60, BETWEEN Predicate examples

EXISTS Predicate

An EXISTS predicate tests for the existence of matching rows.

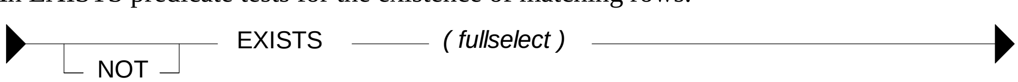

Figure 61, EXISTS Predicate syntax

```
SELECT id, job                                              ANSWER
FROM   staff a                                              =========
WHERE  EXISTS                                               ID  JOB
      (SELECT *                                             --- -----
       FROM   staff b                                        10 Mgr
       WHERE  b.id = a.id                                    20 Sales
         AND  b.id < 50)                                     30 Mgr
ORDER BY id;                                                 40 Sales
```

Figure 62, EXISTS Predicate example

NOTE: See the sub-query chapter on page 199 for more data on this predicate type.

IN Predicate

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

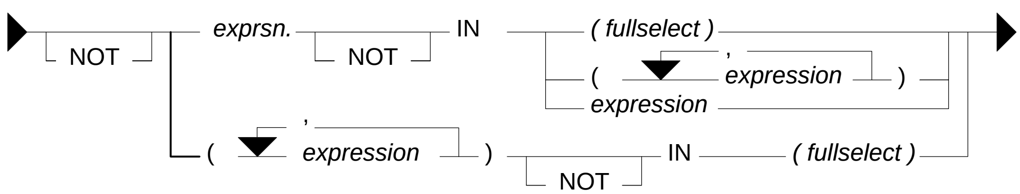

Figure 63, IN Predicate syntax

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.

```
SELECT id, job                                              ANSWER
FROM   staff a                                              =========
WHERE  id IN (10,20,30)                                     ID  JOB
  AND  id IN (SELECT id                                     --- -----
              FROM   staff)                                  10 Mgr
  AND  id NOT IN 99                                          20 Sales
ORDER BY id;                                                 30 Mgr
```

Figure 64, IN Predicate examples, single values

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.

WARNING: 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 page 199 for more details.

```
SELECT   empno, lastname                              ANSWER
FROM     employee                                     ===============
WHERE    (empno, 'AD3113') IN                         EMPNO  LASTNAME
         (SELECT empno, projno                        ------ -------
          FROM   emp_act                              000260 JOHNSON
          WHERE  emptime > 0.5)                       000270 PEREZ
ORDER BY 1;
```

Figure 65, IN Predicate example, multi-value

NOTE: See the sub-query chapter on page 199 for more data on this statement type.

LIKE Predicate

The LIKE predicate does partial checks on character strings.

Figure 66, LIKE Predicate syntax

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'.

```
SELECT id, name                                            ANSWER
FROM   staff                                               ==============
WHERE  name LIKE 'S%n'                                     ID   NAME
   OR  name LIKE '_a_a%'                                   ---  ---------
   OR  name LIKE '%r_%a'                                   130  Yamaguchi
ORDER BY id;                                               200  Scoutten
```

Figure 67, LIKE Predicate examples

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.

```
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
```

Figure 68, LIKE and ESCAPE examples

Now for sample SQL:

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

Figure 69, LIKE and ESCAPE examples

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.

Figure 70, NULL Predicate syntax

```
SELECT    id, comm                                              ANSWER
FROM      staff                                                 =========
WHERE     id    < 100                                           ID   COMM
  AND     id   IS NOT NULL                                      ---  ----
  AND     comm IS     NULL                                       10  -
  AND NOT comm IS NOT NULL                                       30  -
ORDER BY id;                                                     50  -
```

Figure 71, NULL predicate examples

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

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.

```
Example:       555 +    -22  /  (12 - 3) * 66                 ANSWER
                                                              ======
                   ^    ^    ^      ^    ^                       423
                  5th  2nd  3rd    1st  4th
```

Figure 72, Precedence rules example

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:

```
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
```

Figure 73, Precedence rules, integer example

NOTE: DB2 truncates, not rounds, when doing integer arithmetic.

Here is the same done using decimal numbers:

```
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
```

Figure 74, Precedence rules, decimal example

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:

```
SELECT   *                               ANSWER>>   COL1 COL2    TABLE1
FROM     table1                                     ---- ----    +---------+
WHERE    col1  = 'C'                                A    AA      |COL1|COL2|
  AND    col1 >= 'A'                                B    BB      |----|----|
   OR    col2 >= 'AA'                               C    CC      |A   |AA  |
ORDER BY col1;                                                   |B   |BB  |
                                                                 |C   |CC  |
SELECT   *                               ANSWER>>   COL1 COL2    +---------+
FROM     table1                                     ---- ----
WHERE   (col1  = 'C'                                A    AA
  AND    col1 >= 'A')                               B    BB
   OR    col2 >= 'AA'                               C    CC
ORDER BY col1;

SELECT   *                               ANSWER>>   COL1 COL2
FROM     table1                                     ---- ----
WHERE    col1  = 'C'                                C    CC
  AND   (col1 >= 'A'
   OR    col2 >= 'AA')
ORDER BY col1;
```

Figure 75, Use of OR and parenthesis

WARNING: 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.

CAST Expression

The CAST is 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.

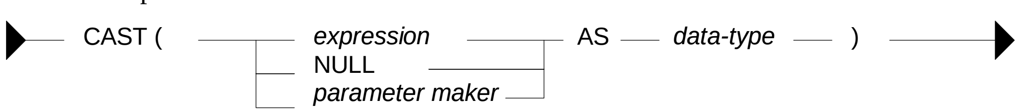

Figure 76, CAST expression syntax

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:

```
SELECT   id                                         ANSWER
        ,salary                                     =================
        ,CAST(salary AS INTEGER) AS sal2            ID SALARY   SAL2
FROM     staff                                      -- -------- -----
WHERE    id < 30                                    10 18357.50 18357
ORDER BY id;                                        20 18171.25 18171
```

Figure 77, Use CAST expression to convert Decimal to Integer

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.

```
SELECT   id                                                ANSWER
        ,job                                               =============
        ,CAST(job AS CHAR(3)) AS job2                      ID JOB   JOB2
FROM     staff                                             -- ----- ----
WHERE    id < 30                                           10 Mgr   Mgr
ORDER BY id;                                               20 Sales Sal
```

Figure 78, Use CAST expression to truncate Char field

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

```
SELECT   id                                                      ANSWER
        ,CAST(NULL AS SMALLINT) AS junk                          =======
FROM     staff                                                   ID JUNK
WHERE    id < 30                                                 -- ----
ORDER BY id;                                                     10    -
                                                                 20    -
```

Figure 79, Use CAST expression to define SMALLINT field with null values

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

```
SELECT   stf.id                                                ANSWER
        ,emp.empno                                             =========
FROM     staff    stf                                          ID EMPNO
LEFT OUTER JOIN                                                -- ------
         employee emp                                          10 -
ON       stf.id   =  CAST(emp.empno AS SMALLINT)               20 000020
AND      emp.job  = 'MANAGER'                                  30 000030
WHERE    stf.id   <  60                                        40 -
ORDER BY stf.id;                                               50 000050
```

Figure 80, CAST expression in join

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

```
SELECT   stf.id                                                ANSWER
        ,emp.empno                                             =========
FROM     staff    stf                                          ID EMPNO
LEFT OUTER JOIN                                                -- ------
         employee emp                                          10 -
ON       stf.id   =  SMALLINT(emp.empno)                       20 000020
AND      emp.job  = 'MANAGER'                                  30 000030
WHERE    stf.id   <  60                                        40 -
ORDER BY stf.id;                                               50 000050
```

Figure 81, Function usage in join

VALUES Clause

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.

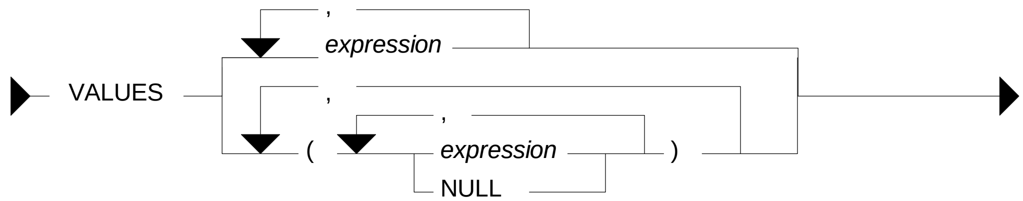

Figure 82, VALUES expression syntax

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   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 columns
```

Figure 83, VALUES usage examples

Sample SQL

The next statement shall define a temporary table containing two columns and three rows. The first column will default to type integer and the second to type varchar.

```
WITH temp1 (col1, col2) AS                                        ANSWER
(VALUES    (   0, 'AA')                                           =========
          ,(   1, 'BB')                                           COL1 COL2
          ,(   2, NULL)                                           ---- ----
)                                                                    0 AA
SELECT *                                                             1 BB
FROM   temp1;                                                        2 -
```

Figure 84, Use VALUES to define a temporary table (1 of 4)

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.

```
WITH temp1 (col1, col2) AS                                        ANSWER
(VALUES    (DECIMAL(0 ,3,1), 'AA')                                =========
          ,(DECIMAL(1 ,3,1), 'BB')                                COL1 COL2
          ,(DECIMAL(2 ,3,1), NULL)                                ---- ----
)                                                                  0.0 AA
SELECT *                                                           1.0 BB
FROM   temp1;                                                      2.0 -
```

Figure 85, Use VALUES to define a temporary table (2 of 4)

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:

```
WITH temp1 (col1, col2) AS                                        ANSWER
(VALUES    (   0, CAST('AA' AS CHAR(1)))                          =========
          ,(   1, CAST('BB' AS CHAR(1)))                          COL1 COL2
          ,(   2, CAST(NULL AS CHAR(1)))                          ---- ----
)                                                                    0 A
SELECT *                                                             1 B
FROM   temp1;                                                        2 -
```

Figure 86, Use VALUES to define a temporary table (3 of 4)

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:

```
WITH temp1 (col1, col2) AS                                    ANSWER
(VALUES    (   0, CHAR('AA',1))                               =========
          ,(   1, CHAR('BB',1))                               COL1 COL2
          ,(   2, NULL)                                       ---- ----
)                                                                0 A
SELECT *                                                         1 B
FROM   temp1;                                                    2 -
```
Figure 87, Use VALUES to define a temporary table (4 of 4)

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.

```
WITH temp1 (col1, col2, col3) AS                             ANSWER
(VALUES    (   0, 'AA', 0.00)                                ==========
          ,(   1, 'BB', 1.11)                                COL1B COLX
          ,(   2, 'CC', 2.22)                                ----- ----
)                                                                0 0.00
,temp2 (col1b, colx) AS                                          1 2.11
(SELECT  col1                                                    2 4.22
        ,col1 + col3
 FROM    temp1
)
SELECT *
FROM   temp2;
```
Figure 88, Derive one temporary table from another

```
CREATE VIEW silly (c1, c2, c3)
AS VALUES (11, 'AAA', SMALLINT(22))
         ,(12, 'BBB', SMALLINT(33))
         ,(13, 'CCC', NULL);
COMMIT;
```
Figure 89, Define a view using a VALUES clause

```
WITH temp1 (col1) AS                                            ANSWER
(VALUES     0                                                   ======
 UNION ALL                                                      COL1
 SELECT col1 + 1                                                ----
 FROM   temp1                                                      0
 WHERE  col1 + 1 < 100                                             1
)                                                                  2
SELECT *                                                           3
FROM   temp1;                                                    etc
```
Figure 90, Use VALUES defined data to seed a recursive SQL statement

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:

```
SELECT   *                                                    ANSWER
FROM    (VALUES (123,'ABC')                                   ======
               ,(234,'DEF')                                   --- ---
        )AS ttt                                               234 DEF
ORDER BY 1 DESC;                                              123 ABC
```
Figure 91, Generate table with unnamed columns

CASE Expression

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

CASE expressions enable one to do if-then-else type processing inside of SQL statements. There are two general flavors of the expression. In the first kind, each WHEN statement does its own independent checking. In the second kind, all of the WHEN conditions are used to do "equal" checks against a common reference expression. With both flavors, the first WHEN that matches is the one chosen.

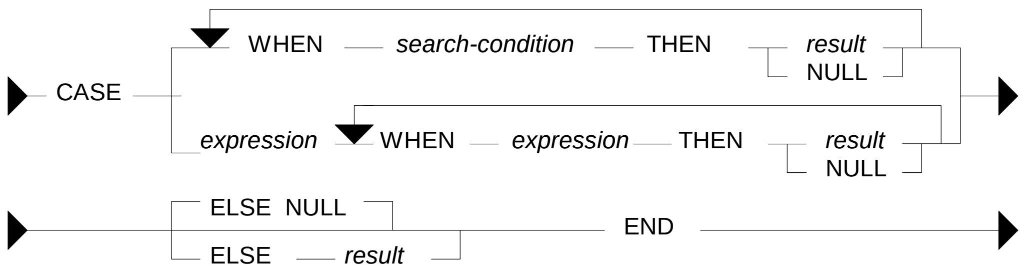

Figure 92, CASE expression syntax

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.

CASE Flavours

The following CASE is of the kind where each WHEN does an equal check against a common expression - in this example, the current value of SEX.

```
SELECT   Lastname                                      ANSWER
        ,sex   AS sx                                   ====================
        ,CASE  sex                                     LASTNAME   SX SEXX
           WHEN 'F' THEN 'FEMALE'                      ---------- -- ------
           WHEN 'M' THEN 'MALE'                        JEFFERSON  M  MALE
           ELSE NULL                                   JOHNSON    F  FEMALE
         END AS sexx                                   JONES      M  MALE
FROM     employee
WHERE    lastname LIKE 'J%'
ORDER BY 1;
```

Figure 93, Use CASE (type 1) to expand a value

The next statement is logically the same as the above, but it uses the alternative form of the CASE notation in order to achieve the same result. In this example, the equal predicate is explicitly stated rather than implied.

```
SELECT   lastname                                   ANSWER
        ,sex   AS sx                                ====================
        ,CASE                                       LASTNAME   SX SEXX
           WHEN sex = 'F' THEN 'FEMALE'             ---------- -- ------
           WHEN sex = 'M' THEN 'MALE'               JEFFERSON  M  MALE
           ELSE NULL                                JOHNSON    F  FEMALE
         END AS sexx                                JONES      M  MALE
FROM     employee
WHERE    lastname LIKE 'J%'
ORDER BY 1;
```

Figure 94, Use CASE (type 2) to expand a value

More Sample SQL

```
SELECT   lastname                                    ANSWER
        ,midinit AS mi                               ===================
        ,sex     AS sx                               LASTNAME   MI SX MX
        ,CASE                                        ---------- -- -- --
           WHEN midinit > SEX                        JEFFERSON  J  M  M
           THEN midinit                              JOHNSON    P  F  P
           ELSE sex                                  JONES      T  M  T
         END AS mx
FROM     employee
WHERE    lastname LIKE 'J%'
ORDER BY 1;
```

Figure 95, Use CASE to display the higher of two values

```
SELECT   COUNT(*)                                    AS tot    ANSWER
        ,SUM(CASE sex WHEN 'F' THEN 1 ELSE 0  END) AS #f      =========
        ,SUM(CASE sex WHEN 'M' THEN 1 ELSE 0  END) AS #m      TOT #F #M
FROM     employee                                              --- -- --
WHERE    lastname LIKE 'J%';                                     3  1  2
```

Figure 96, Use CASE to get multiple counts in one pass

```
SELECT   lastname                                         ANSWER
        ,sex                                              ==============
FROM     employee                                         LASTNAME   SEX
WHERE    lastname LIKE 'J%'                               ---------- ---
  AND    CASE  sex                                        JEFFERSON  M
           WHEN 'F' THEN ''                               JOHNSON    F
           WHEN 'M' THEN ''                               JONES      M
           ELSE NULL
         END IS NOT NULL
ORDER BY 1;
```

Figure 97, Use CASE in a predicate

```
SELECT   lastname                                  ANSWER
        ,LENGTH(RTRIM(lastname)) AS len            =====================
        ,SUBSTR(lastname,1,                        LASTNAME   LEN LASTNM
           CASE                                    ---------- --- ------
            WHEN LENGTH(RTRIM(lastname))           JEFFERSON    9 JEFFER
                 > 6 THEN 6                        JOHNSON      7 JOHNSO
            ELSE LENGTH(RTRIM(lastname))           JONES        5 JONES
           END  ) AS lastnm
FROM     employee
WHERE    lastname LIKE 'J%'
ORDER BY 1;
```

Figure 98, Use CASE inside a function

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 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;
```

Figure 99, UPDATE statement with nested CASE expressions

```
WITH temp1 (c1,c2) AS                                        ANSWER
(VALUES    (88,9),(44,3),(22,0),(0,1))                       ========
SELECT c1                                                    C1 C2 C3
      ,c2                                                    -- -- --
      ,CASE c2                                               88  9  9
         WHEN 0 THEN NULL                                    44  3 14
         ELSE c1/c2                                          22  0  -
       END AS c3                                              0  1  0
FROM   temp1;
```

Figure 100, Use CASE to avoid divide by zero

At least one of the results in a CASE expression must be non-null. This is so that DB2 will know what output type to make the result. One can get around this restriction by using the CAST expression. It is hard to imagine why one might want to do this, but it works:

```
SELECT name                                              ANSWER
      ,CASE                                              ============
         WHEN name = LCASE(name) THEN NULL               NAME    DUMB
         ELSE CAST(NULL AS CHAR(1))                      ------- ----
       END AS dumb                                       Sanders    -
FROM   staff                                             Pernal     -
WHERE  id < 30;
```

Figure 101, Silly CASE expression that always returns NULL

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.

```
SELECT   lastname                                   ANSWER
        ,sex                                        =================
        ,CASE                                       LASTNAME   SX SXX
           WHEN sex >= 'M' THEN 'MAL'               ---------- -- ---
           WHEN sex >= 'F' THEN 'FEM'               JEFFERSON  M  MAL
         END AS sxx                                 JOHNSON    F  FEM
FROM     employee                                   JONES      M  MAL
WHERE    lastname LIKE 'J%'
ORDER BY 1;
```

Figure 102, Use CASE to derive a value (correct)

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.

```
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
```

Figure 103, Use CASE to derive a value (incorrect)

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.

DML (Data Manipulation Language)

The section has a very basic introduction to the INSERT, UPDATE, DELETE, and MERGE statements. See the DB2 manuals for more details.

Select DML Changes

A special kind of SELECT statement (see page 47) 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).

Insert

The INSERT statement is used to insert rows into a table, view, or full-select. To illustrate how it is used, this section will use the EMP_ACT sample table, which is defined thus:

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

Figure 104, EMP_ACT sample table - DDL

Insert Syntax

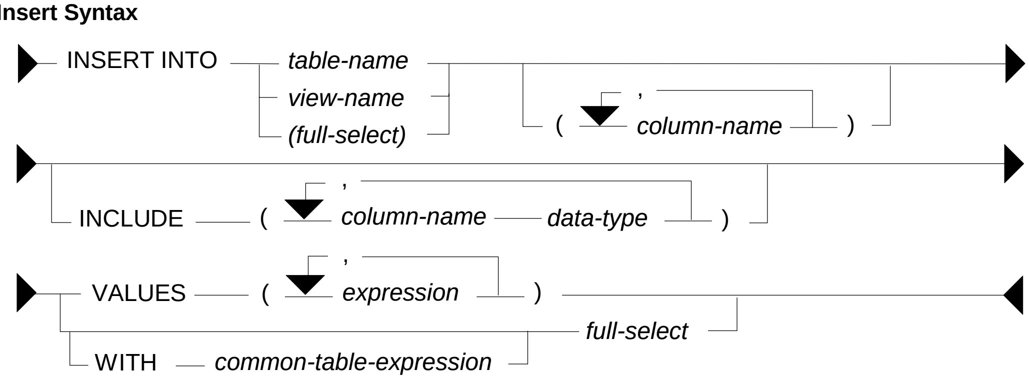

Figure 105, INSERT statement syntax

Usage Notes

- One can insert into a table, view, or full-select. If the object is not a table, then it must be insertable (i.e. refer to a single table, not have any column functions, etc).
- 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 page 47).
- The input data can either be explicitly defined using the VALUES statement, or retrieved from some other table using a full-select.

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:

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

Figure 106, Single row insert

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

```
INSERT INTO emp_act 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');
```

Figure 107, Multi row insert

> NOTE: 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:

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

Figure 108,Using null and default values

To leave some columns out of the insert statement, one has to explicitly list those columns that are included. When this is done, one can refer to the columns (being inserted with data) in any order:

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

Figure 109, Explicitly listing columns being populated during insert

Insert into Full-Select

The next statement inserts a row into a full-select 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
   (SELECT *
    FROM   emp_act
    WHERE  empno < '1'
   )
VALUES ('510000' ,'ABC' ,10 ,1.4 ,'2003-10-22', '2003-11-24');
```

Figure 110, Insert into a full-select

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.

Insert from Select

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

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

Figure 111,Insert result of select statement

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

Figure 112, Insert result of select - specified columns only

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:

```
INSERT INTO emp_act
SELECT *
FROM   emp_act;
```

Figure 113, Stupid - insert - doubles rows

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

```
INSERT INTO emp_act (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;
```

Figure 114, Inserting result of union

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 from during the insert:

```
INSERT INTO emp_act (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;
```
Figure 115, Insert from common table expression

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 INTO emp_act (empno, actno, projno)
SELECT LTRIM(CHAR(id + 920000))
      ,id
      ,'ABC'
FROM   staff
WHERE  id < 40
  AND  NOT EXISTS
      (SELECT *
       FROM   emp_act
       WHERE  empno LIKE '92%');
```
Figure 116, Insert with irrelevant sub-query

Update

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

```
UPDATE   emp_act
SET      emptime   =  NULL
        ,emendate =  DEFAULT
        ,emstdate =  CURRENT DATE + 2 DAYS
        ,actno    =  ACTNO / 2
        ,projno   =  'ABC'
WHERE    empno    = '100000';
```
Figure 117, Single row update

Update Syntax

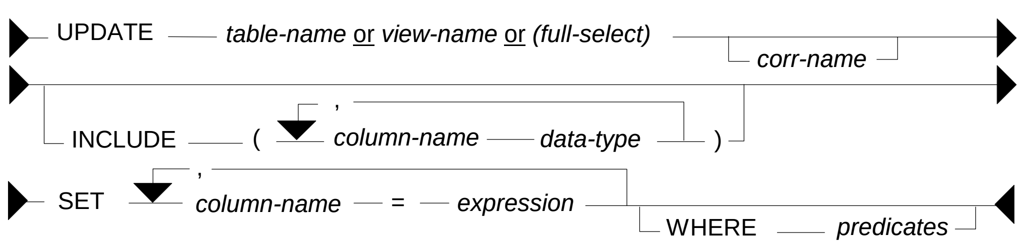

Figure 118, UPDATE statement syntax

Usage Notes

- One can update rows in a table, view, or full-select. 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 page 47).

- The SET statement lists the columns to be updated, and the new value they will get.
- Predicates are optional. If none are provided, all rows in the table are updated.

Update Examples

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

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

Figure 119, Mass update

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:

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

Figure 120, Two columns get same value

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  emp_act
SET     actno    = (SELECT MAX(salary)
                    FROM   staff)
WHERE   empno    = '200000';
```

Figure 121, Update using select

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

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

Figure 122, Multi-row update using select

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.

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

Figure 123, Multi-row update using correlated select

Using Full-selects

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

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

Figure 124, Direct update of table

Below is a logically equivalent update that pushes the predicates up into a full-select:

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

Figure 125, Update of full-select

Using OLAP Functions

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:

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

Figure 126, Set employee-time in row to MAX - for given employee

The same result can be achieved by calling an OLAP function in a full-select, 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:

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

Figure 127, Use OLAP function to get max-time, then apply (correct)

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:

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

Figure 128, Use OLAP function to get max-time, then apply (wrong)

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

Figure 129, Update with correlated query

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

Figure 130, Update with uncorrelated query

Delete

The DELETE statement is used to remove rows from a table , view, or full-select. 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:

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

Figure 131, Single-row delete

Delete Syntax

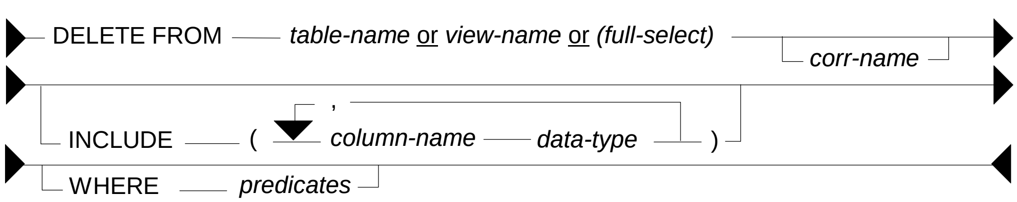

Figure 132, DELETE statement syntax

Usage Notes

- One can delete rows from a table, view, or full-select. 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 page 47).

- Predicates are optional. If none are provided, all rows are deleted.

Basic Delete

The next example would delete all rows in the EMP_ACT table:

```
DELETE
FROM    emp_act;
```

Figure 133, Mass delete

Correlated Delete

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

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

Figure 134, Correlated delete (1 of 2)

Here is another way to write the same:

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

Figure 135, Correlated delete (2 of 2)

The next query is logically equivalent to the prior two, but it works quite differently. It uses a full-select 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 FROM
  (SELECT  id
          ,MAX(id) OVER(PARTITION BY dept) AS max_id
   FROM    staff
  )AS ss
WHERE id <> max_id;
```

Figure 136, Delete using full-select and OLAP function

Select DML Changes

One often needs to know what data a particular insert, update, or delete statement changed. For example, one may need to get the key (e.g. invoice number) that was generated on the fly (using an identity column - see page 229) during an insert, 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.

Select DML Syntax

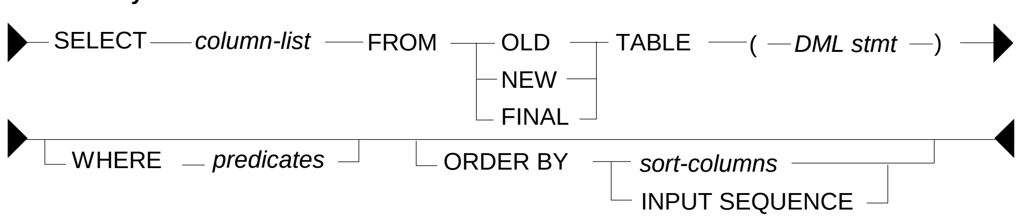

Figure 137, Select DML statement syntax

Table Types

- OLD: Has the before state of the data. This is allowed for an update and delete.
- NEW: Has the after state of the data - before any triggers are applied. This is allowed for an insert and an update.
- FINAL: Has the final state of the data - after all triggers have been applied. 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 page 51), 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.

Insert Examples

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

```
                                                         ANSWER
                                                         ==============
SELECT   empno                                           EMPNO  PRJ ACT
        ,projno AS prj                                   ------ --- ---
        ,actno  AS act                                   200000 ABC  10
FROM     FINAL TABLE                                     200000 DEF  10
   (INSERT INTO emp_act
    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;
```

Figure 138, Select rows inserted

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:

```
SELECT   empno                                            ANSWER
        ,projno AS prj                                    =================
        ,actno  AS act                                    EMPNO  PRJ ACT R#
        ,row#   AS r#                                     ------ --- --- --
FROM     FINAL TABLE                                      300000 ZZZ 999  1
   (INSERT INTO emp_act (empno, projno, actno)            300000 VVV 111  2
    INCLUDE (row# SMALLINT)
    VALUES ('300000','ZZZ',999,1)
          ,('300000','VVV',111,2))
ORDER BY row#;
```

Figure 139, Include column to get insert sequence

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   empno                                            ANSWER
        ,projno AS prj                                    =================
        ,actno  AS act                                    EMPNO  PRJ ACT R#
        ,ROW_NUMBER() OVER() AS r#                        ------ --- --- --
FROM     FINAL TABLE                                      400000 ZZZ 999  1
   (INSERT INTO emp_act (empno, projno, actno)            400000 VVV 111  2
    VALUES ('400000','ZZZ',999)
          ,('400000','VVV',111))
ORDER BY INPUT SEQUENCE;
```

Figure 140, Select rows in insert order

NOTE: 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   empno                                            ANSWER
        ,projno AS prj                                    =================
        ,actno  AS act                                    EMPNO  PRJ ACT R#
        ,ROW_NUMBER() OVER() AS r#                        ------ ---  -- --
FROM     NEW TABLE                                        600010 1    59  1
   (INSERT INTO emp_act (empno, actno, projno)            600020 563  59  2
    SELECT  LTRIM(CHAR(id + 600000))                      600030 193  59  3
           ,SECOND(CURRENT TIME)
           ,CHAR(SMALLINT(RAND(1) * 1000))
    FROM    staff
    WHERE   id < 40)
ORDER BY INPUT SEQUENCE;
```

Figure 141, Select from an insert that has unknown values

Update Examples

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

```
SELECT   empno                                            ANSWER
        ,projno  AS prj                                   ================
        ,emptime AS etime                                 EMPNO  PRJ ETIME
FROM     OLD TABLE                                        ------ --- -----
   (UPDATE emp_act                                        200000 ABC  1.00
    SET    emptime = emptime * 2                          200000 DEF  1.00
    WHERE  empno   = '200000')
ORDER BY projno;
```

Figure 142, Select values - from before update

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   projno  AS prj                                    ANSWER
        ,old_t   AS old_t                                  ===============
        ,emptime AS new_t                                  PRJ OLD_T NEW_T
FROM     NEW TABLE                                         --- ----- -----
   (UPDATE  emp_act                                        ABC  2.00  0.02
    INCLUDE (old_t DECIMAL(5,2))                           DEF  2.00 11.27
    SET      emptime = emptime * RAND(1) * 10
            ,old_t   = emptime
    WHERE    empno   = '200000')
ORDER BY 1;
```

Figure 143, Select values - before and after update

Delete Examples

The following example lists the rows that were deleted:

```
SELECT   projno AS prj                                            ANSWER
        ,actno  AS act                                            =======
FROM     OLD TABLE                                                PRJ ACT
   (DELETE                                                        --- ---
    FROM   emp_act                                                VVV 111
    WHERE  empno = '300000')                                      ZZZ 999
ORDER BY 1,2;
```

Figure 144, List deleted rows

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:

```
SELECT   empno                                         ANSWER
        ,projno                                        ====================
        ,actno  AS act                                 EMPNO  PROJNO ACT R#
        ,row#   AS r#                                  ------ ------ --- --
FROM     OLD TABLE                                     000260 AD3113  70  2
   (DELETE                                             000260 AD3113  80  4
    FROM    emp_act                                    000260 AD3113 180  6
    INCLUDE (row# SMALLINT)
    SET     row#  = ROW_NUMBER() OVER()
    WHERE   empno = '000260')
WHERE    row# = row# / 2 * 2
ORDER BY 1,2,3;
```

Figure 145, Assign row numbers to deleted rows

NOTE: 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:

```
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
    WHERE   empno < '0001')
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
```

Figure 146, Join result to another table

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.

Merge

A 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 in not in the target, it can be inserted.
- If desired, as SQL error can also be generated.

Below is the basic syntax diagram:

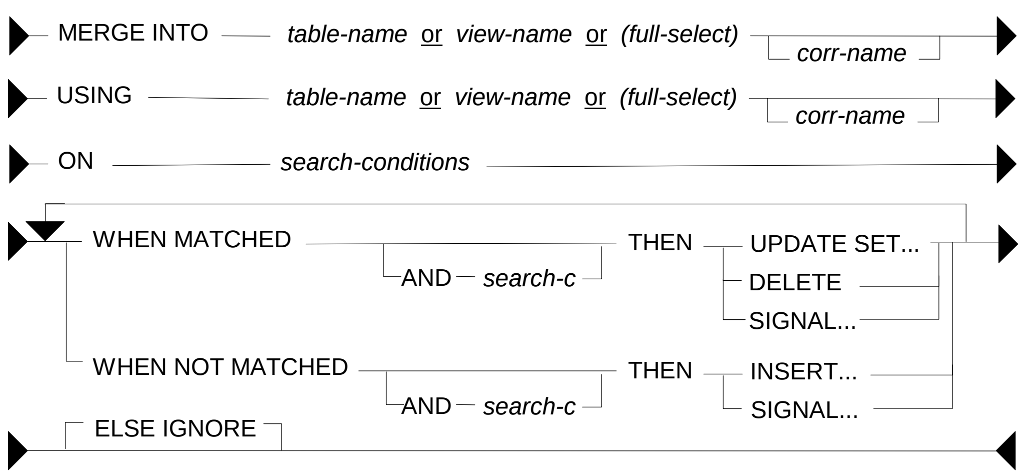

Figure 147, MERGE statement syntax

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 full-select 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.

Sample Tables

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

```
CREATE TABLE old_staff AS                OLD_STAFF              NEW_STAFF
   (SELECT id, job, salary               +-----------------+  +----------+
    FROM   staff)                        |ID|JOB  |SALARY   |  |ID|SALARY |
WITH NO DATA;                            |--|-----|---------|  |--|-------|
                                         |20|Sales|18171.25 |  |30|1750.67|
CREATE TABLE new_staff AS                |30|Mgr  |17506.75 |  |40|1800.60|
   (SELECT id, salary                    |40|Sales|18006.00 |  |50|2065.98|
    FROM   staff)                        +-----------------+  +----------+
WITH NO DATA;

INSERT INTO old_staff                    INSERT INTO new_staff
SELECT id, job, salary                   SELECT id, salary / 10
FROM   staff                             FROM   staff
WHERE  id BETWEEN 20 and 40;             WHERE  id BETWEEN 30 and 50;
```

Figure 148, Sample tables for merge

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 INTO old_staff oo                OLD_STAFF             NEW_STAFF
USING new_staff nn                     +-----------------+   +----------+
ON    oo.id = nn.id                    |ID|JOB  |SALARY  |   |ID|SALARY |
WHEN MATCHED THEN                      |--|-----|--------|   |--|-------|
   UPDATE                              |20|Sales|18171.25|   |30|1750.67|
   SET oo.salary = nn.salary           |30|Mgr  |17506.75|   |40|1800.60|
WHEN NOT MATCHED THEN                  |40|Sales|18006.00|   |50|2065.98|
   INSERT                              +-----------------+   +----------+
   VALUES (nn.id,'?',nn.salary);
                                                       AFTER-MERGE
                                                       =================
                                                       ID JOB   SALARY
                                                       -- ----- --------
                                                       20 Sales 18171.25
                                                       30 Mgr    1750.67
                                                       40 Sales  1800.60
                                                       50 ?      2065.98
```

Figure 149, Merge - do update or insert

Delete-only Merge

The next statement deletes all matching rows:

```
MERGE INTO old_staff oo                                AFTER-MERGE
USING new_staff nn                                     =================
ON    oo.id = nn.id                                    ID JOB   SALARY
WHEN MATCHED THEN                                      -- ----- --------
   DELETE;                                             20 Sales 18171.25
```

Figure 150, Merge - delete if match

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 INTO old_staff oo                OLD_STAFF             NEW_STAFF
USING new_staff nn                     +-----------------+   +----------+
ON    oo.id = nn.id                    |ID|JOB  |SALARY  |   |ID|SALARY |
WHEN MATCHED                           |--|-----|--------|   |--|-------|
AND  oo.salary < 18000 THEN            |20|Sales|18171.25|   |30|1750.67|
   UPDATE                              |30|Mgr  |17506.75|   |40|1800.60|
   SET oo.salary = nn.salary           |40|Sales|18006.00|   |50|2065.98|
WHEN MATCHED                           +-----------------+   +----------+
AND  oo.salary > 18000 THEN
   DELETE                                              AFTER-MERGE
WHEN NOT MATCHED                                       =================
AND  nn.id > 10 THEN                                   ID JOB   SALARY
   INSERT                                              -- ----- --------
   VALUES (nn.id,'?',nn.salary)                        20 Sales 18171.25
WHEN NOT MATCHED THEN                                  30 Mgr    1750.67
   SIGNAL SQLSTATE '70001'                             50 ?      2065.98
   SET MESSAGE_TEXT = 'New ID <= 10';
```

Figure 151, Merge with multiple options

The merge statement is like the case statement (see page 37) 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.

Using a Full-select

The following merge generates an input table (i.e. full-select) 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 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 18171.25
30 Mgr   17506.75
40 Sales 18006.00
41 Sales 18171.25
```

Figure 152, Merge MAX row into table

Here is the same thing written as a plain on 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;
```

Figure 153, Merge logic - done using insert

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

```
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                 NEW_STAFF
+-----------------+      +----------+
|ID|JOB  |SALARY  |      |ID|SALARY |
|--|-----|--------|      |--|-------|
|20|Sales|18171.25|      |30|1750.67|
|30|Mgr  |17506.75|      |40|1800.60|
|40|Sales|18006.00|      |50|2065.98|
+-----------------+      +----------+

AFTER-MERGE
=================
ID JOB   SALARY
-- ----- --------
20 Sales 18171.25
40 ?      1800.60
40 Sales 18006.00
```

Figure 154, Merge using two full-selects

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 full-select that was done before the merge.
- The source row with an ID of 40 was inserted, because it was not found in the target full-select. 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 full-select that was done before the merge.

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:

```
MERGE INTO old_staff oo                                     AFTER-MERGE
USING new_staff nn                                          =================
ON    oo.id = nn.id                                         ID JOB   SALARY
WHEN MATCHED THEN                                           -- ----- --------
   UPDATE                                                   20 Sales 18171.25
   SET (salary,job) = (1234,'?')                            30 ?      1234.00
WHEN NOT MATCHED THEN                                       40 ?      1234.00
   INSERT (id,salary,job)                                   50 ?      5678.90
   VALUES (id,5678.9,'?');
```

Figure 155, Listing columns and values in insert

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.

Introduction

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

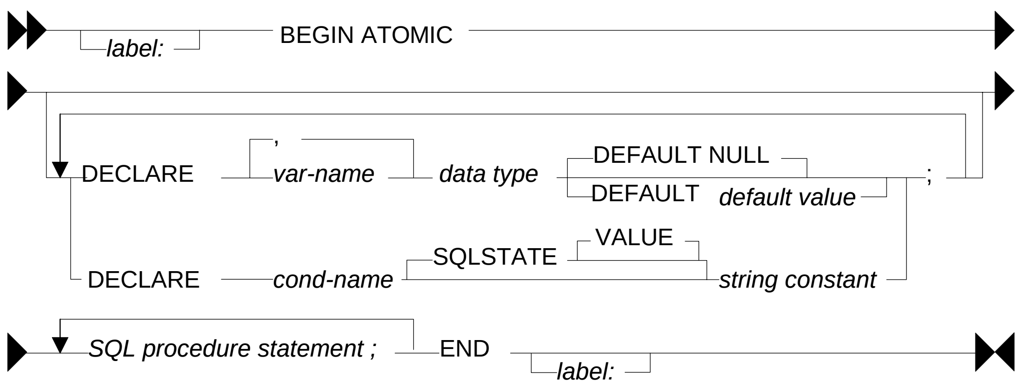

Figure 156, Compound SQL Statement syntax

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.

```
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
```

Figure 157, Sample Compound SQL statement

Statement Delimiter

DB2 SQL does not come with an designated statement delimiter (terminator), though a semi-colon is usually 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 !

SELECT NAME FROM STAFF WHERE ID = 10!

--#SET DELIMITER ;

SELECT NAME FROM STAFF WHERE ID = 20;
```

Figure 158, Set Delimiter example

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

NOTE: There are many more PSM control statements than what is shown above. But only these ones can be used in Compound SQL statements.

The following SQL statement can be issued:

- full-select
- UPDATE
- DELETE
- INSERT
- SET variable statement

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.

```
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
```

Figure 159, DECLARE examples

FOR Statement

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

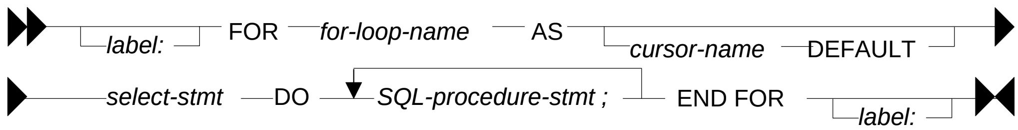

Figure 160, FOR statement syntax

In the example below, one row is fetched per DEPT in the STAFF table. That row is then used to do two independent updates:

```
BEGIN ATOMIC
   FOR V1 AS
      SELECT   dept    AS dname
              ,max(id) AS max_id
      FROM     staff
      GROUP BY dept
      HAVING   COUNT(*) > 1
      ORDER BY dept
   DO
      UPDATE staff
      SET    id   = id * -1
      WHERE  id   = max_id;
      UPDATE staff
      set    dept = dept / 10
      WHERE  dept = dname
        AND  dept < 30;
   END FOR;
END
```

Figure 161, FOR statement example

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).

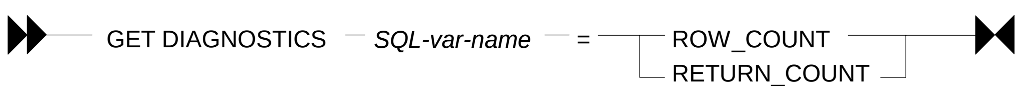

Figure 162, GET DIAGNOSTICS statement syntax

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:

```
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
```

Figure 163, GET DIAGNOSTICS statement example

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.

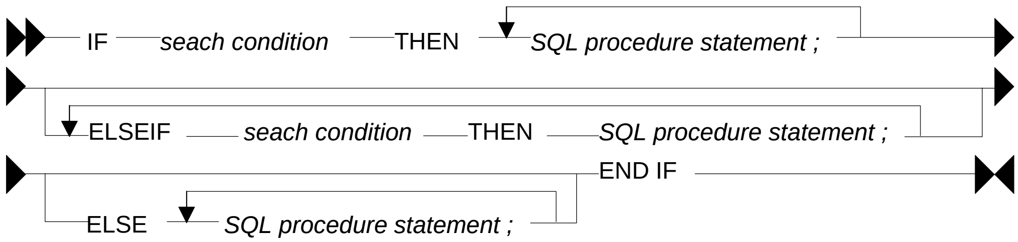

Figure 164, IF statement syntax

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

```
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
```

Figure 165, IF statement example

ITERATE Statement

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

Figure 166, ITERATE statement syntax

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:

```
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
```

Figure 167, ITERATE statement example

LEAVE Statement

The LEAVE statement exits the labeled loop.

Figure 168, LEAVE statement syntax

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:

```
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
```

Figure 169, LEAVE statement example

SIGNAL Statement

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

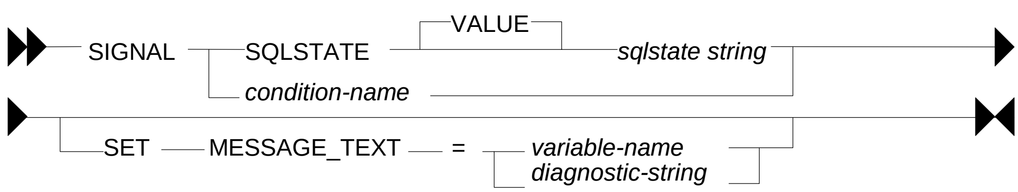

Figure 170, SIGNAL statement syntax

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:

```
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
```

Figure 171, SIGNAL statement example

WHILE Statement

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

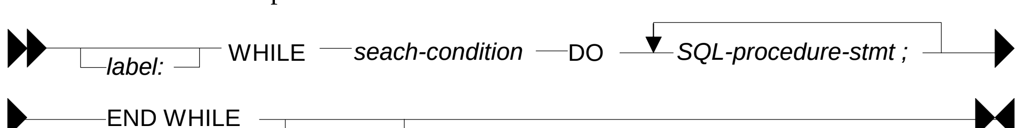

Figure 172, WHILE statement syntax

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

```
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
```

Figure 173, WHILE statement example

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.

NOTE: To write stored procedures in the SQL language, you need a C compiler.

Test Query

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

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

```
ANSWER
==========
DEPT #ROWS
---- -----
  10     4
  15     4
  20     4
  38     5
  42     4
  51     5
  66     5
  84     4
```

Figure 174, List departments in STAFF table

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:

```
--#SET DELIMITER !                                          IMPORTANT
                                                            ============
CREATE TABLE dpt                                            This example
(dept    SMALLINT     NOT NULL                              uses an "!"
,#names  SMALLINT                                           as the stmt
,PRIMARY KEY(dept))!                                        delimiter.
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;                                 ANSWER
END!                                                         ===========
COMMIT!                                                      DEPT #NAMES
                                                             ---- ------
INSERT INTO dpt (dept)                                         10      4
SELECT DISTINCT dept                                           15      4
FROM   staff!                                                  20      4
COMMIT!                                                        38      5
                                                               42      4
SELECT   *                                                     51      5
FROM     dpt                                                   66      5
ORDER BY dept!                                                 84      4
```
Figure 175, Trigger with compound SQL

NOTE: 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:

```
--#SET DELIMITER !                                         IMPORTANT
                                                           ============
CREATE FUNCTION dpt1 (deptin SMALLINT)                     This example
RETURNS SMALLINT                                           uses an "!"
BEGIN ATOMIC                                               as the stmt
   DECLARE num_names SMALLINT;                             delimiter.
   FOR getnames AS
      SELECT   COUNT(*) AS #n
      FROM     staff
      WHERE    dept = deptin
   DO
      SET num_names = #n;
   END FOR;                                                 ANSWER
   RETURN num_names;                                        ===========
END!                                                        DEPT #NAMES
COMMIT!                                                     ---- ------
                                                              10      4
SELECT   XXX.*                                                15      4
        ,dpt1(dept) as #names                                 20      4
FROM    (SELECT   dept                                        38      5
         FROM     staff                                       42      4
         GROUP BY dept                                        51      5
        )AS XXX                                               66      5
ORDER BY dept!                                                84      4
```

Figure 176, Scalar Function with compound SQL

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

```
--#SET DELIMITER !                                         IMPORTANT
                                                           ============
CREATE FUNCTION dpt1 (deptin SMALLINT)                     This example
RETURNS SMALLINT                                           uses an "!"
BEGIN ATOMIC                                               as the stmt
   RETURN                                                  delimiter.
   SELECT COUNT(*)
   FROM   staff
   WHERE  dept = deptin;
END!
COMMIT!

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

Figure 177, Scalar Function with compound SQL

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:

```
--#SET DELIMITER !                                          IMPORTANT
                                                            ============
CREATE FUNCTION dpt2 ()                                     This example
RETURNS TABLE (dept     SMALLINT                            uses an "!"
              ,#names   SMALLINT)                           as the stmt
BEGIN ATOMIC                                                delimiter.
   RETURN
   SELECT   dept
           ,count(*)                                         ANSWER
   FROM     staff                                            ===========
   GROUP BY dept                                             DEPT #NAMES
   ORDER BY dept;                                            ---- ------
END!                                                           10      4
COMMIT!                                                        15      4
                                                               20      4
--#SET DELIMITER ;                                             38      5
                                                               42      4
SELECT   *                                                     51      5
FROM     TABLE(dpt2()) T1                                      66      5
ORDER BY dept;                                                 84      4
```
Figure 178, Table Function with compound SQL

Column Functions

Introduction

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.

WARNING: 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.

Column Functions, Definitions

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.

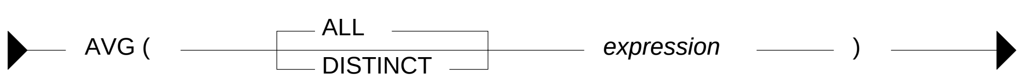

Figure 179, AVG function syntax

```
SELECT   AVG(DEPT)           AS A1                       ANSWER
        ,AVG(ALL DEPT)       AS A2                       ==============
        ,AVG(DISTINCT DEPT)  AS A3                       A1 A2 A3 A4 A5
        ,AVG(DEPT/10)        AS A4                       -- -- -- -- --
        ,AVG(DEPT)/10        AS A5                       41 41 40  3  4
FROM     STAFF
HAVING   AVG(DEPT) > 40;
```

Figure 180, AVG function examples

WARNING: 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:

```
UPDATE STAFF
SET    COMM = 0
WHERE  COMM IS NULL;

SELECT AVG(SALARY) AS SALARY                         ANSWER
      ,AVG(COMM)   AS COMM1                          ===================
      ,AVG(CASE COMM                                 SALARY  COMM1 COMM2
             WHEN 0 THEN NULL                        ------- ----- -----
             ELSE COMM                               16675.6 351.9 513.3
           END) AS COMM2
FROM   STAFF;

UPDATE STAFF
SET    COMM = NULL
WHERE  COMM = 0;
```

Figure 181, Convert zero to null before doing AVG

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 all return a null value when there are no matching rows. One use the COALESCE function, or a CASE expression, to convert the null value into a suitable substitute. Both methodologies are illustrated below:

```
SELECT   COUNT(*) AS C1                                       ANSWER
        ,AVG(SALARY) AS A1                                    ===========
        ,COALESCE(AVG(SALARY),0) AS A2                        C1 A1 A2 A3
        ,CASE                                                 -- -- -- --
           WHEN AVG(SALARY) IS NULL THEN 0                     0  -  0  0
           ELSE AVG(SALARY)
         END AS A3
FROM     STAFF
WHERE    ID < 10;
```

Figure 182, Convert null output (from AVG) to zero

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:

```
SELECT  AVG(DAYS(BIRTHDATE))                        ANSWER
       ,DATE(AVG(DAYS(BIRTHDATE)))                  =================
FROM    EMPLOYEE;                                   1      2
                                                    ------ ----------
                                                    709113 06/27/1942
```

Figure 183, AVG of date column

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  AVG(AVG_SAL) AS AVG_AVG                          ANSWER
FROM   (SELECT   DEPT                                    ================
                ,AVG(SALARY) AS AVG_SAL                  <Overflow error>
        FROM     STAFF
        GROUP BY DEPT
       )AS XXX;
```

Figure 184, Select average of average

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.

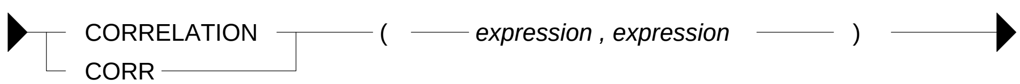

Figure 185, CORRELATION function syntax

```
WITH TEMP1(COL1, COL2, COL3, COL4) AS      ANSWER
(VALUES   (0    , 0    , 0    , RAND(1))   ===========================
 UNION ALL                                 COR11  COR12  COR23  COR34
 SELECT COL1 + 1                           ------ ------ ------ ------
       ,COL2 - 1                            1.000 -1.000 -0.017 -0.005
       ,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;
```

Figure 186, CORRELATION function examples

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.

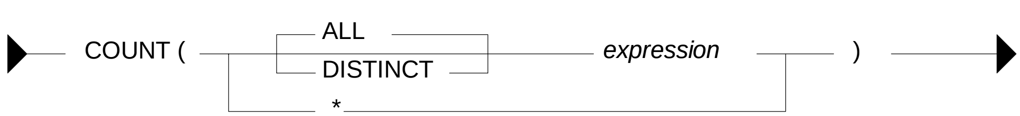

Figure 187, COUNT function syntax

```
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
```

Figure 188, COUNT function examples

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.

```
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
```

Figure 189, COUNT function with and without GROUP BY

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.

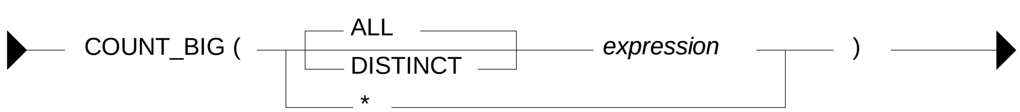

Figure 190, COUNT_BIG function syntax

```
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.
```

Figure 191, COUNT_BIG function examples

COVARIANCE

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

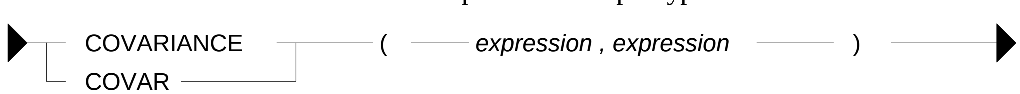

Figure 192, COVARIANCE function syntax

```
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
```

Figure 193, COVARIANCE function examples

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.

Figure 194, GROUPING function syntax

```
SELECT    DEPT
         ,AVG(SALARY)    AS SALARY
         ,GROUPING(DEPT) AS DF
FROM      STAFF
GROUP BY ROLLUP(DEPT)
ORDER BY DEPT;
```

```
ANSWER
================
DEPT SALARY   DF
---- -------- --
  10 20865.86  0
  15 15482.33  0
  20 16071.52  0
  38 15457.11  0
  42 14592.26  0
  51 17218.16  0
  66 17215.24  0
  84 16536.75  0
   - 16675.64  1
```

Figure 195, GROUPING function example

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

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.

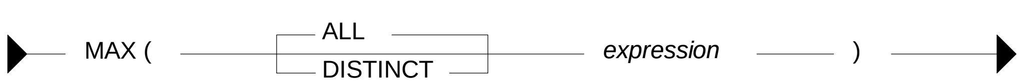

Figure 196, MAX function syntax

```
SELECT    MAX(DEPT)
         ,MAX(ALL DEPT)
         ,MAX(DISTINCT DEPT)
         ,MAX(DISTINCT DEPT/10)
FROM      STAFF;
```

```
ANSWER
===============
 1   2   3   4
--- --- --- ---
 84  84  84   8
```

Figure 197, MAX function examples

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.

```
SELECT MAX(HIREDATE)                     ANSWER
      ,CHAR(MAX(HIREDATE),USA)           ================================
      ,MAX(CHAR(HIREDATE,USA))           1          2          3
FROM   EMPLOYEE;                         ---------- ---------- ----------
                                         09/30/1980 09/30/1980 12/15/1976
```

Figure 198, MAX function with dates

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).

```
SELECT MAX(ID)          AS ID                         ANSWER
      ,MAX(CHAR(ID))    AS CHR                        ====================
      ,MAX(DIGITS(ID))  AS DIG                        ID     CHR    DIG
FROM   STAFF;                                         ------ ------ -----
                                                         350 90     00350
```

Figure 199, MAX function with numbers, 1 of 2

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.

```
SELECT MAX(ID - 250)          AS ID                 ANSWER
      ,MAX(CHAR(ID - 250))    AS CHR                =====================
      ,MAX(DIGITS(ID - 250))  AS DIG                ID    CHR  DIG
FROM   STAFF;                                       ----- ---- ----------
                                                      100 90   0000000240
```

Figure 200, MAX function with numbers, 2 of 2

WARNING: 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.

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.

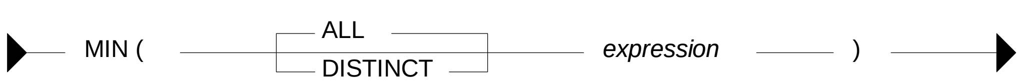

Figure 201, MIN function syntax

```
SELECT   MIN(DEPT)                                    ANSWER
        ,MIN(ALL DEPT)                                ===============
        ,MIN(DISTINCT DEPT)                            1   2   3   4
        ,MIN(DISTINCT DEPT/10)                        --- --- --- ---
FROM     STAFF;                                        10  10  10   1
```

Figure 202, MIN function examples

REGRESSION

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.

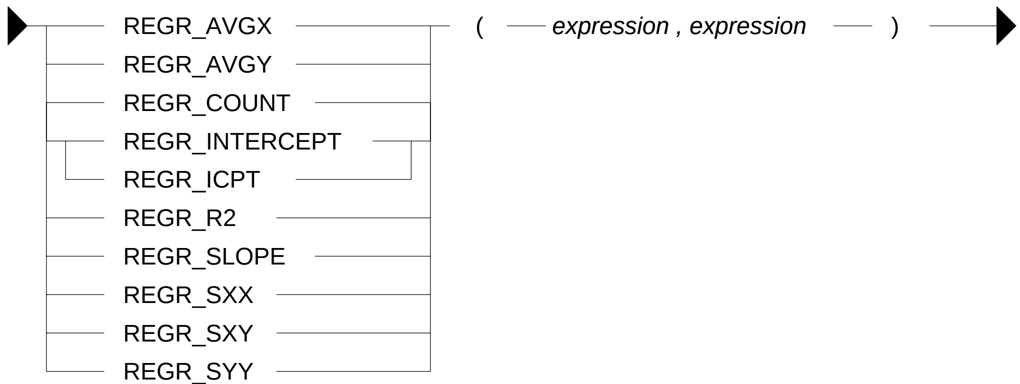

Figure 203, REGRESSION functions syntax

Functions

- 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.

```
                                                              ANSWERS
                                                              ==========
SELECT  DEC(REGR_SLOPE(BONUS,SALARY)     ,7,5)  AS R_SLOPE      0.01710
       ,DEC(REGR_INTERCEPT(BONUS,SALARY),7,3)  AS R_ICPT       100.871
       ,INT(REGR_COUNT(BONUS,SALARY)        )  AS R_COUNT            3
       ,INT(REGR_AVGX(BONUS,SALARY)         )  AS R_AVGX         42833
       ,INT(REGR_AVGY(BONUS,SALARY)         )  AS R_AVGY           833
       ,INT(REGR_SXX(BONUS,SALARY)          )  AS R_SXX      296291666
       ,INT(REGR_SXY(BONUS,SALARY)          )  AS R_SXY        5066666
       ,INT(REGR_SYY(BONUS,SALARY)          )  AS R_SYY          86666
FROM    EMPLOYEE
WHERE   WORKDEPT = 'A00';
```

Figure 204, REGRESSION functions examples

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.

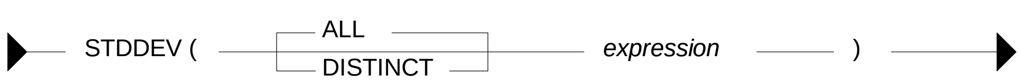

Figure 205, STDDEV function syntax

```
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
```

Figure 206, STDDEV function examples

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.

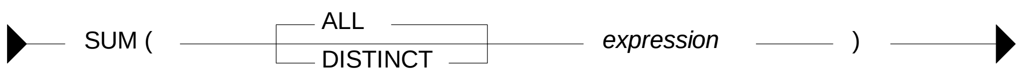

Figure 207, SUM function syntax

```
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
```

Figure 208, SUM function examples

WARNING: 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.

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.

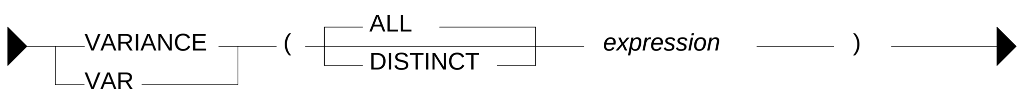

Figure 209, VARIANCE function syntax

```
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
```

Figure 210, VARIANCE function examples

OLAP Functions

Introduction

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

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, below is a simple little query:

```
SELECT   S1.JOB, S1.ID, S1.SALARY                        ANSWER
FROM     STAFF S1                                        =================
WHERE    S1.NAME LIKE '%s%'                              JOB   ID SALARY
  AND    S1.ID      <  90                                ----- -- --------
ORDER BY S1.JOB                                          Clerk 80 13504.60
        ,S1.ID;                                          Mgr   10 18357.50
                                                         Mgr   50 20659.80
```

Figure 211, Select rows from STAFF table

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:

```
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              ANSWER
FROM     STAFF S1                                              ======
WHERE    S1.NAME LIKE '%s%'                JOB   ID SALARY   SUMSAL   R
  AND    S1.ID      <  90                  ----- -- -------- -------- -
ORDER BY S1.JOB                            Clerk 80 13504.60 13504.60 1
        ,S1.ID;                            Mgr   10 18357.50 31862.10 2
                                           Mgr   50 20659.80 52521.90 3
```

Figure 212, Using OLAP functions to get additional fields

If one does not have OLAP functions, or one is too stupid to figure out how to use them, or one gets paid by the hour, 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 are several examples that use plain SQL to get the above answer. All of the examples have the same generic design (i.e. join each matching row to itself and all previous matching rows) and share similar problems (i.e. difficult to read, and poor performance).

Nested Table Expression

Below is a query that uses a nested table expression to get the additional fields. This SQL has the following significant features:

- The TABLE phrase is required because the nested table expression has a correlated reference to the prior table. See page 249 for more details on the use of this phrase.

- There are no join predicates between the nested table expression output and the original STAFF table. They are unnecessary because these predicates are provided in the body of the nested table expression. With them there, and the above TABLE function, the nested table expression is resolved once per row obtained from the STAFF S1 table.
- The original literal predicates have to be repeated in the nested table expression.
- The correlated predicates in the nested table expression have to match the ORDER BY sequence (i.e. first JOB, then ID) in the final output.

Now for the query:

```
SELECT   S1.JOB, S1.ID, S1.SALARY
        ,XX.SUMSAL, XX.R
FROM     STAFF S1
        ,TABLE
        (SELECT SUM(S2.SALARY)  AS SUMSAL
               ,COUNT(*)        AS R
         FROM   STAFF S2
         WHERE  S2.NAME LIKE '%s%'
           AND  S2.ID      <  90
           AND (S2.JOB     <  S1.JOB
            OR (S2.JOB     =  S1.JOB                  ANSWER
           AND  S2.ID     <=  S1.ID))     ============================
        )AS XX                            JOB   ID SALARY   SUMSAL   R
WHERE    S1.NAME LIKE '%s%'               ----- -- -------- -------- -
  AND    S1.ID      <  90                 Clerk 80 13504.60 13504.60 1
ORDER BY S1.JOB                           Mgr   10 18357.50 31862.10 2
        ,S1.ID;                           Mgr   50 20659.80 52521.90 3
```

Figure 213, Using Nested Table Expression to get additional fields

Ignoring any readability issues, this query has some major performance problems:

- The nested table expression is a partial Cartesian product. Each row fetched from "S1" is joined to all prior rows (in "S2"), which quickly gets to be very expensive.
- The join criteria match the ORDER BY fields. If the latter are suitably complicated, then the join is going to be inherently inefficient.

Self-Join and Group By

In the next example, the STAFF table is joined to itself such that each matching row obtained from the "S1" table is joined to all prior rows (plus the current row) in the "S2" table, where "prior" is a function of the ORDER BY clause used. After the join, a GROUP BY is needed in order to roll up the matching "S2" rows up into one:

```
SELECT   S1.JOB, S1.ID, S1.SALARY         ANSWER
        ,SUM(S2.SALARY) AS SUMSAL         ============================
        ,COUNT(*)       AS R              JOB   ID SALARY   SUMSAL   R
FROM     STAFF S1                         ----- -- -------- -------- -
        ,STAFF S2                         Clerk 80 13504.60 13504.60 1
WHERE    S1.NAME LIKE '%s%'               Mgr   10 18357.50 31862.10 2
  AND    S1.ID      <  90                 Mgr   50 20659.80 52521.90 3
  AND    S2.NAME LIKE '%s%'
  AND    S2.ID      <  90
  AND   (S2.JOB     <  S1.JOB
   OR   (S2.JOB     =  S1.JOB
  AND    S2.ID     <=  S1.ID))
GROUP BY S1.JOB
        ,S1.ID
        ,S1.SALARY
ORDER BY S1.JOB
        ,S1.ID;
```

Figure 214, Using Self-Join and Group By to get additional fields

Nested Table Expressions in Select

In our final example, two nested table expression are used to get the answer. Both are done in the SELECT part of the main query:

```
SELECT   S1.JOB, S1.ID, S1.SALARY
        ,(SELECT SUM(S2.SALARY)
          FROM   STAFF S2
          WHERE  S2.NAME LIKE '%s%'
            AND  S2.ID      <  90
            AND (S2.JOB     <  S1.JOB
             OR (S2.JOB     =  S1.JOB
            AND  S2.ID     <=  S1.ID))) AS SUMSAL
        ,(SELECT COUNT(*)
          FROM   STAFF S3
          WHERE  S3.NAME LIKE '%s%'
            AND  S3.ID      <  90
            AND (S3.JOB     <  S1.JOB
             OR (S3.JOB     =  S1.JOB
            AND  S3.ID     <=  S1.ID))) AS R
FROM     STAFF S1
WHERE    S1.NAME LIKE '%s%'                                  ANSWER
  AND    S1.ID      <  90                  ============================
ORDER BY S1.JOB                            JOB   ID SALARY   SUMSAL   R
        ,S1.ID;                            ----- -- -------- -------- -
                                           Clerk 80 13504.60 13504.60 1
                                           Mgr   10 18357.50 31862.10 2
                                           Mgr   50 20659.80 52521.90 3
```

Figure 215, Using Nested Table Expressions in Select to get additional fields

Once again, this query processes the matching rows multiple times, repeats predicates, has join predicates that match the ORDER BY, and does a partial Cartesian product. The only difference here is that this query commits all of the above sins twice.

Conclusion

Almost anything that an OLAP function does can be done some other way using simple SQL. But as the above examples illustrate, the alternatives are neither pretty nor efficient. And remember that the initial query used above was actually very simple. Feel free to try replacing the OLAP functions in the following query with their SQL equivalents:

```
SELECT   DPT.DEPTNAME
        ,EMP.EMPNO
        ,EMP.LASTNAME
        ,EMP.SALARY
        ,SUM(SALARY)  OVER(ORDER BY DPT.DEPTNAME ASC
                                   ,EMP.SALARY   DESC
                                   ,EMP.EMPNO    ASC)  AS SUMSAL
        ,ROW_NUMBER() OVER(ORDER BY DPT.DEPTNAME ASC
                                   ,EMP.SALARY   DESC
                                   ,EMP.EMPNO    ASC)  AS ROW#
FROM     EMPLOYEE   EMP
        ,DEPARTMENT DPT
WHERE    EMP.FIRSTNME LIKE '%S%'
  AND    EMP.WORKDEPT    =  DPT.DEPTNO
  AND    DPT.ADMRDEPT LIKE 'A%'
  AND    NOT EXISTS
        (SELECT  *
         FROM    EMP_ACT EAT
         WHERE   EMP.EMPNO   = EAT.EMPNO
           AND   EAT.EMPTIME > 10)
ORDER BY DPT.DEPTNAME ASC
        ,EMP.SALARY   DESC
        ,EMP.EMPNO    ASC;
```

Figure 216, Complicated query using OLAP functions

OLAP Functions, Definitions

Ranking Functions

The RANK and DENSE_RANK functions enable one to rank the rows returned by a query. The result is of type BIGINT.

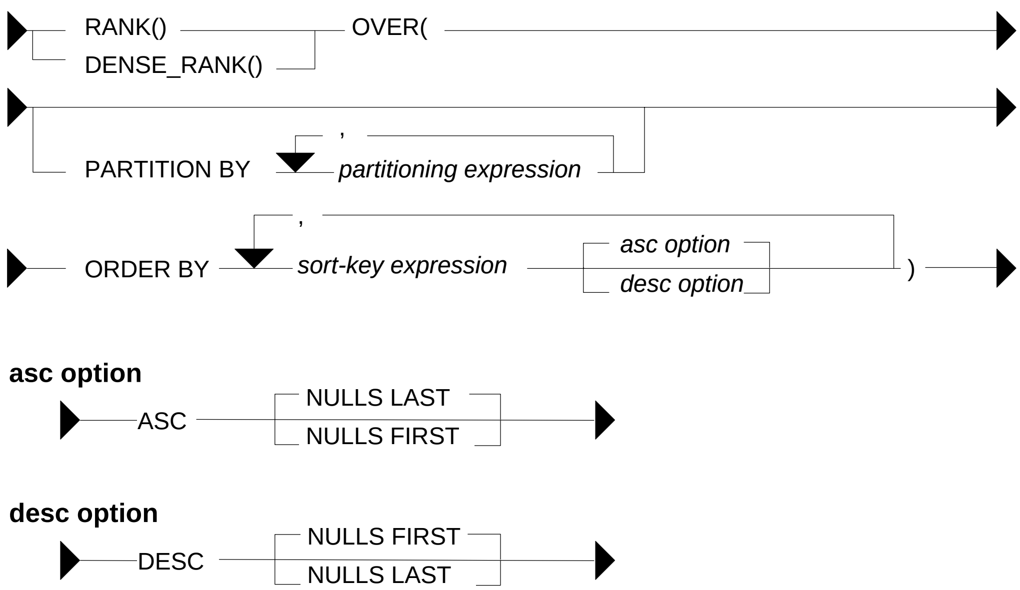

Figure 217, Ranking Functions syntax

NOTE: The ORDER BY phrase, which is required, is used to both sequence the values, and to tell DB2 when to generate a new value. See page 79 for details.

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.

The following query illustrates the use of the two functions:

```
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 IS NOT NULL         ANSWER
ORDER BY YEARS;                    ===================================
                                   ID YEARS SALARY   RANK# DENSE# ROW#
                                   -- ----- -------- ----- ------ ----
                                   30     5 17506.75     1      1    1
                                   40     6 18006.00     2      2    2
                                   90     6 18001.75     2      2    3
                                   10     7 18357.50     4      3    4
                                   70     7 16502.83     4      3    5
                                   20     8 18171.25     6      4    6
                                   50    10 20659.80     7      5    7
```

Figure 218, Ranking functions example

ORDER BY Usage

The ORDER BY phrase, which is mandatory, gives a sequence to the ranking, and also tells DB2 when to start a new rank value. The following query illustrates both uses:

```
SELECT   JOB
        ,YEARS
        ,ID
        ,NAME
        ,SMALLINT(RANK() OVER(ORDER BY JOB   ASC))  AS ASC1
        ,SMALLINT(RANK() OVER(ORDER BY JOB   ASC
                                      ,YEARS ASC))  AS ASC2
        ,SMALLINT(RANK() OVER(ORDER BY JOB   ASC
                                      ,YEARS ASC
                                      ,ID    ASC))  AS ASC3
        ,SMALLINT(RANK() OVER(ORDER BY JOB   DESC)) AS DSC1
        ,SMALLINT(RANK() OVER(ORDER BY JOB   DESC
                                      ,YEARS DESC)) AS DSC2
        ,SMALLINT(RANK() OVER(ORDER BY JOB   DESC
                                      ,YEARS DESC
                                      ,ID    DESC)) AS DSC3
        ,SMALLINT(RANK() OVER(ORDER BY JOB   ASC
                                      ,YEARS DESC
                                      ,ID    ASC))  AS MIX1
        ,SMALLINT(RANK() OVER(ORDER BY JOB   DESC
                                      ,YEARS ASC
                                      ,ID    DESC)) AS MIX2
FROM     STAFF
WHERE    ID      <  150
  AND    YEARS  IN (6,7)
  AND    JOB     >  'L'
ORDER BY JOB
        ,YEARS
        ,ID;
                                ANSWER
      =================================================================
      JOB   YEARS ID  NAME    ASC1 ASC2 ASC3  DSC1 DSC2 DSC3 MIX1 MIX2
      ----- ----- --- ------- ---- ---- ----  ---- ---- ---- ---- ----
      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
```

Figure 219, ORDER BY usage

Observe above that adding more fields to the ORDER BY phrase resulted in more ranking values being generated.

Ordering Nulls

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:

```
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                     ANSWER
        ,SALARY;                   ==================================
                                   ID YR SALARY    A  AF AL  D  DF DL
                                   -- -- --------  -- -- --  -- -- --
                                   30  5 17506.75   1  2  1   6  6  5
                                   90  6 18001.75   2  3  2   5  5  4
                                   40  6 18006.00   2  3  2   5  5  4
                                   70  7 16502.83   3  4  3   4  4  3
                                   10  7 18357.50   3  4  3   4  4  3
                                   20  8 18171.25   4  5  4   3  3  2
                                   50 10 20659.80   5  6  5   2  2  1
                                   80  - 13504.60   6  1  6   1  1  6
                                   60  - 16808.30   6  1  6   1  1  6
```

Figure 220, Overriding the default null ordering sequence

In general, in a relational database 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.

NOTE: The ORDER BY used in the ranking functions (above) has nothing to do with the ORDER BY at the end of the query. The latter defines the row output order, while the former tells each ranking function how to sequence the values. Likewise, one cannot define the null sort sequence when ordering the rows.

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:

```
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                                          ANSWER
ORDER BY 1;                                              =======
                                                         Y#1 Y#2
                                                         --- ---
                                                           5   6
```

Figure 221, Counting distinct values - comparison

PARTITION Usage

The 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:

```
SELECT   ID                                                    ANSWER
        ,YEARS  AS YR                                          =================
        ,SALARY                                                ID YR SALARY   R1
        ,RANK() OVER(PARTITION BY YEARS                        -- -- -------- --
                     ORDER     BY SALARY) AS R1                30  5 17506.75  1
FROM     STAFF                                                 40  6 18006.00  1
WHERE    ID     < 80                                           70  7 16502.83  1
  AND    YEARS IS NOT NULL                                     10  7 18357.50  2
ORDER BY YEARS                                                 20  8 18171.25  1
        ,SALARY;                                               50  0 20659.80  1
```
Figure 222, Values ranked by subset of rows

Multiple Rankings

One can do multiple independent rankings in the 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;
```
Figure 223, Multiple rankings in same query

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:

```
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 RAND()))           AS DUMB4
        ,SMALLINT(RANK() OVER(ORDER BY 1))                AS DUMB5
FROM     STAFF
WHERE    ID     < 40
  AND    YEARS IS NOT NULL
ORDER BY 1;
```
Figure 224, Dumb rankings, SQL

```
ID  YEARS  NAME      SALARY    DUMB1  DUMB2  DUMB3  DUMB4  DUMB5
--  -----  --------  --------  -----  -----  -----  -----  -----
10      7  Sanders   18357.50      1      3      1      1      1
20      8  Pernal    18171.25      3      2      3      3      1
30      5  Marenghi  17506.75      2      1      2      2      1
```
Figure 225, Dumb ranking, Answer

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:

```
SELECT   XXX.*                                          ANSWER
        ,RANK()OVER(ORDER BY ID) AS R2                  ================
FROM    (SELECT   ID                                    ID NAME    R1 R2
                 ,NAME                                  -- ------- -- --
                 ,RANK() OVER(ORDER BY ID) AS R1        40 O'Brien  4  1
         FROM     STAFF                                 50 Hanes    5  2
         WHERE    ID     < 100                          70 Rothman  6  3
           AND    YEARS IS NOT NULL                     90 Koonitz  7  4
        )AS XXX
WHERE    ID > 30
ORDER BY ID;
```

Figure 226, Subsequent processing of ranked data

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:

```
SELECT   ID                                            ANSWER
        ,RANK() OVER(PARTITION BY DEPT                 =================
                     ORDER BY SALARY DESC) AS R1       ID R1 SALARY   DP
        ,SALARY                                        -- -- -------- --
        ,DEPT AS DP                                    50  1 20659.80 15
FROM     STAFF                                         10  1 18357.50 20
WHERE    ID     < 80                                   40  1 18006.00 38
  AND    YEARS IS NOT NULL                             20  2 18171.25 20
ORDER BY R1     ASC                                    30  2 17506.75 38
        ,SALARY DESC;                                  70  2 16502.83 15
```

Figure 227, Ordering rows by rank, using RANK function

Here is the same query, written without the ranking function:

```
SELECT   ID                                            ANSWER
        ,(SELECT COUNT(*)                              =================
          FROM   STAFF S2                              ID R1 SALARY   DP
          WHERE  S2.ID        < 80                     -- -- -------- --
            AND  S2.YEARS IS NOT NULL                  50  1 20659.80 15
            AND  S2.DEPT      = S1.DEPT                10  1 18357.50 20
            AND  S2.SALARY   >= S1.SALARY) AS R1       40  1 18006.00 38
        ,SALARY                                        20  2 18171.25 20
        ,DEPT AS DP                                    30  2 17506.75 38
FROM     STAFF S1                                      70  2 16502.83 15
WHERE    ID     < 80
  AND    YEARS IS NOT NULL
ORDER BY R1     ASC
        ,SALARY DESC;
```

Figure 228, Ordering rows by rank, using sub-query

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.

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:

```
SELECT   ID                                                    ANSWER
        ,SALARY                                                ==============
        ,DEPT AS DP                                            ID SALARY   DP
FROM    (SELECT S1.*                                           -- -------- --
               ,RANK() OVER(PARTITION BY DEPT                  50 20659.80 15
                            ORDER BY SALARY DESC) AS R1        10 18357.50 20
         FROM     STAFF S1                                     40 18006.00 38
         WHERE    ID     < 80
           AND    YEARS IS NOT NULL
        )AS XXX
WHERE    R1 = 1
ORDER BY DP;
```

Figure 229, Get highest salary in each department, use RANK function

Here is the same query, written using a correlated sub-query:

```
SELECT   ID                                                    ANSWER
        ,SALARY                                                ==============
        ,DEPT AS DP                                            ID SALARY   DP
FROM     STAFF S1                                              -- -------- --
WHERE    ID     < 80                                           50 20659.80 15
  AND    YEARS IS NOT NULL                                     10 18357.50 20
  AND    NOT EXISTS                                            40 18006.00 38
        (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;
```

Figure 230, Get highest salary in each department, use correlated sub-query

Here is the same query, written using an uncorrelated sub-query:

```
SELECT   ID                                                    ANSWER
        ,SALARY                                                ==============
        ,DEPT AS DP                                            ID SALARY   DP
FROM     STAFF                                                 -- -------- --
WHERE    ID     < 80                                           50 20659.80 15
  AND    YEARS IS NOT NULL                                     10 18357.50 20
  AND   (DEPT, SALARY) IN                                      40 18006.00 38
        (SELECT   DEPT, MAX(SALARY)
         FROM     STAFF
         WHERE    ID        < 80
           AND    YEARS IS NOT NULL
         GROUP BY DEPT)
ORDER BY DP;
```

Figure 231, Get highest salary in each department, use uncorrelated sub-query

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.

NOTE: If it seems at times that this chapter was written with a poison pen, it is because just about now I had a "Microsoft moment" and my machine crashed. Needless to say, I had backups and, needless to say, they got trashed. It took me four days to get back to where I was. Thanks Bill - may you rot in hell. / Graeme

Row Numbering Function

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:

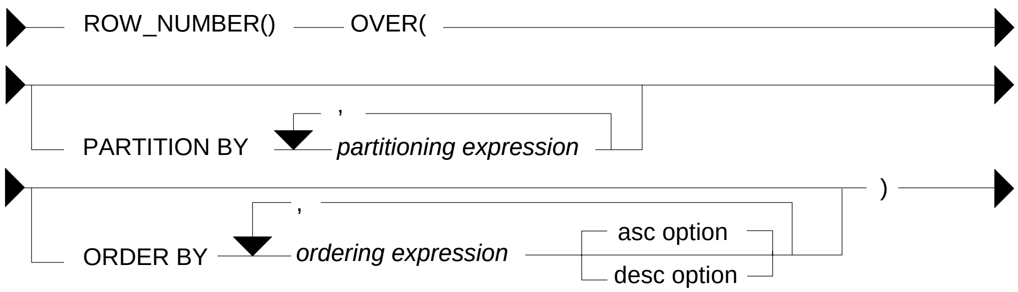

Figure 232, Numbering Function syntax

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:

```
SELECT   ID                                          ANSWER
        ,NAME                                        =================
        ,ROW_NUMBER() OVER()            AS R1        ID NAME     R1 R2
        ,ROW_NUMBER() OVER(ORDER BY ID) AS R2        -- -------- -- --
FROM     STAFF                                       10 Sanders   1  1
WHERE    ID     < 50                                 20 Pernal    2  2
  AND    YEARS IS NOT NULL                           30 Marenghi  3  3
ORDER BY ID;                                         40 O'Brien   4  4
```

Figure 233, ORDER BY example, 1 of 3

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:

```
SELECT   ID                                          ANSWER
        ,NAME                                        =================
        ,ROW_NUMBER() OVER()              AS R1      ID NAME     R1 R2
        ,ROW_NUMBER() OVER(ORDER BY NAME) AS R2      -- -------- -- --
FROM     STAFF                                       10 Sanders   4  4
WHERE    ID     < 50                                 20 Pernal    3  3
  AND    YEARS IS NOT NULL                           30 Marenghi  2  2
ORDER BY ID;                                         40 O'Brien   1  1
```

Figure 234, ORDER BY example, 2 of 3

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:

```
SELECT   ID                                               ANSWER
        ,NAME                                             ====================
        ,ROW_NUMBER() OVER()              AS R1           ID NAME     R1 R2 R3
        ,ROW_NUMBER() OVER(ORDER BY ID)   AS R2           -- -------- -- -- --
        ,ROW_NUMBER() OVER(ORDER BY NAME) AS R3           10 Sanders   1  1  4
FROM     STAFF                                            20 Pernal    2  2  3
WHERE    ID     < 50                                      30 Marenghi  3  3  1
  AND    YEARS IS NOT NULL                                40 O'Brien   4  4  2
ORDER BY ID;
```

Figure 235, ORDER BY example, 3 of 3

Observe that now the first function has reverted back to the original sequence.

The lesson to be learnt here is that the ROW_NUMBER function, when not given an explicit ORDER BY, may create a value in any odd sequence. Usually, the sequence will reflect the order in which the rows are returned - but not always.

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:

```
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)        ANSWER
  AND    JOB     >  'L'         ======================================
ORDER BY JOB                    JOB   YEARS ID  NAME    ROW# RN1# RN2#
        ,YEARS;                 ----- ----- --- ------- ---- ---- ----
                                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
```

Figure 236, Use of PARTITION phrase

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   *                                                      ANSWER
FROM     (SELECT   ID                                           =============
                  ,NAME                                         ID NAME     R
                  ,ROW_NUMBER() OVER(ORDER BY ID) AS R          -- -------- -
          FROM     STAFF                                        10 Sanders  1
          WHERE    ID     < 100                                 20 Pernal   2
            AND    YEARS IS NOT NULL                            30 Marenghi 3
         )AS XXX
WHERE    R <= 3
ORDER BY ID;
```
Figure 237, Select first 3 rows, using ROW_NUMBER function

In the next query, the FETCH FIRST "n" ROWS notation is used to achieve the same result:

```
SELECT   ID                                                     ANSWER
        ,NAME                                                   =============
        ,ROW_NUMBER() OVER(ORDER BY ID) AS R                    ID NAME     R
FROM     STAFF                                                  -- -------- -
WHERE    ID     < 100                                           10 Sanders  1
  AND    YEARS IS NOT NULL                                      20 Pernal   2
ORDER BY ID                                                     30 Marenghi 3
FETCH FIRST 3 ROWS ONLY;
```
Figure 238, Select first 3 rows, using FETCH FIRST notation

So far, the ROW_NUMBER and the FETCH FIRST notations seem to be about the same. But the former technique is much more flexible. To illustrate, in the next query we retrieve the 3rd through 6th matching rows:

```
SELECT   *                                                      ANSWER
FROM     (SELECT   ID                                           =============
                  ,NAME                                         ID NAME     R
                  ,ROW_NUMBER() OVER(ORDER BY ID) AS R          -- -------- -
          FROM     STAFF                                        30 Marenghi 3
          WHERE    ID         < 200                             40 O'Brien  4
            AND    YEARS IS NOT NULL                            50 Hanes    5
         )AS XXX                                                70 Rothman  6
WHERE    R BETWEEN 3 AND 6
ORDER BY ID;
```
Figure 239, Select 3rd through 6th rows

In the next query we get every 5th matching row **-** starting with the first:

```
SELECT   *                                                     ANSWER
FROM     (SELECT   ID                                          ==============
                  ,NAME                                        ID  NAME    R
                  ,ROW_NUMBER() OVER(ORDER BY ID) AS R         --- ------- --
          FROM     STAFF                                        10 Sanders  1
          WHERE    ID         < 200                             70 Rothman  6
            AND    YEARS IS NOT NULL                           140 Fraye   11
         )AS XXX                                               190 Sneider 16
WHERE    (R - 1) = ((R - 1) / 5) * 5
ORDER BY ID;
```
Figure 240, Select every 5th matching row

In the next query we get the last two matching 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                         ANSWER
        )AS XXX                                             ==============
WHERE    R <= 2                                             ID  NAME     R
ORDER BY ID;                                                --- -------- -
                                                            180 Abrahams 2
                                                            190 Sneider  1
```

Figure 241, Select last two rows

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. Put the result into a temporary table - TEMP1.
- Query the TEMP1 table, getting the ranking of whatever row we want to stop fetching at. In this case, it is the 3rd row. Put the result into a temporary table - TEMP2.
- Finally, join to the two temporary tables. Fetch those rows in TEMP1 that have a ranking that is less than or equal to the single row in TEMP2.

```
WITH
TEMP1(YEARS, ID, NAME, RNK, ROW) AS
  (SELECT  YEARS
          ,ID
          ,NAME
          ,RANK()       OVER(ORDER BY YEARS)
          ,ROW_NUMBER() OVER(ORDER BY YEARS, ID)
   FROM    STAFF
   WHERE   ID         < 200
     AND   YEARS IS NOT NULL
),
TEMP2(RNK) AS
  (SELECT  RNK
   FROM    TEMP1
   WHERE   ROW = 3                            ANSWER
)                                             ==========================
SELECT   TEMP1.*                              YEARS ID  NAME     RNK ROW
FROM     TEMP1                                ----- --- -------- --- ---
        ,TEMP2                                    3 180 Abrahams   1   1
WHERE    TEMP1.RNK <= TEMP2.RNK                   4 170 Kermisch   2   2
ORDER BY YEARS                                    5  30 Marenghi   3   3
        ,ID;                                      5 110 Ngan       3   4
```

Figure 242, Select first "n" rows, or more if needed

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.

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 a sample 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:

```
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));
```

Figure 243, Performance test table - definition

The next SQL statement will insert 100,000 rows into the above table. After the rows were inserted, RUNSTATS was run, so the optimizer could choose the best access path.

```
INSERT INTO INVOICE
WITH TEMP (N,M) AS
(VALUES   (INTEGER(0),RAND(1))
 UNION ALL
 SELECT  N+1, RAND()
 FROM    TEMP
 WHERE   N+1 < 100000
)
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;
```

Figure 244, Performance test table - insert 100,000 rows

Imagine we want to retrieve the first five rows (only) from the above table. Below are several queries that will get this result. For each query, for the elapsed time, as measured by the DB2 Event Monitor is provided.

Below we use the "FETCH FIRST n ROWS" notation to stop the query at the 5th row. This query first did a tablespace scan, then sorted all 100,000 matching rows, and then fetched the first five. It was not cheap:

```
SELECT   S.*
        ,ROW_NUMBER()  OVER() AS ROW#
FROM     INVOICE S
ORDER BY INV#
FETCH FIRST 5 ROWS ONLY;
```

Figure 245, Fetch first 5 rows - 2.837 elapsed seconds

The next query is essentially the same as the prior, but this time we told DB2 to optimize the query for fetching five rows. Now one would think that the optimizer would already know this, but it evidently did not. This query used the INV# index to retrieve the rows without sorting. It stopped processing at the 5th row. Observe that it was almost a thousand times faster than the prior example:

```
SELECT   S.*
        ,ROW_NUMBER()  OVER() AS ROW#
FROM     INVOICE S
ORDER BY INV#
FETCH FIRST 5 ROWS ONLY
OPTIMIZE FOR 5 ROWS;
```

Figure 246, Fetch first 5 rows - 0.003 elapsed seconds

The next query uses the ROW_NUMBER function to sequence the rows. Subsequently, only those rows with a row-number less than or equal to five are retrieved. DB2 answers this query using a single non-matching index scan of the whole table. No temporary table is used, and nor is a sort done, but the query is not exactly cheap

```
SELECT   *
FROM    (SELECT   S.*
                 ,ROW_NUMBER()  OVER() AS ROW#
         FROM     INVOICE S
        )XXX
WHERE    ROW# <= 5
ORDER BY INV#;
```

Figure 247, Fetch first 5 rows - 0.691 elapsed seconds

At about this point, almost any halfway-competent idiot would conclude that the best way to make the above query run faster is to add the same "OPTIMIZE FOR 5 ROWS" notation that did wonders in the prior example. So we did (see below), but the access path remained the same, and the query now ran significantly slower:

```
SELECT   *
FROM    (SELECT   S.*
                 ,ROW_NUMBER()  OVER() AS ROW#
         FROM     INVOICE S
        )XXX
WHERE    ROW# <= 5
ORDER BY INV#
OPTIMIZE FOR 5 ROWS;
```

Figure 248, Fetch first 5 rows - 2.363 elapsed seconds

One can also use recursion to get the first "n" rows. One begins by getting the first matching row, and then one uses that row to get the next, and then the next, and so on (in a recursive join), until the required number of rows has 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:

```
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;
```

Figure 249, Fetch first 5 rows - 0.005 elapsed seconds

The above technique is nice to know, but it will have few practical uses, because it has several major disadvantages:

- It is not exactly easy to understand.
- It requires all primary predicates (e.g. get only those rows where the sale-value is greater than $10,000, and the sale-date greater than last month) to 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 conclusion, 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 100,000 matching rows. In situations where there are far fewer matching rows, one can reasonably assume that this problem is not an issue.

Aggregation Function

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.

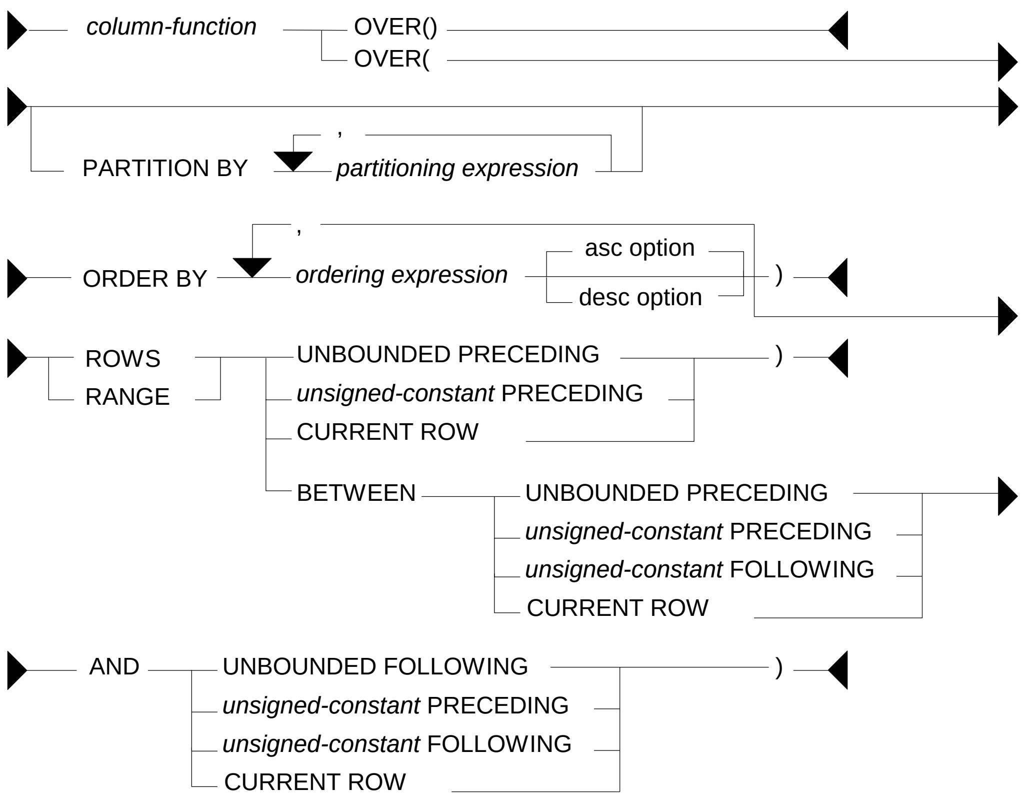

Figure 250, Aggregation Function syntax

Syntax Notes

Guess what - this is a complicated function. Be aware of the following:

- Any DB2 column function (e.g. AVG, SUM, COUNT) 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 (see page 92).
- The PARTITION phrase limits any aggregation to a subset of the matching rows.
- The ORDER BY phrase has two purposes; It defines a set of values to do aggregations on. Each distinct value gets a new result. It also defines a direction for the aggregation function processing - either ascending or descending (see page 93).
- An ORDER BY phrase is required if the aggregation is confined to a set of rows or range of values. In addition, if a RANGE is used, 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.
- The ROWS phrase limits the aggregation result to a set of rows - defined relative to the current row being processed. The applicable rows can either be already processed (i.e. preceding) or not yet processed (i.e. following), or both (see page 94).

- The RANGE phrase limits the aggregation result to a range of values - defined relative to the value of the current row being processed. The range is calculated by taking the value in the current row (defined by the ORDER BY phrase) and adding to and/or subtracting from it, then seeing what other rows are in the range. For this reason, when RANGE is used, only one expression can be specified in the aggregation function ORDER BY, and the expression must be numeric (see page 97).
- Preceding rows have already been fetched. Thus, the phrase "ROWS 3 PRECEDING" refers to the 3 preceding rows - plus the current row. The phrase "UNBOUNDED PRECEDING" refers to all those rows (in the partition) that have already been fetched, plus the current one.
- Following rows have yet to be fetched. The phrase "UNBOUNDED FOLLOWING" refers to all those rows (in the partition) that have yet to be fetched, plus the current one.
- The phrase CURRENT ROW refers to the current row. It is equivalent to getting zero preceding and following rows.
- If either a ROWS or a RANGE phrase is used, but no BETWEEN is provided, then one must provide a starting point for the aggregation (e.g. ROWS 1 PRECEDING). The starting point must either precede or equal the current row - it cannot follow it. The implied end point is the current row.
- When using the BETWEEN phrase, put the "low" value in the first check and the "high" value in the second check. Thus one can go from the 1 PRECEDING to the CURRENT ROW, or from the CURRENT ROW to 1 FOLLOWING, but not the other way round.
- The set of rows that match the BETWEEN phrase differ depending upon whether the aggregation function ORDER BY is ascending or descending.

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:

```
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 < 60
ORDER BY ID;
```
Figure 251, Aggregation function, basic usage, SQL

Below is the answer

```
ID  NAME      SALARY    SUM_SAL   AVG_SAL   MIN_SAL   MAX_SAL   #ROWS
--  --------  --------  --------  --------  --------  --------  -----
10  Sanders   18357.50  92701.30  18540.26  17506.75  20659.80      5
20  Pernal    18171.25  92701.30  18540.26  17506.75  20659.80      5
30  Marenghi  17506.75  92701.30  18540.26  17506.75  20659.80      5
40  O'Brien   18006.00  92701.30  18540.26  17506.75  20659.80      5
50  Hanes     20659.80  92701.30  18540.26  17506.75  20659.80      5
```

Figure 252, Aggregation function, basic usage, Answer

It is possible to do exactly the same thing using old-fashioned SQL, but it is not so pretty:

```
WITH
TEMP1 (ID, NAME, SALARY) AS
  (SELECT   ID, NAME, SALARY
   FROM     STAFF
   WHERE    ID < 60
),
TEMP2 (SUM_SAL, AVG_SAL, MIN_SAL, MAX_SAL, #ROWS) AS
  (SELECT   SUM(SALARY)
           ,AVG(SALARY)
           ,MIN(SALARY)
           ,MAX(SALARY)
           ,COUNT(*)
   FROM     TEMP1
)
SELECT   *
FROM     TEMP1
        ,TEMP2
ORDER BY ID;
```

Figure 253, Select detailed data, plus summary data

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:

```
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;
```

Figure 254, Logically equivalent aggregation functions, SQL

```
ID  NAME      SALARY    SUM1      SUM2      SUM3      SUM4
--  --------  --------  --------  --------  --------  --------
10  Sanders   18357.50  92701.30  92701.30  92701.30  92701.30
20  Pernal    18171.25  92701.30  92701.30  92701.30  92701.30
30  Marenghi  17506.75  92701.30  92701.30  92701.30  92701.30
40  O'Brien   18006.00  92701.30  92701.30  92701.30  92701.30
50  Hanes     20659.80  92701.30  92701.30  92701.30  92701.30
```

Figure 255, Logically equivalent aggregation functions, Answer

ORDER BY Usage

The ORDER BY phrase 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 done 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:

```
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 ROW1
        ,COUNT(*)     OVER(ORDER BY DEPT, NAME)           AS ROW2
FROM     STAFF
WHERE    ID < 60
ORDER BY DEPT
        ,NAME;
```

Figure 256, Aggregation function, order by usage, SQL

The answer is below. 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:

```
DEPT NAME     SALARY   SUM1     SUM2     SUM3     SUM4     ROW1 ROW2
---- -------- -------- -------- -------- -------- -------- ---- ----
  15 Hanes    20659.80 20659.80 92701.30 20659.80 92701.30    1    1
  20 Pernal   18171.25 57188.55 72041.50 38831.05 72041.50    3    2
  20 Sanders  18357.50 57188.55 72041.50 57188.55 53870.25    3    3
  38 Marenghi 17506.75 92701.30 35512.75 74695.30 35512.75    5    4
  38 O’Brien  18006.00 92701.30 35512.75 92701.30 18006.00    5    5
```

Figure 257, Aggregation function, order by usage, Answer

ROWS Usage

The ROWS phrase can be used to limit the aggregation function to a subset of the matching rows or distinct values. If no ROWS or RANGE phrase is provided, the aggregation is done for all preceding rows, up to the current row. Likewise, if no BETWEEN phrase is provided, the aggregation is done from the start-location given, up to the current row. In the following query, all of the examples using the ROWS phrase are of this type:

```
SELECT   DEPT
        ,NAME
        ,YEARS
        ,SMALLINT(SUM(YEARS) OVER(ORDER BY DEPT))               AS D
        ,SMALLINT(SUM(YEARS) OVER(ORDER BY DEPT, NAME))         AS DN
        ,SMALLINT(SUM(YEARS) OVER(ORDER BY DEPT, NAME
                                  ROWS     UNBOUNDED PRECEDING))AS DNU
        ,SMALLINT(SUM(YEARS) OVER(ORDER BY DEPT, NAME
                                  ROWS     3 PRECEDING))        AS DN3
        ,SMALLINT(SUM(YEARS) OVER(ORDER BY DEPT, NAME
                                  ROWS     1 PRECEDING))        AS DN1
        ,SMALLINT(SUM(YEARS) OVER(ORDER BY DEPT, NAME
                                  ROWS     0 PRECEDING))        AS DN0
        ,SMALLINT(SUM(YEARS) OVER(ORDER BY DEPT, NAME
                                  ROWS     CURRENT ROW))        AS DNC
        ,SMALLINT(SUM(YEARS) OVER(ORDER BY DEPT DESC, NAME DESC
                                  ROWS     1 PRECEDING))        AS DNX
FROM     STAFF
WHERE    ID         < 100
  AND    YEARS IS NOT NULL
ORDER BY DEPT
        ,NAME;
```

Figure 258, Starting ROWS usage. Implied end is current row, SQL

Below is the answer. Observe that an aggregation starting at the current row, or including zero proceeding rows, doesn't aggregate anything other than the current row:

```
DEPT   NAME       YEARS   D    DN   DNU   DN3   DN1   DN0   DNC   DNX
----   --------   -----   --   --   ---   ---   ---   ---   ---   ---
  15   Hanes         10   17   10    10    10    10    10    10    17
  15   Rothman        7   17   17    17    17    17     7     7    15
  20   Pernal         8   32   25    25    25    15     8     8    15
  20   Sanders        7   32   32    32    32    15     7     7    12
  38   Marenghi       5   43   37    37    27    12     5     5    11
  38   O'Brien        6   43   43    43    26    11     6     6    12
  42   Koonitz        6   49   49    49    24    12     6     6     6
```

Figure 259, Starting ROWS usage. Implied end is current row, Answer

BETWEEN Usage

In the next query, the BETWEEN phrase is used to explicitly define the start and end rows that are used in the aggregation:

```
SELECT   DEPT
        ,NAME
        ,YEARS
        ,SMALLINT(SUM(YEARS) OVER(ORDER BY DEPT, NAME))         AS UC1
        ,SMALLINT(SUM(YEARS) OVER(ORDER BY DEPT, NAME
                             ROWS         UNBOUNDED PRECEDING)) AS UC2
        ,SMALLINT(SUM(YEARS) OVER(ORDER BY DEPT, NAME
                             ROWS BETWEEN UNBOUNDED PRECEDING
                                          AND CURRENT ROW))     AS UC3
        ,SMALLINT(SUM(YEARS) OVER(ORDER BY DEPT, NAME
                             ROWS BETWEEN CURRENT ROW
                                      AND CURRENT ROW))         AS CU1
        ,SMALLINT(SUM(YEARS) OVER(ORDER BY DEPT, NAME
                             ROWS BETWEEN 1 PRECEDING
                                      AND 1 FOLLOWING))         AS PF1
        ,SMALLINT(SUM(YEARS) OVER(ORDER BY DEPT, NAME
                             ROWS BETWEEN 2 PRECEDING
                                      AND 2 FOLLOWING))         AS PF2
        ,SMALLINT(SUM(YEARS) OVER(ORDER BY DEPT, NAME
                             ROWS BETWEEN 3 PRECEDING
                                      AND 3 FOLLOWING))         AS PF3
        ,SMALLINT(SUM(YEARS) OVER(ORDER BY DEPT, NAME
                             ROWS BETWEEN CURRENT ROW
                                      AND UNBOUNDED FOLLOWING)) AS CU1
        ,SMALLINT(SUM(YEARS) OVER(ORDER BY DEPT, NAME
                             ROWS BETWEEN UNBOUNDED PRECEDING
                                      AND UNBOUNDED FOLLOWING)) AS UU1
FROM     STAFF
WHERE    ID          < 100
  AND    YEARS IS NOT NULL
ORDER BY DEPT
        ,NAME;
```

Figure 260, ROWS usage, with BETWEEN phrase, SQL

Now for the answer. Observe that the first three aggregation calls are logically equivalent:

```
DEPT  NAME      YEARS  UC1  UC2  UC3  CU1  PF1  PF2  PF3  CU1  UU1
----  --------  -----  ---  ---  ---  ---  ---  ---  ---  ---  ---
  15  Hanes        10   10   10   10   10   17   25   32   49   49
  15  Rothman       7   17   17   17    7   25   32   37   39   49
  20  Pernal        8   25   25   25    8   22   37   43   32   49
  20  Sanders       7   32   32   32    7   20   33   49   24   49
  38  Marenghi      5   37   37   37    5   18   32   39   17   49
  38  O'Brien       6   43   43   43    6   17   24   32   12   49
  42  Koonitz       6   49   49   49    6   12   17   24    6   49
```

Figure 261, ROWS usage, with BETWEEN phrase, Answer

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 function 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:

```
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                 ANSWER
ORDER BY ID;                               ===========================
                                           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
```

Figure 262,BETWEEN and ORDER BY usage

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
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


NOTE: Preceding row is always on LEFT of current row.
      Following row is always on RIGHT of current row.
```

Figure 263, Explanation of query

IMPORTANT: 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.

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. 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 rows are in the range. Note that 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:

```
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;
```

Figure 264, RANGE usage, SQL

Now for the 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
```

Figure 265, RANGE usage, Answer

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.

PARTITION Usage

One can take all of the lovely stuff described above, and make it whole lot more complicated by using the PARTITION expression. This phrase limits the current processing of the aggregation to a subset of the matching rows.

In the following query, some of the aggregation functions are broken up by partition range and some are not. When there is a partition, then the ROWS check only works within the range of the partition (i.e. for a given DEPT):

```
SELECT   DEPT
        ,NAME
        ,YEARS
        ,SMALLINT(SUM(YEARS) OVER(ORDER     BY DEPT))        AS X
        ,SMALLINT(SUM(YEARS) OVER(ORDER     BY DEPT
                             ROWS 3 PRECEDING))         AS XO3
        ,SMALLINT(SUM(YEARS) OVER(ORDER     BY DEPT
                             ROWS BETWEEN 1 PRECEDING
                                      AND 1 FOLLOWING)) AS XO11
        ,SMALLINT(SUM(YEARS) OVER(PARTITION BY DEPT))        AS P
        ,SMALLINT(SUM(YEARS) OVER(PARTITION BY DEPT
                             ORDER     BY DEPT))        AS PO
        ,SMALLINT(SUM(YEARS) OVER(PARTITION BY DEPT
                             ORDER     BY DEPT
                             ROWS 1 PRECEDING))         AS PO1
        ,SMALLINT(SUM(YEARS) OVER(PARTITION BY DEPT
                             ORDER     BY DEPT
                             ROWS 3 PRECEDING))         AS PO3
        ,SMALLINT(SUM(YEARS) OVER(PARTITION BY DEPT
                             ORDER     BY DEPT
                             ROWS BETWEEN 1 PRECEDING
                                      AND 1 FOLLOWING)) AS PO11
FROM     STAFF
WHERE    ID BETWEEN 40 AND 120
  AND    YEARS IS NOT NULL
ORDER BY DEPT
        ,NAME;
```

Figure 266, PARTITION usage, SQL

```
DEPT   NAME     YEARS   X     XO3  XO11   P     PO    PO1   PO3  PO11
-----  -------  -----  ----  ----  ----  ----  ----  ----  ----  ----
   15  Hanes       10    22    10    15    22    22    10    10    15
   15  Ngan         5    22    15    22    22    22    15    15    22
   15  Rothman      7    22    22    18    22    22    12    22    12
   38  O'Brien      6    28    28    19     6     6     6     6     6
   42  Koonitz      6    41    24    19    13    13     6     6    13
   42  Plotz        7    41    26    13    13    13    13    13    13
```

Figure 267, PARTITION usage, Answer

PARTITION vs. GROUP BY

The PARTITION clause, when used by itself, returns a very similar result to a GROUP BY, except that it does not remove the duplicate rows. To illustrate, below is a simple query that does a GROUP BY:

```
SELECT   DEPT                                         ANSWER
        ,SUM(YEARS) AS SUM                            ================
        ,AVG(YEARS) AS AVG                            DEPT SUM AVG ROW
        ,COUNT(*)   AS ROW                            ---- --- --- ---
FROM     STAFF                                          15  22   7   3
WHERE    ID BETWEEN 40 AND 120                          38   6   6   1
  AND    YEARS IS NOT NULL                              42  13   6   2
GROUP BY DEPT;
```

Figure 268, Sample query using GROUP BY

Below is a similar query that uses the PARTITION phrase. Observe that the answer is the same, except that duplicate rows have not been removed:

```
SELECT   DEPT                                               ANSWER
        ,SUM(YEARS) OVER(PARTITION BY DEPT) AS SUM          =================
        ,AVG(YEARS) OVER(PARTITION BY DEPT) AS AVG          DEPT  SUM AVG ROW
        ,COUNT(*)   OVER(PARTITION BY DEPT) AS ROW          ----- --- --- ---
FROM     STAFF                                                 15  22   7   3
WHERE    ID BETWEEN 40 AND 120                                 15  22   7   3
  AND    YEARS IS NOT NULL                                     15  22   7   3
ORDER BY DEPT;                                                 38   6   6   1
                                                               42  13   6   2
                                                               42  13   6   2
```

Figure 269, Sample query using PARTITION

Below is another similar query that uses the PARTITION phrase, and then uses a DISTINCT clause to remove the duplicate rows:

```
SELECT   DISTINCT DEPT                                      ANSWER
        ,SUM(YEARS) OVER(PARTITION BY DEPT) AS SUM          =================
        ,AVG(YEARS) OVER(PARTITION BY DEPT) AS AVG          DEPT  SUM AVG ROW
        ,COUNT(*)   OVER(PARTITION BY DEPT) AS ROW          ----- --- --- ---
FROM     STAFF                                                 15  22   7   3
WHERE    ID BETWEEN 40 AND 120                                 38   6   6   1
  AND    YEARS IS NOT NULL                                     42  13   6   2
ORDER BY DEPT;
```

Figure 270, Sample query using PARTITION and DISTINCT

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 certainly is not as easy to understand. Therefore, when in doubt, use the GROUP BY syntax.

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.

WARNING: Some of the scalar functions changed their internal logic between V5 and V6 of DB2. There have been no changes between V6 and V7, or between V7 and V8, except for the addition of a few more functions.

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.

```
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;
```

Figure 271, Sample View DDL - Scalar functions

Below are the view contents:

```
D1       F1          S1   C1       V1       TS1
------   ---------   --   ------   ------   --------------------------
  -2.4   -2.4e+000   -2   ABCDEF   ABCDEF   1996-04-22-23.58.58.123456
   0.0    0.0e+000    0   ABCD     ABCD     1996-08-15-15.15.15.151515
   1.8    1.8e+000    1   AB       AB       0001-01-01-00.00.00.000000


DT1          TM1        TC1
----------   --------   --------------------------
04/22/1996   23:58:58   1996-04-22-23.58.58.123456
08/15/1996   15:15:15   1996-08-15-15.15.15.151515
01/01/0001   00:00:00   0001-01-01-00.00.00.000000
```

Figure 272, SCALAR view, contents (3 rows)

Scalar Functions, Definitions

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).

```
SELECT D1        AS D1                    ANSWER (float output shortened)
      ,ABS(D1)  AS D2                     ================================
      ,F1        AS F1                    D1    D2   F1          F2
      ,ABS(F1)  AS F2                     ----  ---  ----------  ---------
FROM   SCALAR;                            -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
```

Figure 273, ABS function examples

ACOS

Returns the arccosine of the argument as an angle expressed in radians. The output format is double.

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.

```
SELECT C1                                          ANSWER
      ,ASCII(C1)           AS AC1                  ================
      ,ASCII(SUBSTR(C1,2)) AS AC2                  C1      AC1  AC2
FROM   SCALAR                                      ------  ---  ---
WHERE  C1 = 'ABCDEF';                              ABCDEF   65   66
```

Figure 274, ASCII function examples

The CHR function is the inverse of the ASCII function.

ASIN

Returns the arcsine of the argument as an angle expressed in radians. The output format is double.

ATAN

Returns the arctangent of the argument as an angle expressed in radians. The output format is double.

ATANH

Returns the hyperbolic acrctangent of the argument, where the argument is and an angle expressed in radians. The output format is double.

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.

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.

```
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
```

Figure 275, BIGINT function example

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:

```
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;
```

Figure 276, Convert FLOAT to DECIMAL and BIGINT, SQL

```
FLOAT1                  DECIMAL1             BIGINT1
----------------------  -------------------  --------------------
+1.23456789000000E+000                   1.                     1
+1.23456789000000E+002                 123.                   123
+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
```

Figure 277, Convert FLOAT to DECIMAL and BIGINT, answer

See page 329 for a discussion on floating-point number manipulation.

BLOB

Converts the input (1st argument) to a blob. The output length (2nd argument) is optional.

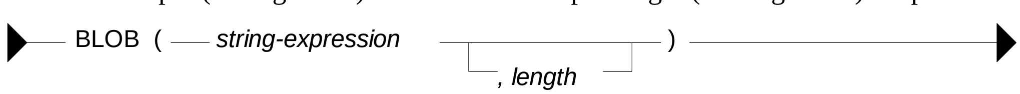

Figure 278, BLOB function syntax

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.

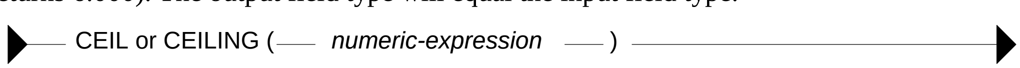

Figure 279, CEILING function syntax

```
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
```

Figure 280, CEIL function examples

NOTE: 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.

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 is a syntax diagram:

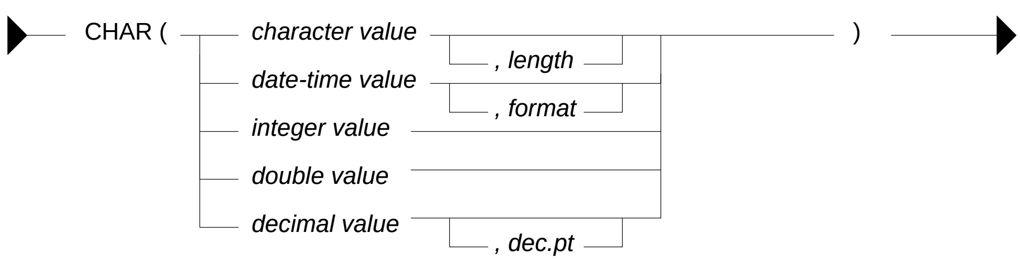

Figure 281, CHAR function syntax

Below are some examples of the CHAR function in action:

```
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        -  -        -
```

Figure 282, CHAR function examples - characters and numbers

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:

```
                                ANSWER
                                ==========================================
                                INT      CHAR_INT CHAR_FLT    CHAR_DEC
                                -------- -------- ----------- ------------
WITH TEMP1 (N) AS                      3 3        3.0E0       00000000003.
(VALUES (3)                            9 9        9.0E0       00000000009.
 UNION ALL                            81 81       8.1E1       00000000081.
 SELECT  N * N                      6561 6561     6.561E3     00000006561.
 FROM    TEMP1                  43046721 43046721 4.3046721E7 00043046721.
 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;
```

Figure 283, CHAR function examples - positive numbers

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:

```
WITH TEMP1 (N1, N2) AS                 ANSWER
(VALUES (SMALLINT(+3)                  ===================================
       ,SMALLINT(-7))                  N1     I1    I2     D1      D2
 UNION ALL                             ------ ----- ------ ------- -------
 SELECT  N1 * N2                            3 3     +3     00003.  +00003.
        ,N2                               -21 -21   -21    -00021. -00021.
 FROM    TEMP1                            147 147   +147   00147.  +00147.
 WHERE   N1 < 300                       -1029 -1029 -1029  -01029. -01029.
)                                        7203 7203  +7203  07203.  +07203.
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;
```

Figure 284, Align CHAR function output - numbers

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.

```
SELECT CHAR(HIREDATE,ISO)                ANSWER
      ,CHAR(HIREDATE,USA)                ================================
      ,CHAR(HIREDATE,EUR)                1          2          3
FROM   EMPLOYEE                          ---------- ---------- ----------
WHERE  LASTNAME < 'C'                    1972-02-12 02/12/1972 12.02.1972
ORDER BY 2;                              1966-03-03 03/03/1966 03.03.1966
```

Figure 285, CHAR function examples - dates

WARNING: Observe that the above data is in day, month, and year (2nd column) order. Had the ORDER BY been on the 1st column (with the ISO output format), the row sequencing would have been different.

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:

```
SELECT   D2                                    ANSWER
        ,CHAR(D2)   AS CD2                     ================
        ,DIGITS(D2) AS DD2                     D2   CD2    DD2
FROM    (SELECT DEC(D1,4,1) AS D2              ---- ------ ----
         FROM   SCALAR                         -2.4 -002.4 0024
        )AS XXX                                 0.0 000.0  0000
ORDER BY 1;                                     1.8 001.8  0018
```

Figure 286, DIGITS vs. CHAR

NOTE: Neither the DIGITS nor the CHAR function do a great job of converting numbers to characters. See page 300 for some user-defined functions that can be used instead.

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.

```
SELECT 'A'               AS "C"               ANSWER
      ,ASCII('A')        AS "C>N"             =================
      ,CHR(ASCII('A'))   AS "C>N>C"           C  C>N  C>N>C  NL
      ,CHR(333)          AS "NL"              -  ---  -----  --
FROM   STAFF                                  A   65  A      ÿ
WHERE  ID = 10;
```

Figure 287, CHR function examples

NOTE: 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.

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:

```
SELECT C1                                 ANSWER
      ,CLOB(C1)   AS CC1                  ===================
      ,CLOB(C1,3) AS CC2                  C1      CC1     CC2
FROM   SCALAR;                            ------  ------  ---
                                          ABCDEF  ABCDEF  ABC
                                          ABCD    ABCD    ABC
                                          AB      AB      AB
```

Figure 288, CLOB function examples

NOTE: At present, the DB2BATCH command processor dies a nasty death whenever it encounters a clob field in the output.

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.

```
SELECT   ID                                          ANSWER
        ,COMM                                        ==================
        ,COALESCE(COMM,0)                            ID  COMM    3
FROM     STAFF                                       --  ------  ------
WHERE    ID < 30                                     10       -    0.00
ORDER BY ID;                                         20  612.45  612.45
```

Figure 289, COALESCE function example

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:

```
WITH TEMP1(C1,C2,C3) AS                              ANSWER
(VALUES (CAST(NULL AS SMALLINT)                      ========
        ,CAST(NULL AS SMALLINT)                      CC1  CC2
        ,CAST(10   AS SMALLINT)))                    ---  ---
SELECT COALESCE(C1,C2,C3) AS CC1                      10   10
      ,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;
```

Figure 290, COALESCE and equivalent CASE expression

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:

```
SELECT COUNT(*)              AS #ROWS          ANSWER
      ,MIN(ID)               AS MIN_ID         ===================
      ,COALESCE(MIN(ID),-1)  AS CCC_ID         #ROWS MIN_ID CCC_ID
FROM   STAFF                                   ----- ------ ------
WHERE  ID < 5;                                     0      -     -1
```

Figure 291, NOT NULL field returning null value

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:

```
SELECT    'A' || 'B'                             ANSWER
         ,'A' CONCAT 'B'                         ===================
         ,CONCAT('A','B')                         1   2   3   4   5
         ,'A' || 'B' || 'C'                      --- --- --- --- ---
         ,CONCAT(CONCAT('A','B'),'C')            AB  AB  AB  ABC ABC
FROM      STAFF
WHERE     ID = 10;
```

Figure 292, CONCAT function examples

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:

```
WITH TEMP1 (COL1,  COL2) AS                              ANSWER
(VALUES     ('A' , 'YYY')                                ===============
           ,('AE', 'OOO')                                COL1 COL2 COL3
           ,('AE', 'YYY')                                ---- ---- -----
)                                                        AE   OOO  AEOOO
SELECT     COL1                                          AE   YYY  AEYYY
          ,COL2                                          A    YYY  AYYY
          ,COL1 CONCAT COL2 AS COL3
FROM       TEMP1
ORDER BY   COL3;
```

Figure 293, CONCAT used with ORDER BY - wrong output sequence

Converting the fields being concatenated to character gets around this problem:

```
WITH TEMP1 (COL1,  COL2) AS                              ANSWER
(VALUES     ('A' , 'YYY')                                ===============
           ,('AE', 'OOO')                                COL1 COL2 COL3
           ,('AE', 'YYY')                                ---- ---- -----
)                                                        A    YYY  A YYY
SELECT     COL1                                          AE   OOO  AEOOO
          ,COL2                                          AE   YYY  AEYYY
          ,CHAR(COL1,2) CONCAT
           CHAR(COL2,3) AS COL3
FROM       TEMP1
ORDER BY   COL3;
```

Figure 294, CONCAT used with ORDER BY - correct output sequence

WARNING: 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).

COS

Returns the cosine of the argument where the argument is an angle expressed in radians. The output format is double.

```
WITH TEMP1(N1) AS                                ANSWER
(VALUES (0)                                      =======================
 UNION ALL                                       N1  RAN    COS    SIN
 SELECT  N1 + 10                                 --  -----  -----  -----
 FROM    TEMP1                                    0  0.000  1.000  0.000
 WHERE   N1 < 90)                                10  0.174  0.984  0.173
SELECT N1                                        20  0.349  0.939  0.342
      ,DEC(RADIANS(N1),4,3)      AS RAN          30  0.523  0.866  0.500
      ,DEC(COS(RADIANS(N1)),4,3) AS COS          40  0.698  0.766  0.642
      ,DEC(SIN(RADIANS(N1)),4,3) AS SIN          50  0.872  0.642  0.766
FROM   TEMP1;                                    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
```

Figure 295, RADIAN, COS, and SIN functions example

COSH

Returns the hyperbolic cosine for the argument, where the argument is an angle expressed in radians. The output format is double.

COT

Returns the cotangent of the argument where the argument is an angle expressed in radians. The output format is double.

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").

```
>>── DATE ( ── expression ── ) ──────────────────────────────><
```

Figure 296, DATE function syntax

If the input can be null, the output will also support null. Null values convert to null output.

```
SELECT TS1                          ANSWER
      ,DATE(TS1) AS DT1             ======================================
FROM   SCALAR;                      TS1                         DT1
                                    --------------------------  ----------
                                    1996-04-22-23.58.58.123456  04/22/1996
                                    1996-08-15-15.15.15.151515  08/15/1996
                                    0001-01-01-00.00.00.000000  01/01/0001
```

Figure 297, DATE function example - timestamp input

```
WITH TEMP1(N1) AS                                      ANSWER
(VALUES   (000001)                                     ===================
         ,(728000)                                     N1       D1
         ,(730120))                                    -------  ----------
SELECT N1                                                    1  01/01/0001
      ,DATE(N1) AS D1                                   728000  03/13/1994
FROM   TEMP1;                                           730120  01/01/2000
```

Figure 298, DATE function example - numeric input

DAY

Returns the day (as in day of the month) part of a date (or equivalent) value. The output format is integer.

```
SELECT DT1                                                ANSWER
      ,DAY(DT1) AS DAY1                                   ================
FROM   SCALAR                                             DT1         DAY1
WHERE  DAY(DT1) > 10;                                     ----------  ----
                                                          04/22/1996    22
                                                          08/15/1996    15
```

Figure 299, DAY function examples

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:

```
SELECT   DT1                                ANSWER
        ,DAY(DT1)                AS DAY1     =========================
        ,DT1 -'1996-04-30'       AS DUR2     DT1        DAY1 DUR2 DAY2
        ,DAY(DT1 -'1996-04-30') AS DAY2     ---------- ---- ---- ----
FROM     SCALAR                              04/22/1996   22  -8.   -8
WHERE    DAY(DT1) > 10                       08/15/1996   15 315.   15
ORDER BY DT1;
```

Figure 300, DAY function, using date-duration input

NOTE: 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.

DAYNAME

Returns the name of the day (e.g. Friday) as contained in a date (or equivalent) value. The output format is varchar(100).

```
SELECT DT1                                   ANSWER
      ,DAYNAME(DT1)          AS DY1          ========================
      ,LENGTH(DAYNAME(DT1)) AS DY2          DT1         DY1      DY2
FROM   SCALAR                                ----------  -------  ---
WHERE  DAYNAME(DT1) LIKE '%a%y'              01/01/0001  Monday     6
ORDER  BY DT1;                               04/22/1996  Monday     6
                                             08/15/1996  Thursday   8
```

Figure 301, DAYNAME function example

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.

```
SELECT   DT1                                 ANSWER
        ,DAYOFWEEK(DT1) AS DWK               =========================
        ,DAYNAME(DT1)   AS DNM               DT1         DWK  DNM
FROM     SCALAR                              ----------  ---  --------
ORDER BY DWK                                 01/01/0001    2  Monday
        ,DNM;                                04/22/1996    2  Monday
                                             08/15/1996    5  Thursday
```

Figure 302, DAYOFWEEK function example

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 year for ISO week 52.

```
WITH                                                     ANSWER
TEMP1 (N) AS                                             ========================
  (VALUES (0)                                            DATE       DAY  W D WI I
   UNION ALL                                             ---------- --- -- - -- -
   SELECT N+1                                            1999-12-25 Sat 52 7 51 6
   FROM   TEMP1                                          1999-12-26 Sun 53 1 51 7
   WHERE  N < 9),                                        1999-12-27 Mon 53 2 52 1
TEMP2 (DT1) AS                                           1999-12-28 Tue 53 3 52 2
  (VALUES(DATE('1999-12-25'))                            1999-12-29 Wed 53 4 52 3
        ,(DATE('2000-12-24'))),                          1999-12-30 Thu 53 5 52 4
TEMP3 (DT2) AS                                           1999-12-31 Fri 53 6 52 5
  (SELECT DT1 + N DAYS                                   2000-01-01 Sat  1 7 52 6
   FROM   TEMP1                                          2000-01-02 Sun  2 1 52 7
         ,TEMP2)                                         2000-01-03 Mon  2 2  1 1
SELECT   CHAR(DT2,ISO)             AS DATE               2000-12-24 Sun 53 1 51 7
        ,SUBSTR(DAYNAME(DT2),1,3)  AS DAY                2000-12-25 Mon 53 2 52 1
        ,WEEK(DT2)                 AS W                  2000-12-26 Tue 53 3 52 2
        ,DAYOFWEEK(DT2)            AS D                  2000-12-27 Wed 53 4 52 3
        ,WEEK_ISO(DT2)             AS WI                 2000-12-28 Thu 53 5 52 4
        ,DAYOFWEEK_ISO(DT2)        AS I                  2000-12-29 Fri 53 6 52 5
FROM     TEMP3                                           2000-12-30 Sat 53 7 52 6
ORDER BY 1;                                              2000-12-31 Sun 54 1 52 7
                                                         2001-01-01 Mon  1 2  1 1
                                                         2001-01-02 Tue  1 3  1 2
```

Figure 303, DAYOFWEEK_ISO function example

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.

```
SELECT   DT1                                    ANSWER
        ,DAYOFYEAR(DT1) AS DYR                  ===============
FROM     SCALAR                                 DT1         DYR
ORDER BY DYR;                                   ---------- ---
                                                01/01/0001    1
                                                04/22/1996  113
                                                08/15/1996  228
```

Figure 304, DAYOFYEAR function example

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.

```
SELECT   DT1                                    ANSWER
        ,DAYS(DT1) AS DY1                       ==================
FROM     SCALAR                                 DT1         DY1
ORDER BY DY1                                    ---------- ------
        ,DT1;                                   01/01/0001      1
                                                04/22/1996 728771
                                                08/15/1996 728886
```

Figure 305, DAYS function example

The DATE function can act as the inverse of the DAYS function. It can convert the DAYS output back into a valid date.

DBCLOB

Converts the input (1st argument) to a dbclob. The output length (2nd argument) is optional.

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.

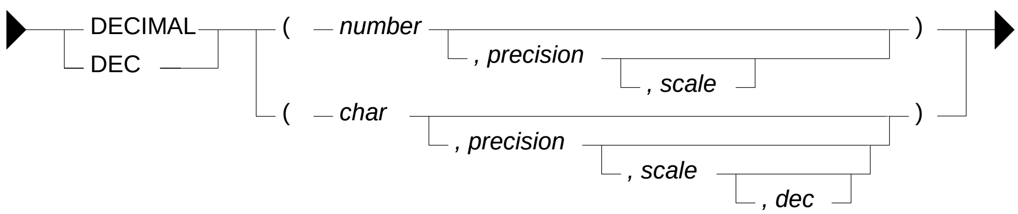

Figure 306, DECIMAL function syntax

```
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
```

Figure 307, DECIMAL function examples

WARNING: Converting a floating-point number to decimal may get different results from converting the same number to integer. See page 329 for a discussion of this issue.

DEGREES

Returns the number of degrees converted from the argument as expressed in radians. The output format is double.

DEREF

Returns an instance of the target type of the argument.

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.

Figure 308, DECRYPT function syntax

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.

```
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;
```

Figure 309, DECRYPT_CHAR function example

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).

```
SELECT   A.NAME           AS N1            ANSWER
        ,SOUNDEX(A.NAME)  AS S1            ==============================
        ,B.NAME           AS N2            N1      S1   N2        S2   DF
        ,SOUNDEX(B.NAME)  AS S2            ------- ---- --------- ---- --
        ,DIFFERENCE                        Sanders S536 Sneider   S536  4
         (A.NAME,B.NAME)  AS DF            Sanders S536 Smith     S530  3
FROM     STAFF A                           Sanders S536 Lundquist L532  2
        ,STAFF B                           Sanders S536 Daniels   D542  1
WHERE    A.ID = 10                         Sanders S536 Molinare  M456  1
  AND    B.ID > 150                        Sanders S536 Scoutten  S350  1
  AND    B.ID < 250                        Sanders S536 Abrahams  A165  0
ORDER BY DF DESC                           Sanders S536 Kermisch  K652  0
        ,N2 ASC;                           Sanders S536 Lu        L000  0
```

Figure 310, DIFFERENCE function example

NOTE: The difference function returns one of five possible values. In many situations, it would imprudent to use a value with such low granularity to rank values.

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.

```
SELECT S1                                       ANSWER
      ,DIGITS(S1) AS DS1                        =========================
      ,D1                                       S1      DS1    D1     DD1
      ,DIGITS(D1) AS DD1                        ------  -----  -----  ---
FROM   SCALAR;                                      -2  00002   -2.4  024
                                                     0  00000    0.0  000
                                                     1  00001    1.8  018
```

Figure 311, DIGITS function examples

The CHAR function can sometimes be used as alternative to the DIGITS function. Their output differs slightly - see page 300 for a comparison.

NOTE: Neither the DIGITS nor the CHAR function do a great job of converting numbers to characters. See page 300 for some user-defined functions that can be used instead.

DLCOMMENT

Returns the comments value, if it exists, from a datalink value.

DLLINKTYPE

Returns the linktype value from a datalink value.

DLURLCOMPLETE

Returns the URL value from a datalink value with a linktype of URL.

DLURLPATH

Returns the path and file name necessary to access a file within a given server from a datalink value with linktype of URL.

DLURLPATHONLY

Returns the path and file name necessary to access a file within a given server from a datalink value with a linktype of URL. The value returned never includes a file access token.

DLURLSCHEME

Returns the scheme from a datalink value with a linktype of URL.

DLURLSERVER

Returns the file server from a datalink value with a linktype of URL.

DLVALUE

Returns a datalink value.

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.

```
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
```

Figure 312, DOUBLE function examples

See page 329 for a discussion on floating-point number manipulation.

ENCRYPT

Returns a encrypted rendition of the input string. The input must be char or varchar. The output is varchar for bit data.

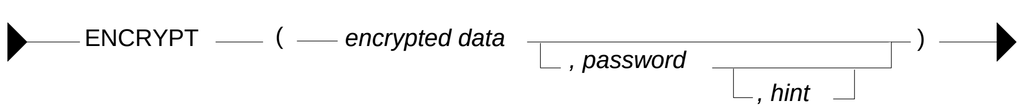

Figure 313, DECRYPT function syntax

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.

```
SELECT   ID
        ,NAME
        ,ENCRYPT(NAME,'THAT IDIOT','MY BROTHER') AS NAME2
FROM     STAFF
WHERE ID < 30
ORDER BY ID;
```

Figure 314, ENCRYPT function example

EVENT_MON_STATE

Returns an operational state of a particular event monitor.

EXP

Returns the exponential function of the argument. The output format is double.

```
WITH TEMP1(N1) AS                         ANSWER
(VALUES (0)                               ==============================
 UNION ALL                                N1 E1                    E2
 SELECT  N1 + 1                           -- --------------------- -----
 FROM    TEMP1                             0  +1.00000000000000E+0     1
 WHERE   N1 < 10)                          1  +2.71828182845904E+0     2
SELECT N1                                  2  +7.38905609893065E+0     7
      ,EXP(N1) AS E1                       3  +2.00855369231876E+1    20
      ,SMALLINT(EXP(N1)) AS E2             4  +5.45981500331442E+1    54
FROM   TEMP1;                              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
```

Figure 315, EXP function examples

FLOAT

Same as DOUBLE.

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.

```
SELECT D1                           ANSWER  (float output shortened)
      ,FLOOR(D1) AS D2              ===================================
      ,F1                           D1     D2    F1           F2
      ,FLOOR(F1) AS F2              -----  ----  ----------  ----------
FROM   SCALAR;                       -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
```

Figure 316, FLOOR function examples

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 GMT, not local time), with the node number appended to the back.

```
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
                         -- -------------- ---------------------------
NOTE: 2ND FIELD =>       10                20011017191648990521000000.
IS UNPRINTABLE. =>       20                20011017191648990615000000.
                         30                20011017191648990642000000.
                         40                20011017191648990669000000.
```

Figure 317, GENERATE_UNIQUE function examples

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.

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.

```
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
```

Figure 318, GENERATE_UNIQUE output, characters reversed to make 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.

GETHINT

Returns the password hint, if one is found in the encrypted data.

```
SELECT   ID
        ,NAME
        ,GETHINT(NAME2) AS HINT
FROM    (SELECT  ID
                ,NAME
                ,ENCRYPT(NAME,'THAT IDIOT','MY BROTHER') AS NAME2
         FROM    STAFF
         WHERE ID < 30                                           ANSWER
        )AS XXX                                     =====================
ORDER BY ID;                                        ID NAME    HINT
                                                    -- ------- ----------
                                                    10 Sanders MY BROTHER
                                                    20 Pernal  MY BROTHER
```

Figure 319, GETHINT function example

GRAPHIC

Converts the input (1st argument) to a graphic data type. The output length (2nd argument) is optional.

HEX

Returns the hexadecimal representation of a value. All input types are supported.

```
WITH TEMP1(N1) AS                            ANSWER
(VALUES (-3)                                 ===============================
 UNION ALL                                   S  SHX  DHX    FHX
 SELECT  N1 + 1                              -- ---- ------ ----------------
 FROM    TEMP1                               -3 FDFF 00003D 00000000000008C0
 WHERE   N1 < 3)                             -2 FEFF 00002D 00000000000000C0
SELECT SMALLINT(N1)       AS S               -1 FFFF 00001D 000000000000F0BF
      ,HEX(SMALLINT(N1))  AS SHX              0 0000 00000C 0000000000000000
      ,HEX(DEC(N1,4,0))   AS DHX              1 0100 00001C 000000000000F03F
      ,HEX(DOUBLE(N1))    AS FHX              2 0200 00002C 0000000000000040
FROM   TEMP1;                                 3 0300 00003C 0000000000000840
```

Figure 320, HEX function examples, numeric data

```
SELECT C1                          ANSWER
      ,HEX(C1) AS CHX              =======================================
      ,V1                          C1     CHX          V1     VHX
      ,HEX(V1) AS VHX              ------ ------------ ------ ------------
FROM   SCALAR;                     ABCDEF 414243444546 ABCDEF 414243444546
                                   ABCD   414243442020 ABCD   41424344
                                   AB     414220202020 AB     4142
```

Figure 321, HEX function examples, character & varchar

```
SELECT DT1                             ANSWER
      ,HEX(DT1) AS DTHX                ===================================
      ,TM1                             DT1        DTHX     TM1      TMHX
      ,HEX(TM1) AS TMHX                ---------- -------- -------- ------
FROM   SCALAR;                         04/22/1996 19960422 23:58:58 235858
                                       08/15/1996 19960815 15:15:15 151515
                                       01/01/0001 00010101 00:00:00 000000
```

Figure 322, HEX function examples, date & time

HOUR

Returns the hour (as in hour of day) part of a time value. The output format is integer.

```
SELECT   TM1                                                 ANSWER
        ,HOUR(TM1) AS HR                                     ============
FROM     SCALAR                                              TM1       HR
ORDER BY TM1;                                                -------- --
                                                             00:00:00   0
                                                             15:15:15  15
                                                             23:58:58  23
```

Figure 323, HOUR function example

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 page 235 for detailed notes on using this function.

```
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);
                                                                   ANSWER
WITH TEMP (IDVAL) AS                                               ======
(VALUES (IDENTITY_VAL_LOCAL()))                                     IDVAL
SELECT *                                                            -----
FROM   TEMP;                                                           1.
```

Figure 324, IDENTITY_VAL_LOCAL function usage

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.

```
INSERT ( source , start-pos , del-bytes , new-value )
```

Figure 325, INSERT function syntax

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.

```
SELECT NAME                        ANSWER (4K output fields shortened)
      ,INSERT(NAME,3,2,'A')        ===================================
      ,INSERT(NAME,3,2,'AB')       NAME     2       3        4
      ,INSERT(NAME,3,2,'ABC')      -------- ------- -------- ---------
FROM   STAFF                       Sanders  SaAers  SaABers  SaABCers
WHERE  ID < 40;                    Pernal   PeAal   PeABal   PeABCal
                                   Marenghi MaAnghi MaABnghi MaABCnghi
```

Figure 326, INSERT function examples

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.

```
SELECT D1                         ANSWER
      ,INTEGER(D1)                ====================================
      ,INT('+123')                D1     2      3       4       5
      ,INT('-123')                ----- -----  ------  ------  ------
      ,INT(' 123 ')                -2.4    -2     123    -123     123
FROM   SCALAR;                      0.0     0     123    -123     123
                                    1.8     1     123    -123     123
```

Figure 327, INTEGER function examples

JULIAN_DAY

Converts a date (or equivalent) value into a number which represents the number of days since January the 1st, 4,713 BC. The output format is integer.

```
WITH TEMP1(DT1) AS                           ANSWER
(VALUES ('0001-01-01-00.00.00')              =========================
       ,('1752-09-10-00.00.00')              DT         DY     DJ
       ,('1993-01-03-00.00.00')              ---------- ------ -------
       ,('1993-01-03-23.59.59'))             01/01/0001      1 1721426
SELECT DATE(DT1) AS DT                       09/10/1752 639793 2361218
      ,DAYS(DT1) AS DY                       01/03/1993 727566 2448991
      ,JULIAN_DAY(DT1) AS DJ                 01/03/1993 727566 2448991
FROM   TEMP1;
```

Figure 328, JULIAN_DAY function example

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 2000-01-01 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.

```
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
---------- ------- -------
05/17/1926 2424653 2424653
03/28/1936 2428256 2428256
07/09/1946 2432011 2432011
04/12/1955 2435210 2435210
```

Figure 329, JULIAN_DAY function examples

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):

```
WITH TEMP1(DT1) AS
(VALUES ('1997-01-01')
       ,('1997-01-02')
       ,('1997-12-31'))
SELECT DATE(DT1) AS DT
      ,DATE(DT1) - 15 DAYS AS DJ1
      ,YEAR(DT1) * 1000 + DAYOFYEAR(DT1) AS DJ2
FROM   TEMP1;
```

```
ANSWER
=============================
DT         DJ1        DJ2
---------- ---------- -------
01/01/1997 12/17/1996 1997001
01/02/1997 12/18/1996 1997002
12/31/1997 12/16/1997 1997365
```

Figure 330, Julian Date outputs

WARNING: DB2 does not make allowances for the days that were not used when English-speaking countries converted from the Julian to the Gregorian calendar in 1752

LCASE or LOWER

Coverts a mixed or upper-case string to lower case. The output is the same data type and length as the input.

```
SELECT NAME                               ANSWER
      ,LCASE(NAME) AS LNAME               =========================
      ,UCASE(NAME) AS UNAME               NAME     LNAME    UNAME
FROM   STAFF                              -------  -------  -------
WHERE  ID < 30;                           Sanders  sanders  SANDERS
                                          Pernal   pernal   PERNAL
```

Figure 331, LCASE function example

Documentation Comment

According to the DB2 UDB V8.1 SQL Reference, the LCASE and UCASE functions are the inverse of each other for the standard alphabetical characters, "a" to "z", but not for some odd European characters. Therefore LCASE(UCASE(string)) may not equal LCASE(string).

This may be true from some code pages, but it is not for the one that I use. The following recursive SQL illustrates the point. It shows that for every ASCII character, the use of both functions gives the same result as the use of just one:

```
WITH TEMP1 (N1,C1) AS                                  ANSWER
(VALUES (SMALLINT(0),CHR(0))                           =================
 UNION ALL                                             N1 C1 U1 U2 L1 L2
 SELECT N1 + 1                                         -- -- -- -- -- --
       ,CHR(N1 + 1)                                    <no rows>
 FROM   TEMP1
 WHERE  N1 < 255
)
SELECT  N1
       ,C1
       ,UCASE(C1)          AS U1
       ,UCASE(LCASE(C1))   AS U2
       ,LCASE(C1)          AS L1
       ,LCASE(UCASE(C1))   AS L2
FROM    TEMP1
WHERE   UCASE(C1) <> UCASE(LCASE(C1))
   OR   LCASE(C1) <> LCASE(UCASE(C1));
```

Figure 332, LCASE and UCASE usage on special characters

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.

```
WITH TEMP1(C1) AS                                       ANSWER
(VALUES ('  ABC')                                       ================
       ,(' ABC ')                                       C1     C2     L2
       ,('ABC  '))                                      -----  -----  --
SELECT C1                                                 ABC    AB    4
      ,LEFT(C1,4) AS C2                                  ABC    ABC    4
      ,LENGTH(LEFT(C1,4)) AS L2                         ABC    ABC     4
FROM   TEMP1;
```

Figure 333, LEFT function examples

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.

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.

```
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
```

Figure 334, LENGTH function examples

LN or LOG

Returns the natural logarithm of the argument (same as LOG). The output format is double.

```
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
```

Figure 335, LOG function example

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.

```
>>── LOCATE ( ──find-string ──, look-in-string ──┬──────────────┬── ) ──────><
                                                 └─, start-pos.─┘
```

Figure 336, LOCATE function syntax

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).

```
SELECT C1
      ,LOCATE('D', C1)
      ,LOCATE('D', C1,2)
      ,LOCATE('EF',C1)
      ,LOCATE('A', C1,2)
FROM   SCALAR;
```

```
ANSWER
==========================
C1      2    3    4    5
------  ---  ---  ---  ---
ABCDEF    4    4    5    0
ABCD      4    4    0    0
AB        0    0    0    0
```

Figure 337, LOCATE function examples

LOG or LN

See the description of the LN function.

LOG10

Returns the base ten logarithm of the argument. The output format is double.

```
WITH TEMP1(N1) AS                     ANSWER
(VALUES (1),(123),(1234)              ===============================
       ,(12345),(123456))             N1      L1
SELECT N1                             ------  -----------------------
      ,LOG10(N1) AS L1                     1   +0.00000000000000E+000
FROM   TEMP1;                            123   +2.08990511143939E+000
                                        1234   +3.09131515969722E+000
                                       12345   +4.09149109426795E+000
                                      123456   +5.09151220162777E+000
```

Figure 338, LOG10 function example

LONG_VARCHAR

Converts the input (1st argument) to a long_varchar data type. The output length (2nd argument) is optional.

LONG_VARGRAPHIC

Converts the input (1st argument) to a long_vargraphic data type. The output length (2nd argument) is optional.

LOWER

See the description for the LCASE function.

LTRIM

Remove leading blanks, but not trailing blanks, from the argument.

```
WITH TEMP1(C1) AS                                    ANSWER
(VALUES ('  ABC')                                    ================
       ,(' ABC ')                                    C1     C2     L2
       ,('ABC  '))                                   -----  -----  --
SELECT C1                                              ABC  ABC     3
      ,LTRIM(C1) AS C2                                  ABC  ABC     4
      ,LENGTH(LTRIM(C1)) AS L2                       ABC    ABC     5
FROM   TEMP1;
```

Figure 339, LTRIM function example

MICROSECOND

Returns the microsecond part of a timestamp (or equivalent) value. The output is integer.

```
SELECT TS1                       ANSWER
      ,MICROSECOND(TS1)          ======================================
FROM   SCALAR                    TS1                        2
ORDER BY TS1;                    -------------------------- -----------
                                 0001-01-01-00.00.00.000000           0
                                 1996-04-22-23.58.58.123456      123456
                                 1996-08-15-15.15.15.151515      151515
```

Figure 340, MICROSECOND function example

MIDNIGHT_SECONDS

Returns the number of seconds since midnight from a timestamp, time or equivalent value. The output format is integer.

```
SELECT TS1                          ANSWER
      ,MIDNIGHT_SECONDS(TS1)        ======================================
      ,HOUR(TS1)*3600 +             TS1                        2     3
       MINUTE(TS1)*60 +             -------------------------- ----- -----
       SECOND(TS1)                  0001-01-01-00.00.00.000000     0     0
FROM   SCALAR                       1996-04-22-23.58.58.123456 86338 86338
ORDER BY TS1;                       1996-08-15-15.15.15.151515 54915 54915
```

Figure 341, MIDNIGHT_SECONDS function example

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:

```
                                                         ANSWER
                                                         ==============
                                                         MS    TM
                                                         ----- --------
WITH TEMP1 (MS) AS                                           0 00:00:00
(SELECT MIDNIGHT_SECONDS(TS1)                            54915 15:15:15
 FROM   SCALAR                                           86338 23:58:58
)
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;
```

Figure 342, Convert MIDNIGHT_SECONDS output back to a time value

NOTE: Imagine a column with two timestamp values: "1996-07-15.24.00.00" and "1996-07-16.00.00.00". These two values represent the same point in time, but will return different MIDNIGHT_SECONDS results. See the chapter titled "Quirks in SQL" on page 319 for a detailed discussion of this problem.

MINUTE

Returns the minute part of a time or timestamp (or equivalent) value. The output is integer.

```
SELECT TS1                          ANSWER
      ,MINUTE(TS1)                  ======================================
FROM   SCALAR                       TS1                        2
ORDER BY TS1;                       -------------------------- -----------
                                    0001-01-01-00.00.00.000000           0
                                    1996-04-22-23.58.58.123456          58
                                    1996-08-15-15.15.15.151515          15
```

Figure 343, MINUTE function example

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.

```
WITH TEMP1(N1,N2) AS                                  ANSWER
(VALUES (-31,+11)                                     =======================
 UNION ALL                                            N1   N2   DIV  MD1  MD2
 SELECT  N1 + 13                                      ---  ---  ---  ---  ---
        ,N2 - 4                                       -31   11   -2   -9   -9
 FROM    TEMP1                                        -18    7   -2   -4   -4
 WHERE   N1 < 60                                       -5    3   -1   -2   -2
)                                                       8   -1   -8    0    0
SELECT   N1                                            21   -5   -4    1    1
        ,N2                                            34   -9   -3    7    7
        ,N1/N2            AS DIV                       47  -13   -3    8    8
        ,N1-((N1/N2)*N2)  AS MD1                       60  -17   -3    9    9
        ,MOD(N1,N2)       AS MD2
FROM     TEMP1
ORDER BY 1;
```

Figure 344, MOD function example

MONTH

Returns an integer value in the range 1 to 12 that represents the month part of a date or time-stamp (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).

```
SELECT DT1                                            ANSWER
      ,MONTH(DT1)                                     =======================
      ,MONTHNAME(DT1)                                 DT1         2   3
FROM   SCALAR                                         ----------  --  -------
ORDER BY DT1;                                         01/01/0001   1  January
                                                      04/22/1996   4  April
                                                      08/15/1996   8  August
```

Figure 345, MONTH and MONTHNAME functions example

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).

```
WITH TEMP1 (N1,N2) AS
(VALUES (DECIMAL(1234,10)                                             ANSWER
        ,DECIMAL(1234,10)))                                          ========
SELECT  N1                                                  >>        1234.
       ,N2                                                  >>        1234.
       ,N1 * N2            AS P1                            >>     1522756.
       ,"*"(N1,N2)         AS P2                            >>     1522756.
       ,MULTIPLY_ALT(N1,N2) AS P3                           >>     1522756.
FROM    TEMP1;
```

Figure 346, Multiplying numbers - examples

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:

```
                                                              <--MULTIPLY_ALT->
                                  RESULT       RESULT          SCALE    PRECSION
 INPUT#1     INPUT#2       "*" OPERATOR   MULTIPLY_ALT      TRUNCATD 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
```

Figure 347, Decimal multiplication - same output lengths

NODENUMBER

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.

NODENUMBER (*column-name*)

Figure 348, NODENUMBER function syntax

```
SELECT   NODENUMBER(ID) AS NN                                   ANSWER
FROM     STAFF                                                  ======
WHERE    ID = 10;                                               NN
                                                                --
                                                                 0
```

Figure 349, NODENUMBER function example

The NODENUMBER 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.

NULLIF

Returns null if the two values being compared are equal, otherwise returns the first value.

```
SELECT S1                                        ANSWER
      ,NULLIF(S1,0)                              =====================
      ,C1                                        S1  2   C1     4
      ,NULLIF(C1,'AB')                           --- --- ------ ------
FROM   SCALAR                                     -2  -2 ABCDEF ABCDEF
WHERE  NULLIF(0,0) IS NULL;                        0   - ABCD   ABCD
                                                   1   1 AB     -
```

Figure 350, NULLIF function examples

PARTITION

Returns the partition map index of the row. The result is zero if the table is not partitioned. The output is of type integer, and is never null.

```
SELECT   PARTITION(ID) AS PP                                         ANSWER
FROM     STAFF                                                       ======
WHERE    ID = 10;                                                    PP
                                                                     --
                                                                      0
```

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.

```
SELECT C1                                         ANSWER
      ,POSSTR(C1,' ')  AS P1                      ==================
      ,POSSTR(C1,'CD') AS P2                      C1      P1  P2  P3
      ,POSSTR(C1,'cd') AS P3                      ------  --  --  --
FROM   SCALAR                                     AB       3   0   0
ORDER BY 1;                                       ABCD     5   3   0
                                                  ABCDEF   0   3   0
```

Figure 351, POSSTR function examples

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.

```
SELECT C1                                ANSWER
      ,POSSTR(C1,' ')    AS P1           ===========================
      ,LOCATE(' ',C1)    AS L1           C1      P1 L1 P2 L2 P3 L3 L4
      ,POSSTR(C1,'CD')   AS P2           ------ -- -- -- -- -- -- --
      ,LOCATE('CD',C1)   AS L2           AB      3  3  0  0  0  0  0
      ,POSSTR(C1,'cd')   AS P3           ABCD    5  5  3  3  0  0  4
      ,LOCATE('cd',C1)   AS L3           ABCDEF  0  0  3  3  0  0  4
      ,LOCATE('D',C1,2)  AS L4
FROM   SCALAR
ORDER BY 1;
```

Figure 352, POSSTR vs. LOCATE functions

POWER

Returns the value of the first argument to the power of the second argument

```
WITH TEMP1(N1) AS                        ANSWER
(VALUES (1),(10),(100))                  ===============================
SELECT N1                                N1      P1      P2      P3
      ,POWER(N1,1) AS P1                 ------- ------- ------- -------
      ,POWER(N1,2) AS P2                       1       1       1       1
      ,POWER(N1,3) AS P3                      10      10     100    1000
FROM   TEMP1;                                100     100   10000 1000000
```

Figure 353, POWER function examples

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.

RADIANS

Returns the number of radians converted from the input, which is expressed in degrees. The output format is double.

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── ( ── sqlstate ── ,error-message── ) ──────────►
```

Figure 354, RAISE_ERROR function syntax

```
SELECT S1                                               ANSWER
      ,CASE                                             ==============
        WHEN S1 < 1 THEN S1                             S1      S2
        ELSE RAISE_ERROR('80001',C1)                    ------  ------
       END AS S2                                            -2      -2
FROM   SCALAR;                                               0       0
                                                        SQLSTATE=80001
```

Figure 355, RAISE_ERROR function example

RAND

WARNING: Using the RAND function in a predicate can result in unpredictable results. See page 322 for a detailed description of this issue.

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:

```
WITH TEMP (NUM, RAN) AS
(VALUES (INT(1)
        ,RAND(2))
 UNION ALL
 SELECT  NUM + 1
        ,RAND()
 FROM    TEMP
 WHERE   NUM < 100000                                      ANSWER
)                                                          =============
SELECT  COUNT(*)              AS #ROWS           ==>       100000
       ,COUNT(DISTINCT RAN)   AS #VALUES         ==>        31242
       ,DEC(AVG(RAN),7,6)     AS AVG_RAN         ==>            0.499838
       ,DEC(STDDEV(RAN),7,6)  AS STD_DEV                        0.288706
       ,DEC(MIN(RAN),7,6)     AS MIN_RAN                        0.000000
       ,DEC(MAX(RAN),7,6)     AS MAX_RAN                        1.000000
       ,DEC(MAX(RAN),7,6) -
        DEC(MIN(RAN),7,6)     AS RANGE                          1.000000
       ,DEC(VAR(RAN),7,6)     AS VARIANCE                       0.083351
FROM    TEMP;
```

Figure 356, Sample output from RAND function

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.

WARNING: 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):

```
SELECT   DEPTNO  AS DNO                      ANSWER
        ,RAND(0) AS RAN                      ===========================
FROM     DEPARTMENT                          DNO  RAN
WHERE    DEPTNO < 'E'                        ---  ----------------------
ORDER BY 1;                                  A00  +1.15970336008789E-003
                                             B01  +2.35572374645222E-001
                                             C01  +6.48152104251228E-001
                                             D01  +7.43736075930052E-002
                                             D11  +2.70241401409955E-001
                                             D21  +3.60026856288339E-001
```

Figure 357, Make reproducible random numbers (use seed)

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:

```
SELECT   DEPTNO  AS DNO                      ANSWER
        ,RAND()  AS RAN                      ===========================
FROM     DEPARTMENT                          DNO  RAN
WHERE    DEPTNO < 'D'                        ---  ----------------------
ORDER BY 1;                                  A00  +2.55287331766717E-001
                                             B01  +9.85290078432569E-001
                                             C01  +3.18918424024171E-001
```

Figure 358, Make non-reproducible random numbers (no seed)

NOTE: 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.

```
WITH TEMP1 (COL1, COL2, COL3) AS                     ANSWER
(VALUES (0                                           ===================
        ,SMALLINT(RAND(2)*35)*10                     COL1  COL2  COL3
        ,DECIMAL(RAND()*10000,7,2))                  ----  ----  -------
 UNION ALL                                              0     0  9342.32
 SELECT  COL1 + 1                                       1   250  8916.28
        ,SMALLINT(RAND()*35)*10                         2   310  5430.76
        ,DECIMAL(RAND()*10000,7,2)                      3   150  5996.88
 FROM   TEMP1                                           4   110  8066.34
 WHERE  COL1 + 1 < 10                                   5    50  5589.77
)                                                       6   130  8602.86
SELECT *                                                7   340   184.94
FROM   TEMP1;                                           8   310  5441.14
                                                        9    70  9267.55
```

Figure 359, Use RAND to make sample data

NOTE: 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:

```
WITH TEMP1 (COL1,RAN1,RAN2) AS                       ANSWER
(VALUES (0                                           ===================
        ,RAND(2)                                     COL#1  RAN#1  RAN#2
        ,RAND()+(RAND()/1E5) )                       -----  -----  -----
 UNION ALL                                           30000  19698  29998
 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;
```

Figure 360, Use RAND to make many distinct random values

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.

NOTE: 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 page 116), these values can also be made random.

Selecting Random Rows, Percentage

WARNING: Using the RAND function in a predicate can result in unpredictable results. See page 322 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:

```
SELECT   ID                                                   ANSWER
        ,NAME                                                 ============
FROM     STAFF                                                ID  NAME
WHERE    RAND() < 0.1                                         --- --------
ORDER BY ID;                                                  140 Fraye
                                                              190 Sneider
                                                              290 Quill
```

Figure 361, Randomly select 10% of matching rows

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 nested table expression) by using the ROW_NUMBER function to assign row numbers to the matching rows in random order (using the RAND function). Subsequently, those rows with the five lowest row numbers are selected:

```
SELECT   ID                                                   ANSWER
        ,NAME                                                 ============
FROM    (SELECT S.*                                           ID  NAME
               ,ROW_NUMBER() OVER(ORDER BY RAND()) AS R       --- --------
         FROM   STAFF S                                        10 Sanders
        )AS XXX                                                30 Marenghi
WHERE    R <= 5                                               190 Sneider
ORDER BY ID;                                                  270 Lea
                                                              280 Wilson
```

Figure 362, Select five random rows

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:

```
UPDATE  STAFF
SET     SALARY = RAND()*10000
WHERE   ID < 50;
```

Figure 363, Use RAND to assign random salaries

REAL

Returns a single-precision floating-point representation of a number.

```
                                                    ANSWERS
                                                    ================================
SELECT  N1            AS DEC         =>  1234567890.123456789012345678901
       ,DOUBLE(N1)    AS DBL         =>             1.23456789012346e+009
       ,REAL(N1)      AS REL         =>                     1.234568e+009
       ,INTEGER(N1)   AS INT         =>                        1234567890
       ,BIGINT(N1)    AS BIG         =>                        1234567890
FROM   (SELECT 1234567890.123456789012345678901 AS N1
        FROM   STAFF
        WHERE  ID = 10) AS XXX;
```

Figure 364, REAL and other numeric function examples

REC2XML

Returns a string formatted with XML tags and containing column names and column data.

REPEAT

Repeats a character string "n" times.

```
►─── REPEAT ── ( ─ string-to-repeat ─ , #times ── ) ──────────────────►
```

Figure 365, REPEAT function syntax

```
SELECT   ID                                  ANSWER
        ,CHAR(REPEAT(NAME,3),40)             ===========================
FROM     STAFF                               ID 2
WHERE    ID < 40                             -- ------------------------
ORDER BY ID;                                 10 SandersSandersSanders
                                             20 PernalPernalPernal
                                             30 MarenghiMarenghiMarenghi
```

Figure 366, REPEAT function example

REPLACE

Replaces all occurrences of one string with another. The output is of type varchar(4000).

```
►─── REPLACE ── ( ─ string-to-change ─ , search-for ─ , replace-with ──) ──►
```

Figure 367, REPLACE function syntax

```
SELECT C1                                          ANSWER
      ,REPLACE(C1,'AB','XY') AS R1                 ======================
      ,REPLACE(C1,'BA','XY') AS R2                 C1      R1      R2
FROM   SCALAR;                                     ------  ------  ------
                                                   ABCDEF  XYCDEF  ABCDEF
                                                   ABCD    XYCD    ABCD
                                                   AB      XY      AB
```

Figure 368, REPLACE function examples

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.

```
SELECT C1                                                  ANSWER
      ,REPLACE(REPLACE(                                    ==============
       REPLACE(REPLACE(C1,                                 C1      R1
        'AB','XY'),'ab','XY'),                             ------  ------
        'Ab','XY'),'aB','XY')                              ABCDEF  XYCDEF
FROM   SCALAR;                                             ABCD    XYCD
                                                           AB      XY
```

Figure 369, Nested REPLACE functions

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.

```
WITH TEMP1(C1) AS                                         ANSWER
(VALUES ('  ABC')                                         ================
       ,(' ABC ')                                         C1     C2     L2
       ,('ABC  '))                                        -----  -----  --
SELECT C1                                                   ABC   ABC    4
      ,RIGHT(C1,4) AS C2                                   ABC   ABC     4
      ,LENGTH(RIGHT(C1,4)) AS L2                          ABC    BC      4
FROM   TEMP1;
```

Figure 370, RIGHT function examples

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.

```
                        ANSWER
                        ===============================================
                        D1      P2      P1      P0      N1      N2
                        ------- ------- ------- ------- ------- -------
WITH TEMP1(D1) AS       123.400 123.400 123.400 123.000 120.000 100.000
(VALUES (123.400)        23.450  23.450  23.400  23.000  20.000   0.000
       ,( 23.450)         3.456   3.460   3.500   3.000   0.000   0.000
       ,(  3.456)         0.056   0.060   0.100   0.000   0.000   0.000
       ,(   .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 N2
FROM   TEMP1;
```

Figure 371, ROUND function examples

RTRIM

Trims the right-most blanks of a character string.

```
SELECT C1                                           ANSWER
      ,RTRIM(C1)          AS R1                     ======================
      ,LENGTH(C1)         AS R2                     C1      R1      R2  R3
      ,LENGTH(RTRIM(C1))  AS R3                     ------  ------  --  --
FROM   SCALAR;                                      ABCDEF  ABCDEF   6   6
                                                    ABCD    ABCD     6   4
                                                    AB      AB       6   2
```

Figure 372, RTRIM function example

SECOND

Returns the second (of minute) part of a time or timestamp (or equivalent) value.

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.

```
SELECT D1                       ANSWER  (float output shortened)
      ,SIGN(D1)                 =========================================
      ,F1                       D1     2           F1          4
      ,SIGN(F1)                 -----  ----------  ----------  ----------
FROM   SCALAR;                   -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
```

Figure 373, SIGN function examples

SIN

Returns the SIN of the argument where the argument is an angle expressed in radians. The output format is double.

```
WITH TEMP1(N1) AS                                   ANSWER
(VALUES (0)                                         =======================
 UNION ALL                                          N1  RAN    SIN    TAN
 SELECT  N1 + 10                                    --  -----  -----  -----
 FROM    TEMP1                                       0  0.000  0.000  0.000
 WHERE   N1 < 80)                                   10  0.174  0.173  0.176
SELECT N1                                           20  0.349  0.342  0.363
      ,DEC(RADIANS(N1),4,3)       AS RAN            30  0.523  0.500  0.577
      ,DEC(SIN(RADIANS(N1)),4,3)  AS SIN            40  0.698  0.642  0.839
      ,DEC(TAN(RADIANS(N1)),4,3)  AS TAN            50  0.872  0.766  1.191
FROM   TEMP1;                                       60  1.047  0.866  1.732
                                                    70  1.221  0.939  2.747
                                                    80  1.396  0.984  5.671
```

Figure 374, SIN function example

SINH

Returns the hyperbolic sin for the argument, where the argument is an angle expressed in radians. The output format is double.

SMALLINT

Converts either a number or a valid character value into a smallint value.

```
SELECT D1                              ANSWER
      ,SMALLINT(D1)                    ==================================
      ,SMALLINT('+123')                D1    2      3      4      5
      ,SMALLINT('-123')                ----- ------ ------ ------ ------
      ,SMALLINT(' 123 ')                -2.4     -2    123   -123    123
FROM   SCALAR;                           0.0      0    123   -123    123
                                         1.8      1    123   -123    123
```

Figure 375, SMALLINT function examples

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.

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.

```
SELECT   A.NAME           AS N1          ANSWER
        ,SOUNDEX(A.NAME)  AS S1          ==============================
        ,B.NAME           AS N2          N1      S1   N2        S2   DF
        ,SOUNDEX(B.NAME)  AS S2          ------- ---- --------- ---- --
        ,DIFFERENCE                      Sanders S536 Sneider   S536  4
         (A.NAME,B.NAME)  AS DF          Sanders S536 Smith     S530  3
FROM     STAFF A                         Sanders S536 Lundquist L532  2
        ,STAFF B                         Sanders S536 Daniels   D542  1
WHERE    A.ID = 10                       Sanders S536 Molinare  M456  1
  AND    B.ID > 150                      Sanders S536 Scoutten  S350  1
  AND    B.ID < 250                      Sanders S536 Abrahams  A165  0
ORDER BY DF DESC                         Sanders S536 Kermisch  K652  0
        ,N2 ASC;                         Sanders S536 Lu        L000  0
```

Figure 376, SOUNDEX function example

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:

B, P, F, V	1
C, G, J, K, Q, S, X, Z	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.

NOTE: 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 similar-sounding names.

SPACE

Returns a string consisting of "n" blanks. The output format is varchar(4000).

```
WITH TEMP1(N1) AS                                    ANSWER
(VALUES (1),(2),(3))                                 ==================
SELECT N1                                            N1  S1    S2  S3
      ,SPACE(N1)         AS S1                       --  ----  --  ----
      ,LENGTH(SPACE(N1)) AS S2                        1         1   X
      ,SPACE(N1) || 'X'  AS S3                        2         2    X
FROM   TEMP1;                                         3         3     X
```

Figure 377, SPACE function examples

SQLCACHE_SNAPSHOT

DB2 maintains a dynamic SQL statement cache. It also has several fields that record usage of the SQL statements in the cache. The following command can be used to access this data:

```
DB2 GET SNAPSHOT FOR DYNAMIC SQL ON SAMPLE WRITE TO FILE


          ANSWER - PART OF (ONE OF THE STATEMENTS IN THE SQL CACHE)
          =============================================================
          Number of executions               = 8
          Number of compilations             = 1
          Worst preparation time (ms)        = 3
          Best preparation time (ms)         = 3
          Rows deleted                       = Not Collected
          Rows inserted                      = Not Collected
          Rows read                          = Not Collected
          Rows updated                       = Not Collected
          Rows written                       = Not Collected
          Statement sorts                    = Not Collected
          Total execution time (sec.ms)      = Not Collected
          Total user cpu time (sec.ms)       = Not Collected
          Total system cpu time (sec.ms)     = Not Collected
          Statement text                     = select min(dept) from staff
```

Figure 378, GET SNAPSHOT command

The SQLCACHE_SNAPSHOT table function can also be used to obtain the same data - this time in tabular format. One first has to run the above GET SNAPSHOT command. Then one can run a query like the following:

```
SELECT    *
FROM      TABLE(SQLCACHE_SNAPSHOT()) SS
WHERE     SS.NUM_EXECUTIONS <> 0;
```

Figure 379, SQLCACHE_SNAPSHOT function example

If one runs the RESET MONITOR command, the above execution and compilation counts will be set to zero, but all other fields will be unaffected.

The following query can be used to list all the columns returned by this function:

```
SELECT    ORDINAL                 AS COLNO
         ,CHAR(PARMNAME,18)       AS COLNAME
         ,TYPENAME                AS COLTYPE
         ,LENGTH
         ,SCALE
FROM      SYSCAT.FUNCPARMS
WHERE     FUNCSCHEMA = 'SYSFUN'
  AND     FUNCNAME   = 'SQLCACHE_SNAPSHOT'
ORDER BY COLNO;
```

Figure 380, List columns returned by SQLCACHE_SNAPSHOT

SQRT

Returns the square root of the input value, which can be any positive number. The output format is double.

```
WITH TEMP1(N1) AS                                          ANSWER
(VALUES (0.5),(0.0)                                        ============
       ,(1.0),(2.0))                                       N1     S1
SELECT DEC(N1,4,3)       AS N1                             -----  -----
      ,DEC(SQRT(N1),4,3) AS S1                             0.500  0.707
FROM   TEMP1;                                              0.000  0.000
                                                           1.000  1.000
                                                           2.000  1.414
```

Figure 381, SQRT function example

SUBSTR

Returns part of a string. If the length is not provided, the output is from the start value to the end of the string.

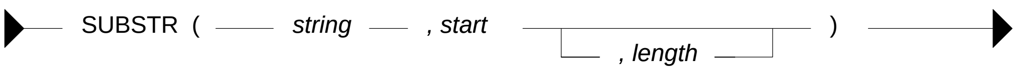

Figure 382, SUBSTR function syntax

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).

```
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
<error>
```

Figure 383, SUBSTR function - error because length parm too long

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:

```
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
```

Figure 384, SUBSTR function - avoid error using CASE (see previous)

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):

```
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
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
```

Figure 385, SUBSTR function - fixed length output if third parm. used

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:

```
SELECT   A.ID                                      ANSWER
        ,A.DEPT                                    =========================
        ,A.SALARY                                  ID DEPT SALARY   DEPTSAL
        ,B.DEPTSAL                                 -- ---- -------- --------
FROM     STAFF  A                                  10 20   18357.50 64286.10
        ,TABLE                                     20 20   18171.25 64286.10
        (SELECT   B.DEPT                           30 38   17506.75 77285.55
                 ,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;
```

Figure 386, Full-select with external table reference

See page 249 for more details on using of the TABLE phrase in a nested table expression.

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.

```
CREATE ALIAS EMP1 FOR EMPLOYEE;                     ANSWER
CREATE ALIAS EMP2 FOR EMP1;                         =======================
                                                    TABSCHEMA TABNAME  CARD
SELECT TABSCHEMA                                    --------- -------- ----
      ,TABNAME                                      GRAEME    EMPLOYEE   -1
      ,CARD
FROM   SYSCAT.TABLES
WHERE  TABNAME   = TABLE_NAME('EMP2','GRAEME');
```

Figure 387, TABLE_NAME function example

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).

```
CREATE VIEW FRED1 (C1, C2, C3)              ANSWER
AS VALUES (11, 'AAA', 'BBB');               ==========================
                                            TAB_SCH  TAB_NME
CREATE ALIAS FRED2 FOR FRED1;               -------- ------------------
CREATE ALIAS FRED3 FOR FRED2;               GRAEME   FRED1
                                            GRAEME   XXXXX
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;
```

Figure 388, TABLE_SCHEMA and TABLE_NAME functions example

TAN

Returns the tangent of the argument where the argument is an angle expressed in radians.

TANH

Returns the hyperbolic tan for the argument, where the argument is an angle expressed in radians. The output format is double.

TIME

Converts the input into a time value.

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.

```
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;
```

Figure 389, TIMESTAMP function examples

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.

```
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
```

Figure 390, TIMESTAMP_FORMAT function example

Note that the only allowed formatting mask is the one shown.

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.

```
SELECT TM1                           ANSWER
      ,TIMESTAMP_ISO(TM1)            ===================================
FROM   SCALAR;                       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
```

Figure 391, TIMESTAMP_ISO function example

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.

```
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'))
SELECT DF1
      ,TIMESTAMPDIFF(16,DF1) AS DIFF
      ,DAYS(TS1) - DAYS(TS2) AS DAYS
FROM   (SELECT TS1
              ,TS2
              ,CHAR(TS1 - TS2) AS DF1
        FROM   (SELECT TIMESTAMP(TS1) AS TS1
                      ,TIMESTAMP(TS2) AS TS2
                FROM   TEMP1
               )AS TEMP2               ANSWER
       )AS TEMP3;                      ================================
                                       DF1                    DIFF DAYS
                                       ---------------------- ---- ----
                                       00010000000001.000000   365  366
                                       00001130235959.000000   360  366
```

Figure 392, TIMESTAMPDIFF function example

WARNING: The microsecond interval option for TIMESTAMPDIFF has a bug. Do not use. The other interval types return estimates, not definitive differences, so should be used with care. To get the difference between two timestamps in days, use the DAYS function as shown above. It is more accurate.

Roll Your Own

The SQL will get the difference, in microseconds, between two timestamp values. It can be used as an alternative to the above function.

```
WITH TEMP1 (TS1,TS2) AS
(VALUES ('1995-03-01-00.12.34.000','1995-03-01-00.00.00.000')
       ,('1995-03-01-00.12.00.034','1995-03-01-00.00.00.000'))
SELECT MS1
      ,MS2
      ,MS1 - MS2 AS DIFF
FROM  (SELECT BIGINT(DAYS(TS1)             * 86400000000
                   + MIDNIGHT_SECONDS(TS1) *     1000000
                   + MICROSECOND(TS1)) AS MS1
             ,BIGINT(DAYS(TS2)             * 86400000000
                   + MIDNIGHT_SECONDS(TS2) *     1000000
                   + MICROSECOND(TS2)) AS MS2
        FROM (SELECT TIMESTAMP(TS1) AS TS1
                    ,TIMESTAMP(TS2) AS TS2
              FROM   TEMP1
             )AS TEMP2
       )AS TEMP3
ORDER BY 1;              ANSWER
                         =============================================
                         MS1               MS2               DIFF
                         ----------------- ----------------- ---------
                         62929699920034000 62929699200000000 720034000
                         62929699954000000 62929699200000000 754000000
```

Figure 393, Difference in microseconds between two timestamps

TO_CHAR

This function is a synonym for VARCHAR_FORMAT (see page 145). It converts a timestamp value into a string using a template to define the output layout.

TO_DATE

This function is a synonym for TIMESTAMP_FORMAT (see page 140). It converts a character string value into a timestamp using a template to define the input layout.

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.

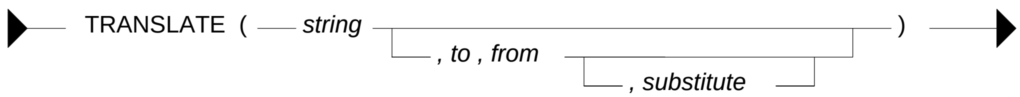

Figure 394, TRANSLATE function syntax

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.

```
                                              ANS. NOTES
                                              ==== =================
SELECT ’abcd’                            ==>  abcd No change
      ,TRANSLATE(’abcd’)                 ==>  ABCD Make upper case
      ,TRANSLATE(’abcd’,’’,’a’)          ==>   bcd ’a’=>’ ’
      ,TRANSLATE(’abcd’,’A’,’A’)              abcd ’A’=>’A’
      ,TRANSLATE(’abcd’,’A’,’a’)              Abcd ’a’=>’A’
      ,TRANSLATE(’abcd’,’A’,’ab’)             A cd ’a’=>’A’,’b’=>’ ’
      ,TRANSLATE(’abcd’,’A’,’ab’,’ ’)         A cd ’a’=>’A’,’b’=>’ ’
      ,TRANSLATE(’abcd’,’A’,’ab’,’z’)         Azcd ’a’=>’A’,’b’=>’z’
      ,TRANSLATE(’abcd’,’AB’,’a’)             Abcd ’a’=>’A’
FROM   STAFF
WHERE  ID = 10;
```

Figure 395, TRANSLATE function examples

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.

```
                                                              ANSWER
                                                              ======
SELECT C1                                                ==>  ABCD
      ,REPLACE(C1,’AB’,’XY’)                             ==>  XYCD
      ,REPLACE(C1,’BA’,’XY’)                             ==>  ABCD
      ,TRANSLATE(C1,’XY’,’AB’)                                XYCD
      ,TRANSLATE(C1,’XY’,’BA’)                                YXCD
FROM   SCALAR
WHERE  C1 = ’ABCD’;
```

Figure 396, REPLACE vs. TRANSLATE

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.

```
                          ANSWER
                          ===============================================
                          D1      POS2    POS1    ZERO    NEG1    NEG2
                          ------- ------- ------- ------- ------- -------
WITH TEMP1(D1) AS         123.400 123.400 123.400 123.000 120.000 100.000
(VALUES (123.400)          23.450  23.440  23.400  23.000  20.000   0.000
       ,( 23.450)           3.456   3.450   3.400   3.000   0.000   0.000
       ,(  3.456)           0.056   0.050   0.000   0.000   0.000   0.000
       ,(   .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;
```

Figure 397, TRUNCATE function examples

TYPE_ID

Returns the internal type identifier of he dynamic data type of the expression.

TYPE_NAME

Returns the unqualified name of the dynamic data type of the expression.

TYPE_SECHEMA

Returns the schema name of the dynamic data type of the expression.

UCASE or UPPER

Coverts a mixed or lower-case string to upper case. The output is the same data type and length as the input.

```
SELECT NAME                                        ANSWER
      ,LCASE(NAME) AS LNAME                        =========================
      ,UCASE(NAME) AS UNAME                        NAME     LNAME    UNAME
FROM   STAFF                                       -------  -------  -------
WHERE  ID < 30;                                    Sanders  sanders  SANDERS
                                                   Pernal   pernal   PERNAL
```

Figure 398, UCASE function example

VALUE

Same as COALESCE.

VARCHAR

Converts the input (1st argument) to a varchar data type. The output length (2nd argument) is optional. Trailing blanks are not removed.

```
SELECT C1                                      ANSWER
      ,LENGTH(C1)          AS L1               ========================
      ,VARCHAR(C1)         AS V2               C1     L1 V2     L2 V3
      ,LENGTH(VARCHAR(C1)) AS L2               ------ -- ------ -- ----
      ,VARCHAR(C1,4)       AS V3               ABCDEF  6 ABCDEF  6 ABCD
FROM   SCALAR;                                 ABCD    6 ABCD    6 ABCD
                                               AB      6 AB      6 AB
```

Figure 399, VARCHAR function examples

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.

```
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
```

Figure 400, VARCHAR_FORMAT function example

Note that the only allowed formatting mask is the one shown.

VARGRAPHIC

Converts the input (1st argument) to a vargraphic data type. The output length (2nd argument) is optional.

VEBLOB_CP_LARGE

This is an undocumented function that IBM has included.

VEBLOB_CP_LARGE

This is an undocumented function that IBM has included.

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.

```
SELECT  WEEK(DATE('2000-01-01')) AS W1             ANSWER
       ,WEEK(DATE('2000-01-02')) AS W2             ==================
       ,WEEK(DATE('2001-01-02')) AS W3             W1  W2  W3  W4  W5
       ,WEEK(DATE('2000-12-31')) AS W4             --  --  --  --  --
       ,WEEK(DATE('2040-12-31')) AS W5              1   2   1  54  53
FROM    SYSIBM.SYSDUMMY1;
```

Figure 401, WEEK function examples

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 page 326).

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.

```
WITH                                            ANSWER
TEMP1 (N) AS                                    ==========================
  (VALUES (0)                                   DTE        DY  WK DY WI DI
   UNION ALL                                    ---------- --- -- -- -- --
   SELECT N+1                                   1998-12-27 Sun 53  1 52  7
   FROM   TEMP1                                 1998-12-28 Mon 53  2 53  1
   WHERE  N < 10),                              1998-12-29 Tue 53  3 53  2
TEMP2 (DT2) AS                                  1998-12-30 Wed 53  4 53  3
  (SELECT DATE('1998-12-27') + Y.N YEARS        1998-12-31 Thu 53  5 53  4
                             + D.N DAYS         1999-01-01 Fri  1  6 53  5
   FROM   TEMP1 Y                               1999-01-02 Sat  1  7 53  6
         ,TEMP1 D                               1999-01-03 Sun  2  1 53  7
   WHERE  Y.N IN (0,2))                         1999-01-04 Mon  2  2  1  1
SELECT   CHAR(DT2,ISO)                DTE       1999-01-05 Tue  2  3  1  2
        ,SUBSTR(DAYNAME(DT2),1,3)     DY        1999-01-06 Wed  2  4  1  3
        ,WEEK(DT2)                    WK        2000-12-27 Wed 53  4 52  3
        ,DAYOFWEEK(DT2)               DY        2000-12-28 Thu 53  5 52  4
        ,WEEK_ISO(DT2)                WI        2000-12-29 Fri 53  6 52  5
        ,DAYOFWEEK_ISO(DT2)           DI        2000-12-30 Sat 53  7 52  6
FROM     TEMP2                                  2000-12-31 Sun 54  1 52  7
ORDER BY 1;                                     2001-01-01 Mon  1  2  1  1
                                                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
```

Figure 402, WEEK_ISO function example

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.

```
SELECT DT1                                      ANSWER
      ,YEAR(DT1) AS YR                          ======================
      ,WEEK(DT1) AS WK                          DT1          YR    WK
FROM   SCALAR;                                  ----------  ----  ----
                                                04/22/1996  1996    17
                                                08/15/1996  1996    33
                                                01/01/0001     1     1
```

Figure 403, YEAR and WEEK functions example

"+" 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:

```
SELECT   ID                                       ANSWER
        ,SALARY                                   ==============================
        ,"+"(SALARY)    AS S2                     ID SALARY   S2       S3
        ,"+"(SALARY,ID) AS S3                     -- -------- -------- --------
FROM     STAFF                                    10 18357.50 18357.50 18367.50
WHERE    ID < 40                                  20 18171.25 18171.25 18191.25
ORDER BY ID;                                      30 17506.75 17506.75 17536.75
```

Figure 404, PLUS function examples

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:

```
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
```

Figure 405, Adding one year to date value

"-" MINUS

The MINUS works the same way as the PLUS function, but does the opposite:

```
SELECT   ID                                     ANSWER
        ,SALARY                                 ================================
        ,"-"(SALARY)    AS S2                   ID SALARY   S2        S3
        ,"-"(SALARY,ID) AS S3                   -- -------- --------- --------
FROM     STAFF                                  10 18357.50 -18357.50 18347.50
WHERE    ID < 40                                20 18171.25 -18171.25 18151.25
ORDER BY ID;                                    30 17506.75 -17506.75 17476.75
```

Figure 406, MINUS function examples

"*" MULTIPLY

The MULTIPLY function is used to multiply two numeric values:

```
SELECT   ID                                   ANSWER
        ,SALARY                               ===================================
        ,SALARY * ID    AS S2                 ID SALARY   S2        S3
        ,"*"(SALARY,ID) AS S3                 -- -------- --------- ---------
FROM     STAFF                                10 18357.50 183575.00 183575.00
WHERE    ID < 40                              20 18171.25 363425.00 363425.00
ORDER BY ID;                                  30 17506.75 525202.50 525202.50
```

Figure 407, MULTIPLY function examples

"/" DIVIDE

The DIVIDE function is used to divide two numeric values:

```
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 18357.50 1835.750 1835.750
20 18171.25  908.562  908.562
30 17506.75  583.558  583.558
```

Figure 408, DIVIDE function examples

"||" CONCAT

Same as the CONCAT function:

```
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
```

Figure 409, CONCAT function examples

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.

WARNING: 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.

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 the basic syntax:

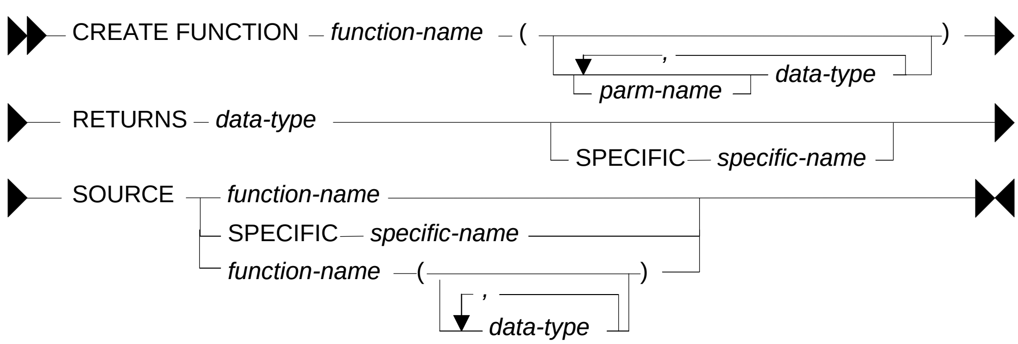

Figure 410, Sourced function syntax

Below is a scalar function that is a variation on the standard DIGITS function, but which only works on small integer fields:

```
CREATE  FUNCTION digi_int (SMALLINT)
RETURNS CHAR(5)
SOURCE  SYSIBM.DIGITS(SMALLINT);
```

Figure 411, Create sourced function

Here is an example of the function in use:

```
SELECT   id              AS ID                              ANSWER
        ,DIGITS(id)      AS I2                              ==============
        ,digi_int(id)    AS I3                              ID I2    I3
FROM     staff                                              -- ----- -----
WHERE    id < 40                                            10 00010 00010
ORDER BY id;                                                20 00020 00020
                                                            30 00030 00030
```

Figure 412, Using sourced function - works

By contrast, the following statement will fail because the input is an integer field:

```
SELECT  id                                                         ANSWER
       ,digi_int(INT(id))                                          =======
FROM    staff                                                      <error>
WHERE   id < 50;
```

Figure 413, Using sourced function - fails

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 us_dollars AS DEC(7,2) WITH COMPARISONS;

CREATE TABLE customers
(ID         SMALLINT     NOT NULL
,balance    us_dollars   NOT NULL);
                                                               ANSWER
INSERT INTO customers VALUES (1 ,111.11),(2 ,222.22);          ==========
                                                               ID balance
SELECT   *                                                     -- -------
FROM     customers                                              1  111.11
ORDER BY ID;                                                    2  222.22
```

Figure 414, Create distinct type and test table

The next query will fail because there is currently no multiply function for "us_dollars":

```
SELECT   ID                                                        ANSWER
        ,balance * 10                                              =======
FROM     customers                                                 <error>
ORDER BY ID;
```

Figure 415, Do multiply - fails

The enable the above, we have to create a sourced function:

```
CREATE FUNCTION "*" (us_dollars,INT)
RETURNS us_dollars
SOURCE SYSIBM."*"(DECIMAL,INT);
```

Figure 416, Create sourced function

Now we can do the multiply:

```
SELECT   ID                                                    ANSWER
        ,balance * 10 AS NEWBAL                                ==========
FROM     customers                                             ID NEWBAL
ORDER BY ID;                                                   -- -------
                                                                1 1111.10
                                                                2 2222.20
```

Figure 417, Do multiply - works

For the record, here is another way to write the same:

```
SELECT   ID                                           ANSWER
        ,"*"(balance,10) AS NEWBAL                    ==========
FROM     customers                                    ID NEWBAL
ORDER BY ID;                                          -- -------
                                                       1 1111.10
                                                       2 2222.20
```

Figure 418, Do multiply - works

Scalar Functions

A scalar function has as input a specific number of values (i.e. not a table) and returns a single output item. Here is the syntax (also for table function):

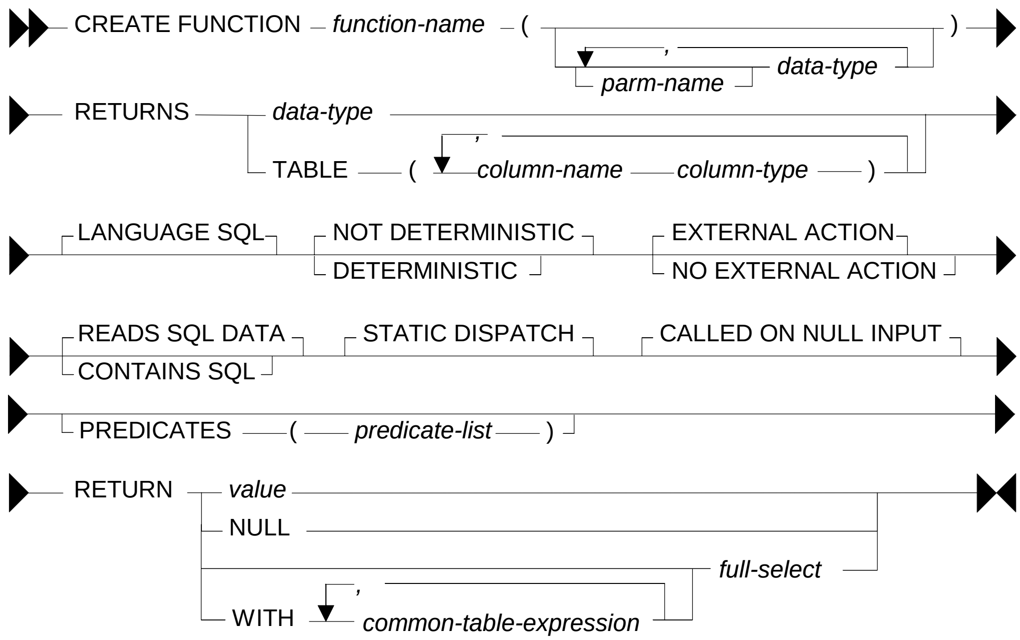

Figure 419, Scalar and Table function syntax

Description

- FUNCTION NAME: A qualified or unqualified name, that along with the number and type of parameters, uniquely identifies the function.
- RETURNS: The type of value returned, if a scalar function. For a table function, the list of columns, with their type.
- LANGUAGE SQL: This the default, and the only one that is supported.
- DETERMINISTIC: Specifies whether the function always returns the same result for a given input. For example, a function that multiplies the input number by ten is deterministic, whereas a function that gets the current timestamp is not. The optimizer needs to know this information.
- EXTERNAL ACTION: Whether the function takes some action, or changes some object that is not under the control of DB2. The optimizer needs to know this information.

- READS SQL DATA: Whether the function reads SQL data only, or doesn't even do that. The function cannot modify any DB2 data, except via an external procedure call.
- STATIC DISPATCH: At function resolution time, DB2 chooses the function to run based on the parameters of the function.
- CALLED ON NULL INPUT: The function is called, even when the input is null.
- PREDICATES: For predicates using this function, this clause lists those that can use the index extensions. If this clause is specified, function must also be DETERMINISTIC with NO EXTERNAL ACTION. See the DB2 documentation for details.
- RETURN: The value or table (result set) returned by the function.

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 page 301 for a detailed description of this problem.

Examples

In addition to the examples shown in this section, there are also the following:

- Check character input is a numeric value - page 298
- Covert numeric data to character (right justified) - page 300.
- Locate string in input, a block at a time - page 268.
- Sort character field contents - page 313.
- Strip characters from text - page 311.

Below is a very simple scalar function - that always returns zero:

```
CREATE FUNCTION returns_zero() RETURNS SMALLINT RETURN 0;
                                                                ANSWER
SELECT   id              AS ID                                  ======
        ,returns_zero()  AS ZZ                                  ID ZZ
FROM     staff                                                  -- --
WHERE    id = 10;                                               10  0
```

Figure 420, Simple function usage

Two functions can be created with the same name. Which one is used depends on the input type that is provided:

```
CREATE FUNCTION calc(inval SMALLINT) RETURNS INT RETURN inval * 10;
CREATE FUNCTION calc(inval INTEGER)  RETURNS INT RETURN inval *  5;

SELECT   id                 AS ID                          ANSWER
        ,calc(SMALLINT(id)) AS C1                          ==========
        ,calc(INTEGER (id)) AS C2                          ID C1  C2
FROM     staff                                             -- --- ---
WHERE    id < 30                                           10 100  50
ORDER BY id;                                               20 200 100

DROP FUNCTION calc(SMALLINT);
DROP FUNCTION calc(INTEGER);
```

Figure 421, Two functions with same name

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:

```
CREATE FUNCTION rnd(inval INT)
RETURNS SMALLINT
NOT DETERMINISTIC
RETURN RAND() * 50;                                            ANSWER
                                                               ======
SELECT   id     AS ID                                          ID RND
        ,rnd(1) AS RND                                         -- ---
FROM     staff                                                 10  37
WHERE    id < 40                                               20   8
ORDER BY id;                                                   30  42
```

Figure 422, Not deterministic function

The next function uses a query to return a single row/column value:

```
CREATE FUNCTION get_sal(inval SMALLINT)
RETURNS DECIMAL(7,2)
RETURN SELECT salary
       FROM   staff
       WHERE  ID = inval;                                  ANSWER
                                                           ===========
SELECT   id          AS ID                                 ID SALARY
        ,get_sal(id) AS SALARY                             -- --------
FROM     staff                                             10 18357.50
WHERE    id < 40                                           20 18171.25
ORDER BY id;                                               30 17506.75
```

Figure 423, Function using query

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:

```
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;
                                                   ANSWER
SELECT   id          AS ID                         ====================
        ,salary      AS SAL1                       ID SAL1     SAL2
        ,max_sal(id) AS SAL2                       -- -------- --------
FROM     staff                                     10 18357.50 22959.20
WHERE    id < 40                                   20 18171.25 18357.50
ORDER BY id;                                       30 17506.75 19260.25
```

Figure 424, Function using common table expression

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:

```
CREATE FUNCTION remove_e(instr VARCHAR(50))
RETURNS VARCHAR(50)
RETURN replace(instr,'e','');

UPDATE   staff
SET      name = remove_e(name)
WHERE    id < 40;
```

Figure 425, Function used in update

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:

```
--#SET DELIMITER !                                          IMPORTANT
                                                            ============
CREATE FUNCTION reverse(instr VARCHAR(50))                  This example
RETURNS VARCHAR(50)                                         uses an "!"
BEGIN ATOMIC                                                as the stmt
   DECLARE outstr  VARCHAR(50) DEFAULT '';                  delimiter.
   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!
                                                    ANSWER
SELECT   id             AS ID                       ====================
        ,name           AS NAME1                    ID NAME1    NAME2
        ,reverse(name)  AS NAME2                    -- -------- -------
FROM     staff                                      10 Sanders  srednaS
WHERE    id < 40                                    20 Pernal   lanreP
ORDER BY id!                                        30 Marenghi ihgneraM
```

Figure 426, Function using compound SQL

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:

```
--#SET DELIMITER !                                          IMPORTANT
                                                            ============
CREATE FUNCTION check_len(instr VARCHAR(50))                This example
RETURNS SMALLINT                                            uses an "!"
BEGIN ATOMIC                                                as the stmt
   IF instr IS NULL THEN                                    delimiter.
      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);                               ANSWER
END!                                                   =================
                                                       ID NAME1    NAME2
SELECT   id              AS ID                         -- -------- -----
        ,name            AS NAME1                      10 Sanders      7
        ,check_len(name) AS NAME2                      20 Pernal      -1
FROM     staff                                         30 Marenghi     8
WHERE    id < 60                                       40 O'Brien      7
ORDER BY id!                                           <error>
```

Figure 427, Function with error checking logic

The above query failed when it got to the name "Hanes", which is less than six bytes long.

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:

```
CREATE FUNCTION get_staff()
RETURNS TABLE (ID      SMALLINT
              ,NAME    VARCHAR(9)
              ,YR      SMALLINT)
RETURN SELECT  id
              ,name
              ,years                                         ANSWER
       FROM    staff;                                        ==============
                                                             ID NAME     YR
SELECT   *                                                   -- -------- --
FROM     TABLE(get_staff()) AS s                             10 Sanders   7
WHERE    id < 40                                             20 Pernal    8
ORDER BY id;                                                 30 Marenghi  5
```

Figure 428, Simple table function

NOTE: See page 151 for the create table function syntax diagram.

Description

The basic syntax for selecting from a table function goes as follows:

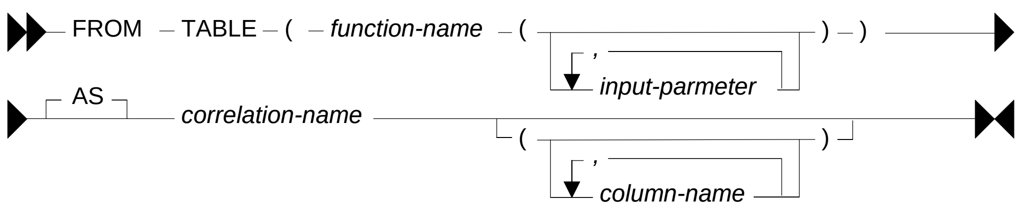

Figure 429, Table function usage - syntax

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:

```
CREATE FUNCTION get_st(inval INTEGER)
RETURNS TABLE (ID      SMALLINT
              ,NAME    VARCHAR(9)
              ,YR      SMALLINT)
RETURN SELECT  id
              ,name
              ,years
       FROM    staff                                         ANSWER
       WHERE   id = inval;                                   ==============
                                                             ID NNN      YY
SELECT   *                                                   -- -------- --
FROM     TABLE(get_st(30)) AS sss (ID, NNN, YY);             30 Marenghi  5
```

Figure 430, Table function with parameters

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:

```
CREATE FUNCTION make_data()
RETURNS TABLE (KY    SMALLINT
              ,DAT   CHAR(5))
RETURN  WITH temp1 (k#) AS (VALUES (1),(2),(3))              ANSWER
        SELECT k#                                            ========
              ,DIGITS(SMALLINT(k#))                          KY DAT
        FROM   temp1;                                        -- -----
                                                              1 00001
SELECT   *                                                    2 00002
FROM     TABLE(make_data()) AS ttt;                           3 00003
```

Figure 431, Table function that creates data

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:

```
CREATE FUNCTION staff_list(lo_key INTEGER                IMPORTANT
                          ,lo_sal INTEGER)               ============
RETURNS TABLE (id      SMALLINT                          This example
              ,salary  DECIMAL(7,2)                      uses an "!"
              ,max_sal DECIMAL(7,2)                      as the stmt
              ,id_max  SMALLINT)                         delimiter.
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                     ANSWER
      FROM   staff                        ============================
      WHERE  id >= lo_key;                ID  SALARY   MAX_SAL  ID_MAX
END!                                      --- -------- -------- ------
                                           70 16502.83 22959.20    160
SELECT   *                                 80 13504.60 22959.20    160
FROM     TABLE(staff_list(66,1)) AS ttt    90 18001.75 22959.20    160
WHERE    id < 111                         100 18352.80 22959.20    160
ORDER BY id!                              110 12508.20 22959.20    160
```

Figure 432, Table function with compound SQL

Order By, Group By, and Having

Introduction

The GROUP BY statement is used to combine multiple rows into one. The HAVING expression is where one can select which of the combined rows are to be retrieved. In this sense, the HAVING and the WHERE expressions are very similar. The ORDER BY statement is used to sequence the rows in the final output.

Order By

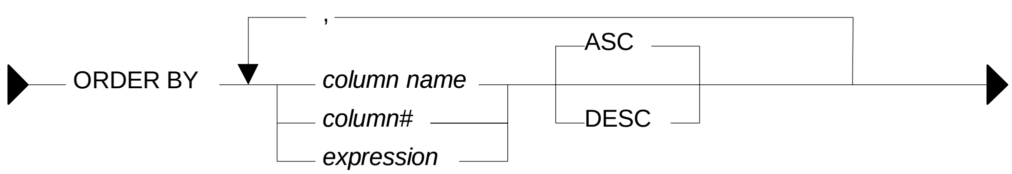

Figure 433, ORDER BY syntax

The ORDER BY statement can only be applied to the final result set of the SQL statement. Unlike the GROUP BY, it can not be used on any intermediate result set (e.g. a sub-query or a nested-table expression). Nor can it be used in a view definition.

Sample Data

```
CREATE VIEW seq_data(col1,col2) AS VALUES
('ab','xy'),('AB','xy'),('ac','XY'),('AB','XY'),('Ab','12');
```

Figure 434, ORDER BY sample data definition

Order by Examples

```
SELECT   col1                                                   ANSWER
        ,col2                                                   =========
FROM     seq_data                                               COL1 COL2
ORDER BY col1 ASC                                               ---- ----
        ,col2;                                                  ab   xy
                                                                ac   XY
                                                                Ab   12
                                                                AB   xy
                                                                AB   XY
```

Figure 435, Simple ORDER BY

Observe how in the above example all of the lower case data comes before the upper case data. Use the TRANSLATE function to display the data in case-independent order:

```
SELECT   col1                                                   ANSWER
        ,col2                                                   =========
FROM     seq_data                                               COL1 COL2
ORDER BY TRANSLATE(col1) ASC                                    ---- ----
        ,TRANSLATE(col2) ASC                                    Ab   12
                                                                ab   xy
                                                                AB   XY
                                                                AB   xy
                                                                ac   XY
```

Figure 436, Case insensitive ORDER BY

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:

```
SELECT   col2                                                 ANSWER
FROM     seq_data                                             ======
ORDER BY col1                                                 COL2
        ,col2;                                                ----
                                                              xy
                                                              XY
                                                              12
                                                              xy
                                                              XY
```

Figure 437, ORDER BY on not-displayed column

In the next example, the data is (primarily) sorted in descending sequence, based on the second byte of the first column:

```
SELECT   col1                                              ANSWER
        ,col2                                              =========
FROM     seq_data                                          COL1 COL2
ORDER BY SUBSTR(col1,2) DESC                               ---- ----
        ,col2                                              ac   XY
        ,1;                                                AB   xy
                                                           AB   XY
                                                           Ab   12
                                                           ab   xy
```

Figure 438, ORDER BY second byte of first column

If a character column is defined FOR BIT DATA, the data is returned in internal ASCII sequence, as opposed to the standard collating sequence where 'a' < 'A' < 'b' < 'B'. In ASCII sequence all upper case characters come before all lower case characters. In the following example, the HEX function is used to display ordinary character data in bit-data order:

```
SELECT   col1                                    ANSWER
        ,HEX(col1) AS hex1                       ===================
        ,col2                                    COL1 HEX1 COL2 HEX2
        ,HEX(col2) AS hex2                       ---- ---- ---- ----
FROM     seq_data                                AB   4142 XY   5859
ORDER BY HEX(col1)                               AB   4142 xy   7879
        ,HEX(col2)                               Ab   4162 12   3132
                                                 ab   6162 xy   7879
                                                 ac   6163 XY   5859
```

Figure 439, ORDER BY in bit-data sequence

Arguably, either the BLOB or CLOB functions should be used (instead of HEX) to get the data in ASCII sequence. However, when these two were tested (in DB2BATCH) they caused the ORDER BY to fail.

Notes

- 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 data output order.
- If the ORDER BY column list does not uniquely identify each row, those 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 always sort high.

Group By and Having

The GROUP BY statement is used to group individual rows into combined sets based on the value in one, or more, columns. The GROUPING SETS clause is used to define multiple independent GROUP BY clauses in one query. The ROLLUP and CUBE clauses are shorthand forms of the GROUPING SETS statement.

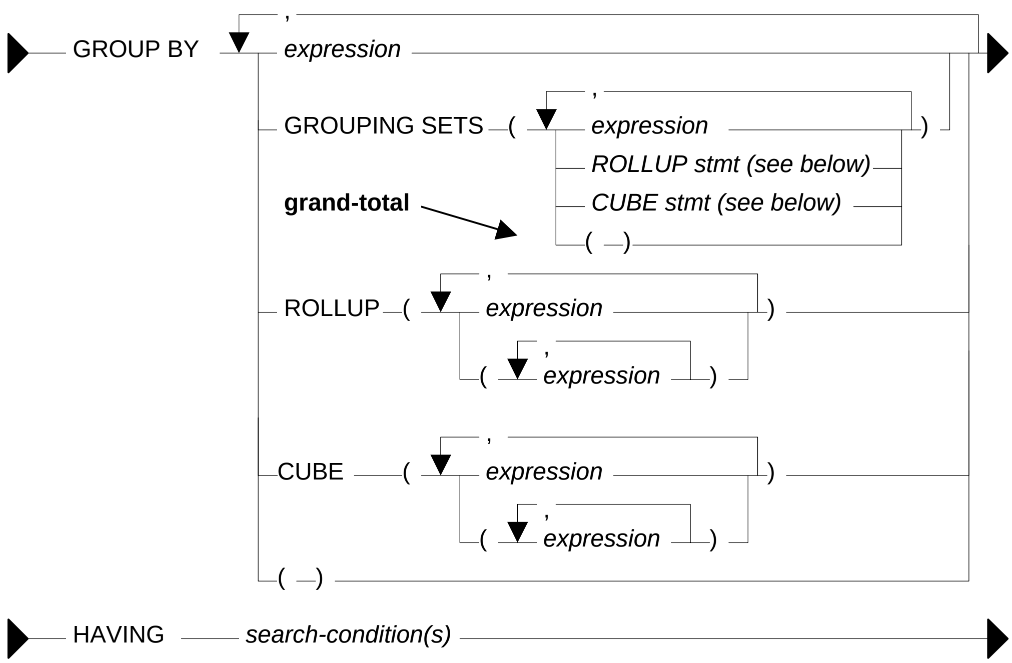

Figure 440, GROUP BY syntax

GROUP BY Sample Data

```
CREATE VIEW employee_view AS
SELECT   SUBSTR(workdept,1,1) AS d1
        ,workdept             AS dept
        ,sex                  AS sex
        ,INTEGER(salary)      AS salary
FROM     employee
WHERE    workdept < 'D20';
COMMIT;


SELECT   *
FROM     employee_view
ORDER BY 1,2,3,4;
```

ANSWER

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

Figure 441, GROUP BY Sample Data

Simple GROUP BY Statements

A simple GROUP BY is used to combine individual rows into a distinct set of summary rows.

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 (i.e. with no GROUPING SETS, ROLLUP or CUBE clause) is always a distinct set of rows, where the unique identifier is whatever fields were grouped on.
- There is no guarantee that the rows resulting from a GROUP BY will come back in any particular order, unless an ORDER BY is also specified.
- 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.

Sample Queries

In this first query we group our sample data by the first three fields in the view:

```
SELECT   d1, dept, sex                          ANSWER
        ,SUM(salary)        AS salary           ========================
        ,SMALLINT(COUNT(*)) AS #rows            D1 DEPT SEX SALARY #ROWS
FROM     employee_view                          -- ---- --- ------ -----
WHERE    dept <> 'ABC'                          A  A00  F    52750     1
GROUP BY d1, dept, sex                          A  A00  M    75750     2
HAVING   dept         > 'A0'                    B  B01  M    41250     1
   AND  (SUM(salary)  > 100                     C  C01  F    90470     3
    OR   MIN(salary)  >  10                     D  D11  F    73430     3
    OR   COUNT(*)    <>  22)                    D  D11  M   148670     6
ORDER BY d1, dept, sex;
```

Figure 442, Simple GROUP BY

There is no need to have the a field in the GROUP BY in the SELECT list, but the answer really doesn't make much sense if one does this:

```
SELECT   sex                                            ANSWER
        ,SUM(salary)        AS salary                   ================
        ,SMALLINT(COUNT(*)) AS #rows                    SEX SALARY #ROWS
FROM     employee_view                                  --- ------ -----
WHERE    sex IN ('F','M')                               F    52750     1
GROUP BY dept                                           F    90470     3
        ,sex                                            F    73430     3
ORDER BY sex;                                           M    75750     2
                                                        M    41250     1
                                                        M   148670     6
```

Figure 443, GROUP BY on non-displayed field

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:

```
SELECT   SUM(salary)        AS salary                       ANSWER
        ,SMALLINT(COUNT(*)) AS #rows                        ============
FROM     employee_view                                      SALARY #ROWS
WHERE    d1 <> 'X'                                          ------ -----
GROUP BY SUBSTR(dept,3,1)                                   128500     3
HAVING   COUNT(*) <> 99;                                    353820    13
```

Figure 444, GROUP BY on derived field, not shown

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:

```
SELECT   SUBSTR(dept,3,1)   AS wpart                 ANSWER
        ,SUM(salary)        AS salary                ==================
        ,SMALLINT(COUNT(*)) AS #rows                 WPART SALARY #ROWS
FROM     employee_view                               ----- ------ -----
GROUP BY SUBSTR(dept,3,1)                            1     353820    13
ORDER BY wpart DESC;                                 0     128500     3
```

Figure 445, GROUP BY on derived field, shown

GROUPING SETS Statement

The GROUPING SETS statement enable one to get multiple GROUP BY result sets from 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.

```
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
                                                                 ,BY C
```

Figure 446, GROUPING SETS in parenthesis vs. not

Multiple GROUPING SETS in the same GROUP BY are combined together as if they were simple fields in a GROUP BY list:

```
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
```

Figure 447, Multiple GROUPING SETS

One can mix simple expressions and GROUPING SETS in the same GROUP BY:

```
GROUP BY A                            is equivalent to      GROUP BY A
        ,GROUPING SETS ((B,C))                                      ,B
                                                                    ,C
```

Figure 448, Simple GROUP BY expression and GROUPING SETS combined

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:

```
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
```

Figure 449, Mixing simple GROUP BY expressions and GROUPING SETS

A single GROUPING SETS statement can contain multiple sets of implied GROUP BY phrases (obviously). These are combined using implied UNION ALL statements:

```
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
```

Figure 450, GROUPING SETS with multiple components

The null-field list "( )" can be used to get a grand total. This is equivalent to not having the GROUP BY at all.

```
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
```

Figure 451, GROUPING SET with multiple components, using grand-total

The above GROUPING SETS statement is equivalent to a ROLLUP(A,B,C), while the next is equivalent to a CUBE(A,B,C):

```
GROUP BY GROUPING SETS ((A,B,C)        is equivalent to       GROUP BY A
                       ,(A,B)                                         ,B
                       ,(A,C)                                         ,C
                       ,(B,C)                                 UNION ALL
                       ,(A)                                   GROUP BY A
                       ,(B)                                           ,B
                       ,(C)                                   UNION ALL
                       ,())                                   GROUP BY A
                                                                      ,C
                                                              UNION ALL
                                                              GROUP BY B
is equivalent to                                                      ,C
                                                              UNION ALL
                                                              GROUP BY A
                                                              UNION ALL
CUBE(A,B,C)                                                   GROUP BY B
                                                              UNION ALL
                                                              GROUP BY C
                                                              UNION ALL
                                                              grand-totl
```

Figure 452, GROUPING SET with multiple components, using grand-total

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:

```
SELECT   d1                                ANSWER
        ,dept                              ==============================
        ,sex                               D1 DEPT SEX    SAL  #R DF WF SF
        ,SUM(salary)        AS sal         -- ---- --- ------ -- -- -- --
        ,SMALLINT(COUNT(*)) AS #r          A  A00  F    52750  1  0  0  0
        ,GROUPING(d1)       AS f1          A  A00  M    75750  2  0  0  0
        ,GROUPING(dept)     AS fd          B  B01  M    41250  1  0  0  0
        ,GROUPING(sex)      AS fs          C  C01  F    90470  3  0  0  0
FROM     employee_view                     D  D11  F    73430  3  0  0  0
GROUP BY GROUPING SETS (d1)                D  D11  M   148670  6  0  0  0
        ,GROUPING SETS ((dept,sex))
ORDER BY d1
        ,dept
        ,sex;
```

Figure 453, Multiple GROUPING SETS, making one GROUP BY

NOTE: 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:

```
SELECT   d1                                ANSWER
        ,dept                              ==============================
        ,sex                               D1 DEPT SEX    SAL  #R F1 FD FS
        ,SUM(salary)        AS sal         -- ---- --- ------ -- -- -- --
        ,SMALLINT(COUNT(*)) AS #r          A   A00 -   128500  3  0  0  1
        ,GROUPING(d1)       AS f1          A   -   F    52750  1  0  1  0
        ,GROUPING(dept)     AS fd          A   -   M    75750  2  0  1  0
        ,GROUPING(sex)      AS fs          B   B01 -    41250  1  0  0  1
FROM     employee_view                     B   -   M    41250  1  0  1  0
GROUP BY GROUPING SETS (d1)                C   C01 -    90470  3  0  0  1
        ,GROUPING SETS (dept,sex)          C   -   F    90470  3  0  1  0
ORDER BY d1                                D   D11 -   222100  9  0  0  1
        ,dept                              D   -   F    73430  3  0  1  0
        ,sex;                              D   -   M   148670  6  0  1  0
```

Figure 454, Multiple GROUPING SETS, making two GROUP BY results

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.

```
SELECT   d1                                ANSWER
        ,dept                              ==============================
        ,sex                               D1 DEPT SEX SAL    #R F1 FD FS
        ,SUM(salary)        AS sal         -- ---- --- ------ -- -- -- --
        ,SMALLINT(COUNT(*)) AS #r          A  A00  F    52750  1  0  0  0
        ,GROUPING(d1)       AS f1          A  A00  M    75750  2  0  0  0
        ,GROUPING(dept)     AS fd          B  B01  M    41250  1  0  0  0
        ,GROUPING(sex)      AS fs          C  C01  F    90470  3  0  0  0
FROM     employee_view                     D  D11  F    73430  3  0  0  0
GROUP BY d1                                D  D11  M   148670  6  0  0  0
        ,dept
        ,GROUPING SETS ((dept,sex))
ORDER BY d1
        ,dept
        ,sex;
```

Figure 455, Repeated field essentially ignored

```
SELECT   d1                                ANSWER
        ,dept                              ==============================
        ,sex                               D1 DEPT SEX SAL    #R F1 FD FS
        ,SUM(salary)        AS sal         -- ---- --- ------ -- -- -- --
        ,SMALLINT(COUNT(*)) AS #r          A  A00  F    52750  1  0  0  0
        ,GROUPING(d1)       AS f1          A  A00  M    75750  2  0  0  0
        ,GROUPING(dept)     AS fd          A  A00  -   128500  3  0  0  1
        ,GROUPING(sex)      AS fs          B  B01  M    41250  1  0  0  0
FROM     employee_view                     B  B01  -    41250  1  0  0  1
GROUP BY d1                                C  C01  F    90470  3  0  0  0
        ,DEPT                              C  C01  -    90470  3  0  0  1
        ,GROUPING SETS (dept,sex)          D  D11  F    73430  3  0  0  0
ORDER BY d1                                D  D11  M   148670  6  0  0  0
        ,dept                              D  D11  -   222100  9  0  0  1
        ,sex;
```

Figure 456, Repeated field impacts query result

The above two queries can be rewritten as follows:

```
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
```

Figure 457, Repeated field impacts query result

NOTE: 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.

ROLLUP Statement

A ROLLUP 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.

```
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)
                                                               ,())
```

Figure 458, ROLLUP vs. GROUPING SETS

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:

```
GROUP BY ROLLUP(A,(B,C))      ===>       GROUP BY GROUPING SETS((A,B,C)
                                                               ,(A)
                                                               ,())
```

Figure 459, ROLLUP vs. GROUPING SETS

Multiple ROLLUP statements in the same GROUP BY act independently of each other:

```
GROUP BY ROLLUP(A)            ===>       GROUP BY GROUPING SETS((A,B,C)
        ,ROLLUP(B,C)                                           ,(A,B)
                                                               ,(A)
                                                               ,(B,C)
                                                               ,(B)
                                                               ,())
```

Figure 460, ROLLUP vs. GROUPING SETS

SQL Examples

Here is a standard GROUP BY that gets no sub-totals:

```
SELECT   dept                                     ANSWER
        ,SUM(salary)          AS salary           ====================
        ,SMALLINT(COUNT(*))   AS #rows            DEPT SALARY #ROWS FD
        ,GROUPING(dept)       AS fd               ---- ------ ----- --
FROM     employee_view                            A00  128500     3 0
GROUP BY dept                                     B01   41250     1 0
ORDER BY dept;                                    C01   90470     3 0
                                                  D11  222100     9 0
```

Figure 461, Simple GROUP BY

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:

```
SELECT   dept                                     ANSWER
        ,SUM(salary)          AS salary           ====================
        ,SMALLINT(COUNT(*))   AS #rows            DEPT SALARY #ROWS FD
        ,GROUPING(dept)       AS FD               ---- ------ ----- --
FROM     employee_view                            A00  128500     3  0
GROUP BY ROLLUP(dept)                             B01   41250     1  0
ORDER BY dept;                                    C01   90470     3  0
                                                  D11  222100     9  0
                                                  -    482320    16  1
```

Figure 462, GROUP BY with ROLLUP

NOTE: 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:

```
SELECT   dept                                         ANSWER
        ,SUM(salary)             AS salary           ====================
        ,SMALLINT(COUNT(*))      AS #rows            DEPT SALARY #ROWS FD
        ,GROUPING(dept)          AS fd               ---- ------ ----- --
FROM     employee_view                               A00  128500     3  0
GROUP BY dept                                        B01   41250     1  0
UNION ALL                                            C01   90470     3  0
SELECT   CAST(NULL AS CHAR(3)) AS dept              D11  222100     9  0
        ,SUM(salary)             AS salary           -    482320    16  1
        ,SMALLINT(COUNT(*))      AS #rows
        ,CAST(1 AS INTEGER)      AS fd
FROM     employee_view
ORDER BY dept;
```

Figure 463, ROLLUP done the old-fashioned way

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:

```
SELECT   dept                                         ANSWER
        ,SUM(salary)        AS salary                ====================
        ,SMALLINT(COUNT(*)) AS #rows                 DEPT SALARY #ROWS FD
        ,GROUPING(dept)     AS fd                    ---- ------ ----- --
FROM     employee_view                               A00  128500     3  0
GROUP BY dept                                        A00  128500     3  0
        ,ROLLUP(dept)                                B01   41250     1  0
ORDER BY dept;                                       B01   41250     1  0
                                                     C01   90470     3  0
                                                     C01   90470     3  0
                                                     D11  222100     9  0
                                                     D11  222100     9  0
```

Figure 464, Repeating a field in GROUP BY and ROLLUP (error)

Below is a graphic representation of why the data rows were repeated above. Observe that two GROUP BY statements were, in effect, generated:

```
GROUP BY dept        => GROUP BY dept                => GROUP BY dept
        ,ROLLUP(dept)           ,GROUPING SETS((dept)    UNION ALL
                                              ,())       GROUP BY dept
                                                                 ,()
```

Figure 465, Repeating a field, explanation

In the next example the GROUP BY, is on two fields, with the second also being rolled up:

```
SELECT   dept                                  ANSWER
        ,sex                                   ===========================
        ,SUM(salary)        AS salary          DEPT SEX SALARY #ROWS FD FS
        ,SMALLINT(COUNT(*)) AS #rows           ---- --- ------ ----- -- --
        ,GROUPING(dept)     AS fd              A00  F    52750     1  0  0
        ,GROUPING(sex)      AS fs              A00  M    75750     2  0  0
FROM     employee_view                         A00  -   128500     3  0  1
GROUP BY dept                                  B01  M    41250     1  0  0
        ,ROLLUP(sex)                           B01  -    41250     1  0  1
ORDER BY dept                                  C01  F    90470     3  0  0
        ,sex;                                  C01  -    90470     3  0  1
                                               D11  F    73430     3  0  0
                                               D11  M   148670     6  0  0
                                               D11  -   222100     9  0  1
```

Figure 466, GROUP BY on 1st field, ROLLUP on 2nd

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).

```
SELECT   dept                                   ANSWER
        ,sex                                    ===========================
        ,SUM(salary)        AS salary           DEPT SEX SALARY #ROWS FD FS
        ,SMALLINT(COUNT(*)) AS #rows            ---- --- ------ ----- -- --
        ,GROUPING(dept)     AS fd               A00  F    52750     1  0  0
        ,GROUPING(sex)      AS fs               A00  M    75750     2  0  0
FROM     employee_view                          A00  -   128500     3  0  1
GROUP BY ROLLUP(dept                            B01  M    41250     1  0  0
               ,sex)                            B01  -    41250     1  0  1
ORDER BY dept                                   C01  F    90470     3  0  0
        ,sex;                                   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
```

Figure 467, ROLLUP on DEPT, then SEX

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:

```
SELECT   sex                                    ANSWER
        ,dept                                   ===========================
        ,SUM(salary)        AS salary           SEX DEPT SALARY #ROWS FD FS
        ,SMALLINT(COUNT(*)) AS #rows            --- ---- ------ ----- -- --
        ,GROUPING(dept)     AS fd               F   A00   52750     1  0  0
        ,GROUPING(sex)      AS fs               F   C01   90470     3  0  0
FROM     employee_view                          F   D11   73430     3  0  0
GROUP BY ROLLUP(sex                             F   -    216650     7  1  0
               ,dept)                           M   A00   75750     2  0  0
ORDER BY sex                                    M   B01   41250     1  0  0
        ,dept;                                  M   D11  148670     6  0  0
                                                M   -    265670     9  1  0
                                                -   -    482320    16  1  1
```

Figure 468, ROLLUP on SEX, then DEPT

The next statement is the same as the prior, but it uses the logically equivalent GROUPING SETS syntax:

```
SELECT   sex                                    ANSWER
        ,dept                                   ===========================
        ,SUM(salary)        AS salary           SEX DEPT SALARY #ROWS FD FS
        ,SMALLINT(COUNT(*)) AS #rows            --- ---- ------ ----- -- --
        ,GROUPING(dept)     AS fd               F   A00   52750     1  0  0
        ,GROUPING(sex)      AS fs               F   C01   90470     3  0  0
FROM     employee_view                          F   D11   73430     3  0  0
GROUP BY GROUPING SETS ((sex, dept)             F   -    216650     7  1  0
                       ,(sex)                   M   A00   75750     2  0  0
                       ,())                     M   B01   41250     1  0  0
ORDER BY sex                                    M   D11  148670     6  0  0
        ,dept;                                  M   -    265670     9  1  0
                                                -   -    482320    16  1  1
```

Figure 469, ROLLUP on SEX, then DEPT

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:

```
SELECT   sex                                  ANSWER
        ,dept                                 ==========================
        ,SUM(salary)         AS salary        SEX DEPT SALARY #ROWS FD FS
        ,SMALLINT(COUNT(*))  AS #rows         --- ---- ------ ----- -- --
        ,GROUPING(dept)      AS fd            F   A00   52750     1  0  0
        ,GROUPING(sex)       AS fs            F   C01   90470     3  0  0
FROM     employee_view                        F   D11   73430     3  0  0
GROUP BY ROLLUP(sex)                          F   -    216650     7  1  0
        ,ROLLUP(dept)                         M   A00   75750     2  0  0
ORDER BY sex                                  M   B01   41250     1  0  0
        ,dept;                                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
```

Figure 470, Two independent ROLLUPS

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:

```
SELECT   dept                                 ANSWER
        ,sex                                  ==========================
        ,SUM(salary)         AS salary        DEPT SEX SALARY #ROWS FD FS
        ,SMALLINT(COUNT(*))  AS #rows         ---- --- ------ ----- -- --
        ,GROUPING(dept)      AS fd            A00  F    52750     1  0  0
        ,GROUPING(sex)       AS fs            A00  M    75750     2  0  0
FROM     employee_view                        B01  M    41250     1  0  0
GROUP BY ROLLUP((dept,sex))                   C01  F    90470     3  0  0
ORDER BY dept                                 D11  F    73430     3  0  0
        ,sex;                                 D11  M   148670     6  0  0
                                              -    -   482320    16  1  1
```

Figure 471, Combined-field ROLLUP

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:

```
SELECT   SUM(salary)         AS salary                       ANSWER
        ,SMALLINT(COUNT(*))  AS #rows                        ============
FROM     employee_view                                       SALARY #ROWS
GROUP BY ROLLUP(sex                                          ------ -----
               ,dept)                                        482320    16
HAVING   GROUPING(dept) = 1
   AND   GROUPING(sex)  = 1
ORDER BY salary;
```

Figure 472, Use HAVING to get only grand-total row

Below is a logically equivalent SQL statement:

```
SELECT   SUM(salary)         AS salary                       ANSWER
        ,SMALLINT(COUNT(*))  AS #rows                        ============
FROM     employee_view                                       SALARY #ROWS
GROUP BY GROUPING SETS(());                                  ------ -----
                                                             482320    16
```

Figure 473, Use GROUPING SETS to get grand-total row

Here is another:

```
SELECT   SUM(salary)         AS salary                    ANSWER
        ,SMALLINT(COUNT(*))  AS #rows                     ============
FROM     employee_view                                    SALARY #ROWS
GROUP BY ();                                              ------ -----
                                                          482320    16
```

Figure 474, Use GROUP BY to get grand-total row

And another:

```
SELECT   SUM(salary)         AS salary                    ANSWER
        ,SMALLINT(COUNT(*))  AS #rows                     ============
FROM     employee_view;                                   SALARY #ROWS
                                                          ------ -----
                                                          482320    16
```

Figure 475, Get grand-total row directly

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.

```
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)
                                                              ,())
```

Figure 476, CUBE vs. GROUPING SETS

As with the ROLLLUP statement, any set of fields in nested parenthesis is treated by the CUBE as a single field:

```
GROUP BY CUBE(A,(B,C))        ===>       GROUP BY GROUPING SETS((A,B,C)
                                                              ,(B,C)
                                                              ,(A)
                                                              ,())
```

Figure 477, CUBE vs. GROUPING SETS

Having multiple CUBE statements is allowed, but very, very silly:

```
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),())
```

Figure 478, CUBE vs. GROUPING SETS

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),())

SQL Examples

Below is a standard CUBE statement:

```
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    52750  1  1  0  0
-  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
```

Figure 479, CUBE example

Here is the same query expressed as 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
A  A00  M    75750  2  0  0  0
etc... (same as prior query)
```

Figure 480, CUBE expressed using multiple GROUPING SETS

Here is the same CUBE statement expressed as a ROLLUP, plus the required additional GROUPING SETS:

```
SELECT   d1                                   ANSWER
        ,dept                                 ==============================
        ,sex                                  D1 DEPT SEX   SAL   #R F1 FD FS
        ,INT(SUM(salary))    AS sal           -- ---- --- ------ -- -- -- --
        ,SMALLINT(COUNT(*))  AS #r            A  A00  F    52750  1  0  0  0
        ,GROUPING(d1)        AS f1            A  A00  M    75750  2  0  0  0
        ,GROUPING(dept)      AS fd            etc... (same as prior query)
        ,GROUPING(sex)       AS fs
FROM     employee_view
GROUP BY GROUPING SETS (ROLLUP(d1, dept, sex)
                              ,(dept, sex)
                              ,(sex, dept)
                              ,(d1, sex))
ORDER BY d1
        ,dept
        ,sex;
```

Figure 481, CUBE expressed using ROLLUP and GROUPING SETS

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:

```
SELECT   d1                                   ANSWER
        ,dept                                 ==============================
        ,sex                                  D1 DEPT SEX  SAL    #R F1 FD FS
        ,INT(SUM(salary))    AS sal           ------------------------------
        ,SMALLINT(COUNT(*))  AS #r            A  A00  F    52750  1  0  0  0
        ,GROUPING(d1)        AS f1            A  A00  M    75750  2  0  0  0
        ,GROUPING(dept)      AS fd            B  B01  M    41250  1  0  0  0
        ,GROUPING(sex)       AS fs            C  C01  F    90470  3  0  0  0
FROM     employee_VIEW                        D  D11  F    73430  3  0  0  0
GROUP BY CUBE((d1, dept, sex))                D  D11  M   148670  6  0  0  0
ORDER BY d1                                   -  -    -   482320 16  1  1  1
        ,dept
        ,sex;
```

Figure 482, CUBE on compound fields

The above query is resolved thus:

```
GROUP BY CUBE((A,B,C) => GROUP BY GROUING SETS ((A,B,C) =>  GROUP BY A
                                               ,())                  ,B
                                                                     ,C
                                                            UNION ALL
                                                            GROUP BY()
```

Figure 483, CUBE on compound field, explanation

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.

NOTE: 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:

```
SELECT   d1                  AS d1                       ANSWER
        ,dept                AS dpt                      ==================
        ,sex                 AS sx                       D1 DPT SX    SAL  R
        ,INT(SUM(salary))    AS sal                      -- --- -- ------ -
        ,SMALLINT(COUNT(*))  AS r                        A  A00 F    52750 1
FROM     employee_VIEW                                   A  A00 M    75750 2
GROUP BY d1                                              B  B01 M    41250 1
        ,dept                                            C  C01 F    90470 3
        ,sex                                             D  D11 F    73430 3
ORDER BY 1,2,3;                                          D  D11 M   148670 6
```

Figure 484, Basic GROUP BY example

Now imagine that we want to extend the above query to get the following sub-total rows:

```
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)
```

Figure 485, Sub-totals that we want to get

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:

```
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                                       ANSWER
         GROUP BY CUBE(d1,dept,sex)         ============================
        )AS xxx                             D1 DPT SX  SAL    #R G1 GD GS
WHERE    (g1,gd,gs) = (0,0,0)               -- --- -- ------ -- -- -- --
   OR    (g1,gd,gs) = (0,0,1)               A  A00 F   52750  1  0  0  0
   OR    (g1,gd,gs) = (0,1,0)               A  A00 M   75750  2  0  0  0
   OR    (g1,gd,gs) = (0,1,1)               A  A00 -  128500  3  0  0  1
   OR    (g1,gd,gs) = (1,1,0)               A  -   F   52750  1  0  1  0
   OR    (g1,gd,gs) = (1,1,1)               A  -   M   75750  2  0  1  0
ORDER BY 1,2,3;                             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
```

Figure 486, Get lots of sub-totals, using CUBE

In the above query, the GROUPING function (see page 71) 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:

```
(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
```

Figure 487, Predicates used - explanation

Here is the same query written using two ROLLUP expressions. You can be the judge as to which is the easier to understand:

```
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
```

Figure 488, Get lots of sub-totals, using ROLLUP

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.

```
SELECT   dept, job
        ,COUNT(*)
FROM     staff
GROUP BY dept, job
ORDER BY dept, job;
```

Figure 489, GROUP BY with ORDER BY

NOTE: 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.

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:

```
WITH staff2 (dept, avgsal) AS                         ANSWER
  (SELECT   dept                                      =================
           ,AVG(salary)                               ID  NAME     DEPT
   FROM     staff                                     --- -------- ----
   GROUP BY dept                                      160 Molinare   10
   HAVING   AVG(salary) > 18000                       210 Lu         10
  )                                                   240 Daniels    10
SELECT   a.id                                         260 Jones      10
        ,a.name
        ,a.dept
FROM     staff  a
        ,staff2 b
WHERE    a.dept = b.dept
ORDER BY a.id;
```

Figure 490, GROUP BY on one side of join - using common table expression

In the next example, we shall use a full-select:

```
SELECT   a.id                                         ANSWER
        ,a.name                                       =================
        ,a.dept                                       ID  NAME     DEPT
FROM     staff  a                                     --- -------- ----
        ,(SELECT   dept        AS dept                160 Molinare   10
                  ,AVG(salary) AS avgsal              210 Lu         10
          FROM     staff                              240 Daniels    10
          GROUP BY dept                               260 Jones      10
          HAVING   AVG(salary) > 18000
         )AS b
WHERE    a.dept = b.dept
ORDER BY a.id;
```

Figure 491, GROUP BY on one side of join - using full-select

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:

```
SELECT   COUNT(*)  AS c1                                         ANSWER
FROM     staff                                                   ======
WHERE    id < 1;                                                      0


SELECT   COUNT(*)  AS c1                                         ANSWER
FROM     staff                                                   ======
WHERE    id < 1                                                  no row
GROUP BY id;
```

Figure 492, COUNT and No Rows

See page 320 for a comprehensive discussion of what happens when no rows match.

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 page 213) the matching rows are joined (in a sense) one-above-the-other to make the result table.

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.

Sample Views

```
CREATE VIEW STAFF_V1 AS                  STAFF_V1         STAFF_V2
SELECT ID, NAME                          +-----------+    +---------+
FROM   STAFF                             |ID|NAME    |    |ID|JOB   |
WHERE  ID BETWEEN 10 AND 30;             |--|--------|    |--|------|
                                         |10|Sanders |    |20|Sales |
CREATE VIEW STAFF_V2 AS                  |20|Pernal  |    |30|Clerk |
SELECT ID, JOB                           |30|Marenghi|    |30|Mgr   |
FROM   STAFF                             +-----------+    |40|Sales |
WHERE  ID BETWEEN 20 AND 50                               |50|Mgr   |
UNION ALL                                                 +---------+
SELECT ID, 'Clerk' AS JOB
FROM   STAFF
WHERE  ID = 30;
```

Figure 493, Sample Views used in Join Examples

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.

Join Syntax

DB2 UDB SQL comes with two quite different ways to represent a join. Both syntax styles will be shown throughout this section though, in truth, one 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.

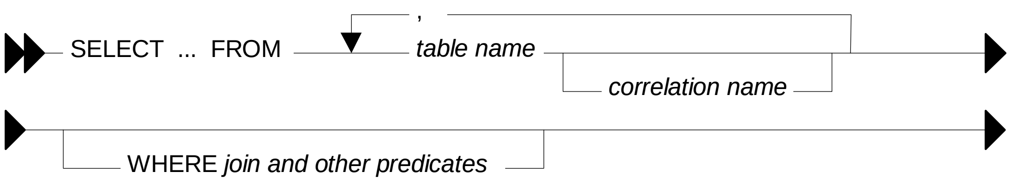

Figure 494, Join Syntax #1

Here are some sample joins:

```
SELECT   V1.ID                                        JOIN ANSWER
        ,V1.NAME                                      =================
        ,V2.JOB                                       ID NAME     JOB
FROM     STAFF_V1 V1                                  -- -------- -----
        ,STAFF_V2 V2                                  20 Pernal   Sales
WHERE    V1.ID = V2.ID                                30 Marenghi Clerk
ORDER BY V1.ID                                        30 Marenghi Mgr
        ,V2.JOB;
```

Figure 495, Sample two-table join

```
SELECT   V1.ID                                        JOIN ANSWER
        ,V2.JOB                                       =================
        ,V3.NAME                                      ID JOB   NAME
FROM     STAFF_V1 V1                                  -- ----- --------
        ,STAFF_V2 V2                                  30 Clerk Marenghi
        ,STAFF_V1 V3                                  30 Mgr   Marenghi
WHERE    V1.ID = V2.ID
  AND    V2.ID = V3.ID
  AND    V3.NAME LIKE 'M%'
ORDER BY V1.NAME
        ,V2.JOB;
```

Figure 496, Sample three-table join

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.

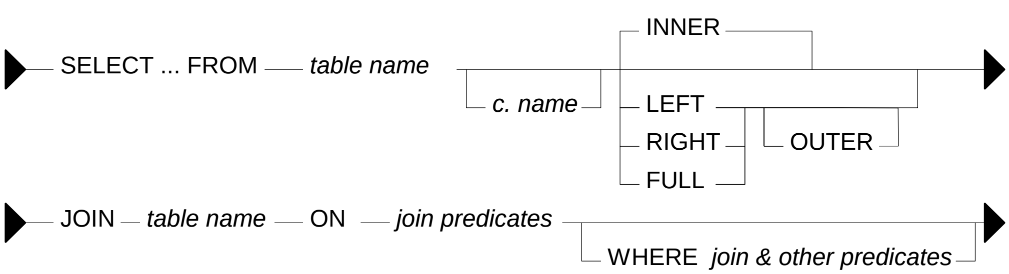

Figure 497, Join Syntax #2

The following sample joins are logically equivalent to the two given above:

```
SELECT   V1.ID                                        JOIN ANSWER
        ,V1.NAME                                      =================
        ,V2.JOB                                       ID NAME     JOB
FROM     STAFF_V1 V1                                  -- -------- -----
INNER JOIN                                            20 Pernal   Sales
         STAFF_V2 V2                                  30 Marenghi Clerk
ON       V1.ID = V2.ID                                30 Marenghi Mgr
ORDER BY V1.ID
        ,V2.JOB;
```

Figure 498, Sample two-table inner join

```
SELECT   V1.ID                           STAFF_V1        STAFF_V2
        ,V2.JOB                          +-----------+   +---------+
        ,V3.NAME                         |ID|NAME    |   |ID|JOB   |
FROM     STAFF_V1 V1                     |--|--------|   |--|------|
JOIN                                     |10|Sanders |   |20|Sales |
         STAFF_V2 V2                     |20|Pernal  |   |30|Clerk |
ON       V1.ID = V2.ID                   |30|Marenghi|   |30|Mgr   |
JOIN                                     +-----------+   |40|Sales |
         STAFF_V1 V3                                     |50|Mgr   |
ON       V2.ID = V3.ID               JOIN ANSWER         +---------+
WHERE    V3.NAME LIKE 'M%'           =================
ORDER BY V1.NAME                     ID JOB   NAME
        ,V2.JOB;                     -- ----- --------
                                     30 Clerk Marenghi
                                     30 Mgr   Marenghi
```

Figure 499, Sample three-table inner join

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:

```
SELECT   *                                  ANSWER
FROM     STAFF_V1 V1                        ====================
LEFT OUTER JOIN                             ID NAME     ID JOB
         STAFF_V2 V2                        -- -------- -- -----
ON       1       = 1                        10 Sanders  -  -
AND      V1.ID   = V2.ID                    20 Pernal   20 Sales
ORDER BY V1.ID                              30 Marenghi 30 Clerk
        ,V2.JOB;                            30 Marenghi 30 Mgr
```

Figure 500, Sample Views used in Join Examples

Now lets replace the second ON check with a WHERE check:

```
SELECT   *                                  ANSWER
FROM     STAFF_V1 V1                        ====================
LEFT OUTER JOIN                             ID NAME     ID JOB
         STAFF_V2 V2                        -- -------- -- -----
ON       1       = 1                        20 Pernal   20 Sales
WHERE    V1.ID   = V2.ID                    30 Marenghi 30 Clerk
ORDER BY V1.ID                              30 Marenghi 30 Mgr
        ,V2.JOB;
```

Figure 501, Sample Views used in Join Examples

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 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.

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          STAFF_V2                                  INNER-JOIN ANSWER
+-----------+     +---------+                               ===================
|ID|NAME    |     |ID|JOB   |     Join on ID                ID NAME     ID JOB
|--|--------|     |--|------|     ==========>               -- -------- -- -----
|10|Sanders |     |20|Sales |                               20 Pernal   20 Sales
|20|Pernal  |     |30|Clerk |                               30 Marenghi 30 Clerk
|30|Marenghi|     |30|Mgr   |                               30 Marenghi 30 Mgr
+-----------+     |40|Sales |
                  |50|Mgr   |
                  +---------+
```

Figure 502, Example of Inner Join

```
SELECT   *                                                  ANSWER
FROM     STAFF_V1 V1                                        ===================
        ,STAFF_V2 V2                                        ID NAME     ID JOB
WHERE    V1.ID = V2.ID                                      -- -------- -- -----
ORDER BY V1.ID                                              20 Pernal   20 Sales
        ,V2.JOB;                                            30 Marenghi 30 Clerk
                                                            30 Marenghi 30 Mgr
```

Figure 503, Inner Join SQL (1 of 2)

```
SELECT   *                                                  ANSWER
FROM     STAFF_V1 V1                                        ===================
INNER JOIN                                                  ID NAME     ID JOB
         STAFF_V2 V2                                        -- -------- -- -----
ON       V1.ID = V2.ID                                      20 Pernal   20 Sales
ORDER BY V1.ID                                              30 Marenghi 30 Clerk
        ,V2.JOB;                                            30 Marenghi 30 Mgr
```

Figure 504, Inner Join SQL (2 of 2)

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:

```
SELECT   *                                            ANSWER
FROM     STAFF_V1 V1                                  ====================
INNER JOIN                                            ID NAME     ID JOB
         STAFF_V2 V2                                  -- -------- -- -----
ON       V1.ID   = V2.ID                              20 Pernal   20 Sales
AND      V2.JOB <> 'Mgr'                              30 Marenghi 30 Clerk
ORDER BY V1.ID
        ,V2.JOB;
```

Figure 505, Inner join, using ON check

Here is the same query written using a WHERE check

```
SELECT   *                                            ANSWER
FROM     STAFF_V1 V1                                  ====================
INNER JOIN                                            ID NAME     ID JOB
         STAFF_V2 V2                                  -- -------- -- -----
ON       V1.ID   = V2.ID                              20 Pernal   20 Sales
WHERE    V2.JOB <> 'Mgr'                              30 Marenghi 30 Clerk
ORDER BY V1.ID
        ,V2.JOB;
```

Figure 506, Inner join, using WHERE check

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         STAFF_V2                          LEFT-OUTER-JOIN ANSWER
+-----------+    +---------+                       ======================
|ID|NAME    |    |ID|JOB   |                         ID NAME     ID JOB
|--|--------|    |--|------|      =========>         -- -------- -- -----
|10|Sanders |    |20|Sales |                         10 Sanders  -  -
|20|Pernal  |    |30|Clerk |                         20 Pernal   20 Sales
|30|Marenghi|    |30|Mgr   |                         30 Marenghi 30 Clerk
+-----------+    |40|Sales |                         30 Marenghi 30 Mgr
                 |50|Mgr   |
                 +---------+
```

Figure 507, Example of Left Outer Join

```
SELECT   *
FROM     STAFF_V1 V1
LEFT OUTER JOIN
         STAFF_V2 V2
ON       V1.ID = V2.ID
ORDER BY 1,4;
```

Figure 508, Left Outer Join SQL (1 of 2)

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:

```
SELECT   V1.*                                      <== This join gets all
        ,V2.*                                          rows in STAFF_V1
FROM     STAFF_V1 V1                                   that match rows
        ,STAFF_V2 V2                                   in STAFF_V2.
WHERE    V1.ID = V2.ID
UNION
SELECT   V1.*                                      <== This query gets
        ,CAST(NULL AS SMALLINT) AS ID                  all the rows in
        ,CAST(NULL AS CHAR(5))  AS JOB                 STAFF_V1 with no
FROM     STAFF_V1 V1                                   matching rows
WHERE    V1.ID NOT IN                                  in STAFF_V2.
        (SELECT ID FROM STAFF_V2)
ORDER BY 1,4;
```

Figure 509, Left Outer Join SQL (2 of 2)

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:

```
SELECT   *                                       ANSWER
FROM     STAFF_V1 V1                             ====================
LEFT OUTER JOIN                                  ID NAME     ID JOB
         STAFF_V2 V2                             -- -------- -- -----
ON       V1.ID   = V2.ID                         10 Sanders  -  -
AND      V2.JOB <> 'Mgr'                         20 Pernal   20 Sales
ORDER BY V1.ID                                   30 Marenghi 30 Clerk
        ,V2.JOB;
```

Figure 510, ON check on table being joined to

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:

```
SELECT   *                                       ANSWER
FROM     STAFF_V1 V1                             ====================
LEFT OUTER JOIN                                  ID NAME     ID JOB
         STAFF_V2 V2                             -- -------- -- -----
ON       V1.ID   = V2.ID                         20 Pernal   20 Sales
WHERE    V2.JOB <> 'Mgr'                         30 Marenghi 30 Clerk
ORDER BY V1.ID
        ,V2.JOB;
```

Figure 511, WHERE check on table being joined to (1 of 2)

We could make the WHERE equivalent to the ON, if we also checked for nulls:

```
SELECT   *                                       ANSWER
FROM     STAFF_V1 V1                             ====================
LEFT OUTER JOIN                                  ID NAME     ID JOB
         STAFF_V2 V2                             -- -------- -- -----
ON       V1.ID   = V2.ID                         10 Sanders  -  -
WHERE   (V2.JOB <> 'Mgr'                         20 Pernal   20 Sales
   OR    V2.JOB IS  NULL)                        30 Marenghi 30 Clerk
ORDER BY V1.ID
        ,V2.JOB;
```

Figure 512, WHERE check on table being joined to (2 of 2)

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:

```
SELECT   *                                       ANSWER
FROM     STAFF_V1 V1                             ====================
LEFT OUTER JOIN                                  ID NAME     ID JOB
         STAFF_V2 V2                             -- -------- -- -----
ON       V1.ID   = V2.ID                         10 Sanders  -  -
AND      V1.NAME > 'N'                           20 Pernal   20 Sales
ORDER BY V1.ID                                   30 Marenghi -  -
        ,V2.JOB;
```

Figure 513, ON check on table being joined from

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:

```
SELECT   *                                            ANSWER
FROM     STAFF_V1 V1                                  ====================
LEFT OUTER JOIN                                       ID NAME     ID JOB
         STAFF_V2 V2                                  -- -------- -- -----
ON       V1.ID    = V2.ID                             10 Sanders  -  -
WHERE    V1.NAME > 'N'                                20 Pernal   20 Sales
ORDER BY V1.ID
        ,V2.JOB;
```

Figure 514, WHERE check on table being joined from

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.

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         STAFF_V2                          RIGHT-OUTER-JOIN ANSWER
+-----------+    +---------+                       =======================
|ID|NAME    |    |ID|JOB   |                          ID NAME     ID JOB
|--|--------|    |--|------|      =========>          -- -------- -- -----
|10|Sanders |    |20|Sales |                          20 Pernal   20 Sales
|20|Pernal  |    |30|Clerk |                          30 Marenghi 30 Clerk
|30|Marenghi|    |30|Mgr   |                          30 Marenghi 30 Mgr
+-----------+    |40|Sales |                          -  -        40 Sales
                 |50|Mgr   |                          -  -        50 Mgr
                 +---------+
```

Figure 515, Example of Right Outer Join

```
SELECT   *                                            ANSWER
FROM     STAFF_V1 V1                                  ====================
RIGHT OUTER JOIN                                      ID NAME     ID JOB
         STAFF_V2 V2                                  -- -------- -- -----
ON       V1.ID = V2.ID                                20 Pernal   20 Sales
ORDER BY V2.ID                                        30 Marenghi 30 Clerk
        ,V2.JOB;                                      30 Marenghi 30 Mgr
                                                      -  -        40 Sales
                                                      -  -        50 Mgr
```

Figure 516, Right Outer Join SQL (1 of 2)

It is also possible to code a right outer join using the standard inner join syntax:

```
SELECT   V1.*                                         ANSWER
        ,V2.*                                         ====================
FROM     STAFF_V1 V1                                  ID NAME     ID JOB
        ,STAFF_V2 V2                                  -- -------- -- -----
WHERE    V1.ID = V2.ID                                20 Pernal   20 Sales
UNION                                                 30 Marenghi 30 Clerk
SELECT   CAST(NULL AS SMALLINT)    AS ID              30 Marenghi 30 Mgr
        ,CAST(NULL AS VARCHAR(9))  AS NAME            -  -        40 Sales
        ,V2.*                                         -  -        50 Mgr
FROM     STAFF_V2 V2
WHERE    V2.ID NOT IN
        (SELECT ID FROM STAFF_V1)
ORDER BY 3,4;
```

Figure 517, Right Outer Join SQL (2 of 2)

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 page 182), except that the relevant tables are reversed.

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          STAFF_V2                              FULL-OUTER-JOIN ANSWER
+-----------+     +---------+                           ======================
|ID|NAME    |     |ID|JOB   |                             ID NAME     ID JOB
|--|--------|     |--|------|      =========>             -- -------- -- -----
|10|Sanders |     |20|Sales |                             10 Sanders  -  -
|20|Pernal  |     |30|Clerk |                             20 Pernal   20 Sales
|30|Marenghi|     |30|Mgr   |                             30 Marenghi 30 Clerk
+-----------+     |40|Sales |                             30 Marenghi 30 Mgr
                  |50|Mgr   |                              - -        40 Sales
                  +---------+                              - -        50 Mgr
```

Figure 518, Example of Full Outer Join

```
SELECT   *                                                ANSWER
FROM     STAFF_V1 V1                                      ====================
FULL OUTER JOIN                                           ID NAME     ID JOB
         STAFF_V2 V2                                      -- -------- -- -----
ON       V1.ID = V2.ID                                    10 Sanders  -  -
ORDER BY V1.ID                                            20 Pernal   20 Sales
        ,V2.ID                                            30 Marenghi 30 Clerk
        ,V2.JOB;                                          30 Marenghi 30 Mgr
                                                           - -        40 Sales
                                                           - -        50 Mgr
```

Figure 519, Full Outer Join SQL

Here is the same done using the standard inner join syntax:

```
SELECT   V1.*                                             ANSWER
        ,V2.*                                             ====================
FROM     STAFF_V1 V1                                      ID NAME     ID JOB
        ,STAFF_V2 V2                                      -- -------- -- -----
WHERE    V1.ID = V2.ID                                    10 Sanders  -  -
UNION                                                     20 Pernal   20 Sales
SELECT   V1.*                                             30 Marenghi 30 Clerk
        ,CAST(NULL AS SMALLINT) AS ID                     30 Marenghi 30 Mgr
        ,CAST(NULL AS CHAR(5))  AS JOB                     - -        40 Sales
FROM     STAFF_V1 V1                                       - -        50 Mgr
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;
```

Figure 520, Full Outer Join SQL

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:

```
SELECT   *                                   ANSWER
FROM     STAFF_V1 V1                         ====================
FULL OUTER JOIN                              ID NAME     ID JOB
         STAFF_V2 V2                         -- -------- -- -----
ON       V1.ID = V2.ID                       10 Sanders   - -
ORDER BY V1.ID                               20 Pernal   20 Sales
        ,V2.ID                               30 Marenghi 30 Clerk
        ,V2.JOB;                             30 Marenghi 30 Mgr
                                              - -        40 Sales
                                              - -        50 Mgr
```

Figure 521, Full Outer Join, match on keys

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:

```
SELECT   *                                   ANSWER
FROM     STAFF_V1 V1                         ====================
FULL OUTER JOIN                              ID NAME     ID JOB
         STAFF_V2 V2                         -- -------- -- -----
ON       V1.ID = V2.ID                       10 Sanders   - -
AND      V1.ID > 20                          20 Pernal    - -
ORDER BY V1.ID                               30 Marenghi 30 Clerk
        ,V2.ID                               30 Marenghi 30 Mgr
        ,V2.JOB;                              - -        20 Sales
                                              - -        40 Sales
                                              - -        50 Mgr
```

Figure 522, Full Outer Join, match on keys > 20

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:

```
SELECT   *                                   ANSWER
FROM     STAFF_V1 V1                         ====================
FULL OUTER JOIN                              ID NAME     ID JOB
         STAFF_V2 V2                         -- -------- -- -----
ON       V1.ID = V2.ID                       10 Sanders   - -
AND      +1 = -1                             20 Pernal    - -
ORDER BY V1.ID                               30 Marenghi  - -
        ,V2.ID                                - -        20 Sales
        ,V2.JOB;                              - -        30 Clerk
                                              - -        30 Mgr
                                              - -        40 Sales
                                              - -        50 Mgr
```

Figure 523, Full Outer Join, match on keys (no rows match)

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:

```
SELECT    *                                           ANSWER
FROM      STAFF_V1 V1                                 ====================
FULL OUTER JOIN                                       ID NAME     ID JOB
          STAFF_V2 V2                                 -- -------- -- -----
ON        +1 = -1                                     10 Sanders   - -
ORDER BY V1.ID                                        20 Pernal    - -
        ,V2.ID                                        30 Marenghi  - -
        ,V2.JOB;                                       - -        20 Sales
                                                       - -        30 Clerk
                                                       - -        30 Mgr
                                                       - -        40 Sales
                                                       - -        50 Mgr
```

Figure 524, Full Outer Join, don't match on keys (no rows match)

What happens if everything matches and we don't identify the join fields? The result in a Cartesian Product:

```
SELECT    *                                           ANSWER
FROM      STAFF_V1 V1                                 ====================
FULL OUTER JOIN                                       ID NAME     ID JOB
          STAFF_V2 V2                                 -- -------- -- -----
ON        +1 <> -1                                    10 Sanders  20 Sales
ORDER BY V1.ID                                        10 Sanders  30 Clerk
        ,V2.ID                                        10 Sanders  30 Mgr
        ,V2.JOB;                                      10 Sanders  40 Sales
                                                      10 Sanders  50 Mgr
                                                      20 Pernal   20 Sales
STAFF_V1         STAFF_V2                             20 Pernal   30 Clerk
+-----------+    +---------+                          20 Pernal   30 Mgr
|ID|NAME    |    |ID|JOB   |                          20 Pernal   40 Sales
|--|--------|    |--|------|                          20 Pernal   50 Mgr
|10|Sanders |    |20|Sales |                          30 Marenghi 20 Sales
|20|Pernal  |    |30|Clerk |                          30 Marenghi 30 Clerk
|30|Marenghi|    |30|Mgr   |                          30 Marenghi 30 Mgr
+-----------+    |40|Sales |                          30 Marenghi 40 Sales
                 |50|Mgr   |                          30 Marenghi 50 Mgr
                 +---------+
```

Figure 525, Full Outer Join, don't match on keys (all rows match)

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:

```
SELECT    *                                           ANSWER
FROM      STAFF_V1 V1                                 ====================
FULL JOIN                                             ID NAME     ID JOB
          STAFF_V2 V2                                 -- -------- -- -----
ON        V1.ID = V2.ID                               20 Pernal   20 Sales
WHERE     V1.ID = V2.ID                               30 Marenghi 30 Clerk
ORDER BY 1,3,4;                                       30 Marenghi 30 Mgr
```

Figure 526, Full Outer Join, turned into an inner join by WHERE

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        STAFF_V2
 +-----------+   +---------+                         ANSWER
 |ID|NAME    |   |ID|JOB   |      OUTER-JOIN CRITERIA     ============
 |--|--------|   |--|------|      ==================>     ???, DEPENDS
 |10|Sanders |   |20|Sales |         V1.ID = V2.ID
 |20|Pernal  |   |30|Clerk |         V1.ID < 30
 |30|Marenghi|   |30|Mgr   |
 +-----------+   |40|Sales |
                 |50|Mgr   |
                 +---------+
```

Figure 527, Outer join V1.ID < 30, sample data

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):

```
SELECT   *                                          ANSWER
FROM     STAFF_V1 V1                                ====================
FULL JOIN                                           ID NAME     ID JOB
         STAFF_V2 V2                                -- -------- -- -----
ON       V1.ID = V2.ID                              10 Sanders  -  -
WHERE    V1.ID < 30                                 20 Pernal   20 Sales
ORDER BY 1,3,4;
```

Figure 528, Outer join V1.ID < 30, check applied in WHERE (after join)

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:

```
SELECT   *                                          ANSWER
FROM     STAFF_V1 V1                                ====================
FULL JOIN                                           ID NAME     ID JOB
         STAFF_V2 V2                                -- -------- -- -----
ON       V1.ID = V2.ID                              10 Sanders  -  -
AND      V1.ID < 30                                 20 Pernal   20 Sales
ORDER BY 1,3,4;                                     30 Marenghi -  -
                                                    -  -        30 Clerk
                                                    -  -        30 Mgr
                                                    -  -        40 Sales
                                                    -  -        50 Mgr
```

Figure 529, Outer join V1.ID < 30, check applied in ON (during join)

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:

```
SELECT   *                                          ANSWER
FROM     (SELECT *                                  ====================
          FROM   STAFF_V1                           ID NAME     ID JOB
          WHERE  ID < 30) AS V1                     -- -------- -- -----
FULL OUTER JOIN                                     10 Sanders  -  -
         STAFF_V2 V2                                20 Pernal   20 Sales
ON       V1.ID = V2.ID                              -  -        30 Clerk
ORDER BY 1,3,4;                                     -  -        30 Mgr
                                                    -  -        40 Sales
                                                    -  -        50 Mgr
```

Figure 530, Outer join V1.ID < 30, check applied in WHERE (before join)

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:

```
SELECT   *                                          ANSWER
FROM     STAFF_V1 V1                                ====================
FULL OUTER JOIN                                     ID NAME     ID JOB
         STAFF_V2 V2                                -- -------- -- -----
ON       V1.ID  = V2.ID                             10 Sanders  -  -
WHERE    V1.ID  < 30                                20 Pernal   20 Sales
   OR    V1.ID IS NULL                              -  -        40 Sales
ORDER BY 1,3,4;                                     -  -        50 Mgr
```

Figure 531, Outer join V1.ID < 30, (gives wrong answer - see text)

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:

```
SELECT   *                                          ANSWER
FROM     STAFF_V1 V1                                ====================
FULL OUTER JOIN                                     ID NAME     ID JOB
         STAFF_V2 V2                                -- -------- -- -----
ON       V1.ID  = V2.ID                             10 Sanders  -  -
WHERE    V1.ID  < 30                                20 Pernal   20 Sales
   OR    V1.ID  = V2.ID                             30 Marenghi 30 Clerk
   OR    V1.ID IS NULL                              30 Marenghi 30 Mgr
ORDER BY 1,3,4;                                     -  -        40 Sales
                                                    -  -        50 Mgr
```

Figure 532, Outer join V1.ID < 30, (gives wrong answer - see text)

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.

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        STAFF_V2                           CARTESIAN-PRODUCT
+-----------+   +---------+                        ====================
|ID|NAME    |   |ID|JOB   |                        ID NAME     ID JOB
|--|--------|   |--|------|         =========>     -- -------- -- -----
|10|Sanders |   |20|Sales |                        ID NAME     ID JOB
|20|Pernal  |   |30|Clerk |                        -- -------- -- -----
|30|Marenghi|   |30|Mgr   |                        10 Sanders  20 Sales
+-----------+   |40|Sales |                        10 Sanders  30 Clerk
                |50|Mgr   |                        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
```

Figure 533, Example of Cartesian Product

Writing a Cartesian Product is simplicity itself. One simply omits the WHERE conditions:

```
SELECT   *
FROM     STAFF_V1 V1
        ,STAFF_V2 V2
ORDER BY V1.ID
        ,V2.ID
        ,V2.JOB;
```

Figure 534, Cartesian Product SQL (1 of 2)

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:

```
SELECT   *
FROM     STAFF_V1 V1
INNER JOIN
         STAFF_V2 V2
ON       'A' <> 'B'
ORDER BY V1.ID
        ,V2.ID
        ,V2.JOB;
```

Figure 535, Cartesian Product SQL (2 of 2)

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:

```
SELECT   V2A.ID                                             ANSWER
        ,V2A.JOB                                            ===========
        ,V2B.ID                                             ID JOB   ID
FROM     STAFF_V2 V2A                                       -- ----- --
        ,STAFF_V2 V2B                                       20 Sales 20
WHERE    V2A.JOB = V2B.JOB                                  20 Sales 40
  AND    V2A.ID  < 40                                       30 Clerk 30
ORDER BY V2A.ID                                             30 Mgr   30
        ,V2B.ID;                                            30 Mgr   50
```

Figure 536, Partial Cartesian Product SQL

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:

```
SELECT   V2.JOB                                             ANSWER
        ,COUNT(*) AS #ROWS                                  ===========
FROM     STAFF_V1 V1                                        JOB   #ROWS
        ,STAFF_V2 V2                                        ----- -----
GROUP BY V2.JOB                                             Clerk     3
ORDER BY #ROWS                                              Mgr       6
        ,V2.JOB;                                            Sales     6
```

Figure 537, Partial Cartesian Product SQL, with GROUP BY

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.

HINT: 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.

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.

```
SELECT   COALESCE(V1.ID,V2.ID) AS ID                    ANSWER
        ,COALESCE(V1.NAME,'?') AS NAME                  =================
        ,V2.JOB                                         ID NAME     JOB
FROM     STAFF_V1 V1                                    -- -------- -----
FULL OUTER JOIN                                         10 Sanders  -
         STAFF_V2 V2                                    20 Pernal   Sales
ON       V1.ID = V2.ID                                  30 Marenghi Clerk
ORDER BY V1.ID                                          30 Marenghi Mgr
        ,V2.JOB;                                        40 ?        Sales
                                                        50 ?        Mgr
```

Figure 538, Use of COALESCE function in outer join

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         STAFF_V2                             ANSWER
+-----------+    +---------+    NON-MATCHING          ===================
|ID|NAME    |    |ID|JOB   |     OUTER-JOIN           ID NAME    ID JOB
|--|--------|    |--|------|    ===========>          -- ------- -- -----
|10|Sanders |    |20|Sales |                          10 Sanders -  -
|20|Pernal  |    |30|Clerk |                          -  -       40 Sales
|30|Marenghi|    |30|Mgr   |                          -  -       50 Mgr
+-----------+    |40|Sales |
                 |50|Mgr   |
                 +---------+
```

Figure 539, Example of outer join, only getting the non-matching rows

One way to express the above is to use the standard inner-join syntax:

```
SELECT   V1.*                                     <== Get all the rows
        ,CAST(NULL AS SMALLINT) AS ID                 in STAFF_V1 that
        ,CAST(NULL AS CHAR(5))  AS JOB                have no matching
FROM     STAFF_V1 V1                                  row in STAFF_V2.
WHERE    V1.ID NOT IN
        (SELECT ID FROM STAFF_V2)
UNION
SELECT   CAST(NULL AS SMALLINT)   AS ID           <== Get all the rows
        ,CAST(NULL AS VARCHAR(9)) AS NAME             in STAFF_V2 that
        ,V2.*                                         have no matching
FROM     STAFF_V2 V2                                  row in STAFF_V1.
WHERE    V2.ID NOT IN
        (SELECT ID FROM STAFF_V1)
ORDER BY 1,3,4;
```

Figure 540, Outer Join SQL, getting only non-matching rows

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:

```
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                                    ANSWER
   OR    V2.FLAG IS NULL                 =============================
ORDER BY V1.ID                           ID NAME    FLAG ID JOB   FLAG
        ,V2.ID                           -- ------- ---- -- ----- ----
        ,V2.JOB;                         10 Sanders V1    -  -     -
                                         -  -       -    40 Sales V2
                                         -  -       -    50 Mgr   V2
```

Figure 541, Outer Join SQL, getting only non-matching rows

Alternatively, one can use two common table expressions to do the same job:

```
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                                                ANSWER
FULL OUTER JOIN                          =============================
       V2 V2                             ID NAME    FLAG ID JOB   FLAG
ON       V1.ID  =  V2.ID                 -- ------- ---- -- ----- ----
WHERE    V1.FLAG IS NULL                 10 Sanders V1    -  -     -
   OR    V2.FLAG IS NULL                 -  -       -    40 Sales V2
ORDER BY V1.ID, V2.ID, V2.JOB;           -  -       -    50 Mgr   V2
```

Figure 542, Outer Join SQL, getting only non-matching rows

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:

```
SELECT   *                                 STAFF_V1        STAFF_V2
FROM     STAFF_V1 V1                       +-----------+   +---------+
FULL OUTER JOIN                            |ID|NAME    |   |ID|JOB   |
         STAFF_V2 V2                       |--|--------|   |--|------|
ON       V1.ID  =  V2.ID                   |10|Sanders |   |20|Sales |
WHERE    V1.ID IS NULL                     |20|Pernal  |   |30|Clerk |
   OR    V2.ID IS NULL                     |30|Marenghi|   |30|Mgr   |
ORDER BY V1.ID                             +-----------+   |40|Sales |
        ,V2.ID                                             |50|Mgr   |
        ,V2.JOB;                                           +---------+
```

Figure 543, Outer Join SQL, getting only non-matching rows

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       STAFF_V2                        ANSWER
+-----------+  +---------+   LEFT OUTER JOIN   ===================
|ID|NAME    |  |ID|JOB   |   ==============>   ID NAME    ID JOB
|--|--------|  |--|------|   V1.ID  = V2.ID    -- ------- -- -----
|10|Sanders |  |20|Sales |   V1.ID <> 30       10 Sanders -  -
|20|Pernal  |  |30|Clerk |                     20 Pernal  20 Sales
|30|Marenghi|  |30|Mgr   |
+-----------+  |40|Sales |
               |50|Mgr   |
               +---------+
```

Figure 544, Left outer join example

Here is one way to express the above as a query:

```
SELECT   V1.ID                                         ANSWER
        ,V1.NAME                                       =================
        ,V2.JOB                                        ID NAME     JOB
FROM     STAFF_V1 V1                                   -- -------- -----
LEFT OUTER JOIN                                        10 Sanders  -
         STAFF_V2 V2                                   20 Pernal   Sales
ON       V1.ID  = V2.ID
WHERE    V1.ID <> 30
ORDER BY V1.ID ;
```

Figure 545, Outer Join done in FROM phrase of SQL

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:

```
SELECT   V1.ID                                         ANSWER
        ,V1.NAME                                       =================
        ,(SELECT  V2.JOB                               ID NAME     JB
          FROM    STAFF_V2 V2                          -- -------- -----
          WHERE   V1.ID = V2.ID) AS JB                 10 Sanders  -
FROM     STAFF_V1 V1                                   20 Pernal   Sales
WHERE    V1.ID <> 30
ORDER BY V1.ID;
```

Figure 546, Outer Join done in SELECT phrase of SQL

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):

```
SELECT   V1.ID                                         ANSWER
        ,V1.NAME                                       =================
        ,(SELECT  V2.JOB                               ID NAME     JB
          FROM    STAFF_V2 V2                          -- -------- -----
          WHERE   V1.ID = V2.ID) AS JB                 10 Sanders  -
FROM     STAFF_V1 V1                                   20 Pernal   Sales
ORDER BY V1.ID;                                        <error>
```

Figure 547, Outer Join done in SELECT phrase of SQL - gets error

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:

```
SELECT   V1.ID                                         ANSWER
        ,V1.NAME                                       =================
        ,(SELECT  MAX(V2.JOB)                          ID NAME     JB
          FROM    STAFF_V2 V2                          -- -------- -----
          WHERE   V1.ID = V2.ID) AS JB                 10 Sanders  -
FROM     STAFF_V1 V1                                   20 Pernal   Sales
ORDER BY V1.ID;                                        30 Marenghi Mgr
```

Figure 548, Outer Join done in SELECT phrase of SQL - fixed

The above is equivalent to the following query:

```
SELECT   V1.ID                                          ANSWER
        ,V1.NAME                                        =================
        ,MAX(V2.JOB) AS JB                              ID NAME     JB
FROM     STAFF_V1 V1                                    -- -------- -----
LEFT OUTER JOIN                                         10 Sanders  -
         STAFF_V2 V2                                    20 Pernal   Sales
ON       V1.ID  = V2.ID                                 30 Marenghi Mgr
GROUP BY V1.ID
        ,V1.NAME
ORDER BY V1.ID ;
```

Figure 549, Same as prior query - using join and GROUP BY

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:

```
SELECT   V2.ID                                                ANSWER
        ,CASE                                                 ===========
            WHEN V2.JOB <> 'Mgr'                              ID J2
            THEN V2.JOB                                       -- --------
            ELSE (SELECT V1.NAME                              20 Sales
                  FROM   STAFF_V1 V1                          30 Clerk
                  WHERE  V1.ID = V2.ID)                       30 Marenghi
         END AS J2                                            40 Sales
FROM     STAFF_V2 V2                                          50 -
ORDER BY V2.ID
        ,J2;
```

Figure 550, Sample Views used in Join Examples

Multiple Columns

If you want to retrieve two columns using this type of join, you need to have two independent nested table expressions:

```
SELECT   V2.ID                                        ANSWER
        ,V2.JOB                                       ====================
        ,(SELECT  V1.NAME                             ID JOB   NAME     N2
          FROM    STAFF_V1 V1                         -- ----- -------- --
          WHERE   V2.ID = V1.ID)                      20 Sales Pernal    6
        ,(SELECT  LENGTH(V1.NAME) AS N2               30 Clerk Marenghi  8
          FROM    STAFF_V1 V1                         30 Mgr   Marenghi  8
          WHERE   V2.ID = V1.ID)                      40 Sales -         -
FROM     STAFF_V2 V2                                  50 Mgr   -         -
ORDER BY V2.ID
        ,V2.JOB;
```

Figure 551, Outer Join done in SELECT, 2 columns

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:

```
SELECT   V2.ID                                         ANSWER
        ,V2.JOB                                        ====================
        ,V1.NAME                                       ID JOB   NAME     N2
        ,LENGTH(V1.NAME) AS N2                         -- ----- -------- --
FROM     STAFF_V2 V2                                   20 Sales Pernal    6
LEFT OUTER JOIN                                        30 Clerk Marenghi  8
         STAFF_V1 V1                                   30 Mgr   Marenghi  8
ON       V2.ID = V1.ID                                 40 Sales -         -
ORDER BY V2.ID                                         50 Mgr   -         -
        ,V2.JOB;
```

Figure 552, Outer Join done in FROM, 2 columns

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:

```
SELECT   V1.ID                                           ANSWER
        ,V1.NAME                                         ==================
        ,(SELECT  SUM(X1.ID)                             ID NAME     SUM_ID
          FROM    STAFF_V1 X1                            -- -------- ------
          WHERE   X1.ID <= V1.ID                         10 Sanders      10
         )AS SUM_ID                                      20 Pernal       30
FROM     STAFF_V1 V1                                     30 Marenghi     60
ORDER BY V1.ID
        ,V2.JOB;
```

Figure 553, Running total, using JOIN in SELECT

An easier way to do the same as the above is to use an OLAP function:

```
SELECT   V1.ID                                           ANSWER
        ,V1.NAME                                         ==================
        ,SUM(ID) OVER(ORDER BY ID) AS SUM_ID             ID NAME     SUM_ID
FROM     STAFF_V1 V1                                     -- -------- ------
ORDER BY V1.ID;                                          10 Sanders      10
                                                         20 Pernal       30
                                                         30 Marenghi     60
```

Figure 554, Running total, using OLAP function

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        STAFF_V2                                   ANSWER
+-----------+   +---------+                                =================
|ID|NAME    |   |ID|JOB   |    OUTER-JOIN CRITERIA         ID NAME     JOB
|--|--------|   |--|------|    ==================>         -- -------- -----
|10|Sanders |   |20|Sales |    V1.ID      = V2.ID          10 Sanders  -
|20|Pernal  |   |30|Clerk |    V2.JOB LIKE 'S%'            20 Pernal   Sales
|30|Marenghi|   |30|Mgr   |                                30 Marenghi -
+-----------+   |40|Sales |
                |50|Mgr   |
                +---------+
```

Figure 555, Outer join, with WHERE filter

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:

```
SELECT   V1.ID                                          ANSWER (WRONG)
        ,V1.NAME                                        =================
        ,V2.JOB                                         ID NAME     JOB
FROM     STAFF_V1 V1                                    -- -------- -----
LEFT OUTER JOIN                                         20 Pernal   Sales
         STAFF_V2 V2
ON       V1.ID     = V2.ID
WHERE    V2.JOB LIKE 'S%'
ORDER BY V1.ID
        ,V2.JOB;
```

Figure 556, Outer Join, WHERE done after - wrong

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:

```
SELECT   V1.ID                                          ANSWER
        ,V1.NAME                                        =================
        ,V2.JOB                                         ID NAME     JOB
FROM     STAFF_V1 V1                                    -- -------- -----
LEFT OUTER JOIN                                         10 Sanders  -
        (SELECT  *                                      20 Pernal   Sales
         FROM    STAFF_V2                               30 Marenghi -
         WHERE   JOB LIKE 'S%'
        )AS V2
ON       V1.ID = V2.ID
ORDER BY V1.ID
        ,V2.JOB;
```

Figure 557, Outer Join, WHERE done before - correct

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:

```
SELECT   V1.ID                                          ANSWER
        ,V1.NAME                                        =================
        ,(SELECT V2.JOB                                 ID NAME     JOB
          FROM   STAFF_V2 V2                            -- -------- -----
          WHERE  V1.ID     = V2.ID                      10 Sanders  -
            AND  V2.JOB LIKE 'S%')                      20 Pernal   Sales
FROM     STAFF_V1 V1                                    30 Marenghi -
ORDER BY V1.ID
        ,JOB;
```

Figure 558, Outer Join, WHERE done independently - correct

Joins - Things to Remember

- You get nulls in an outer join, whether you want them or not, because the fields in non-matching rows are set to null. If they bug you, use the COALESCE function to remove them. See page 190 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 page 184). 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 (see page 182).

- 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 page 191), 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.

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:

```
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
==========================
EMPNO  PROJNO ACTNO FORMAT
------ ------ ----- ------
000010 MA2110    10 -
000070 -          - -
000130 -          - bitmap
000150 MA2112    60 bitmap
000150 MA2112   180 bitmap
000160 MA2113    60 -
```

Figure 559, Join from Employee to Activity and Photo

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:

```
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 -
```

Figure 560, Join from Employee to Activity, then from Activity to Photo

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 inner join, followed by an outer join, followed by an inner join. We are trying to do the following:

- Get a list of matching departments - based on some local predicates.
- For each matching department, get the related employees. If no employees exist, do not list the department (i.e. inner join).
- 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:

```
SELECT   ddd.deptno AS dp#
        ,eee.empno
        ,aaa.projno
        ,ppp.projname
FROM    (SELECT  *
         FROM    department
         WHERE   deptname     LIKE '%A%'
           AND   deptname NOT LIKE '%U%'
           AND   deptno          < 'E'
        )AS ddd
INNER JOIN
         employee   eee
ON       ddd.deptno       = eee.workdept
AND      eee.lastname  LIKE '%A%'
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 ddd.deptno
        ,eee.empno
        ,aaa.projno;
```

```
                                 ANSWER
=================================
DP# EMPNO  PROJNO PROJNAME
--- ------ ------ --------------
C01 000030 IF1000 QUERY SERVICES
C01 000130 IF1000 QUERY SERVICES
```

Figure 561, Complex join - wrong

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:

```
SELECT   ddd.deptno AS dp#
        ,eee.empno
        ,xxx.projno
        ,xxx.projname
FROM    (SELECT  *
         FROM    department
         WHERE   deptname     LIKE '%A%'
           AND   deptname NOT LIKE '%U%'
           AND   deptno          < 'E'
        )AS ddd
INNER JOIN
         employee   eee
ON       ddd.deptno       = eee.workdept
AND      eee.lastname  LIKE '%A%'
LEFT OUTER JOIN
        (SELECT   aaa.empno
                 ,aaa.emptime
                 ,aaa.projno
                 ,ppp.projname
         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 ddd.deptno
        ,eee.empno
        ,xxx.projno;
```

```
                                          ANSWER
================================
DP# EMPNO  PROJNO PROJNAME
--- ------ ------ --------------
C01 000030 IF1000 QUERY SERVICES
C01 000130 IF1000 QUERY SERVICES
D21 000070 -      -
D21 000240 -      -
```

Figure 562, Complex join - right

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.

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 full-select 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.

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:

```
CREATE TABLE table1                                      TABLE1     TABLE2
(t1a      CHAR(1)    NOT NULL                            +-------+  +-----------+
,t1b      CHAR(2)    NOT NULL                            |T1A|T1B|  |T2A|T2B|T2C|
,PRIMARY KEY(t1a));                                      |---|---|  |---|---|---|
COMMIT;                                                  |A  |AA |  |A  |A  |A  |
                                                         |B  |BB |  |B  |A  | - |
CREATE TABLE table2                                      |C  |CC |  +-----------+
(t2a      CHAR(1)    NOT NULL                            +-------+  "-" = 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);
```

Figure 563, Sample tables used in sub-query examples

Sub-query Flavours

Sub-query Syntax

A sub-query compares an expression against a full-select. The type of comparison done is a function of which, if any, keyword is used:

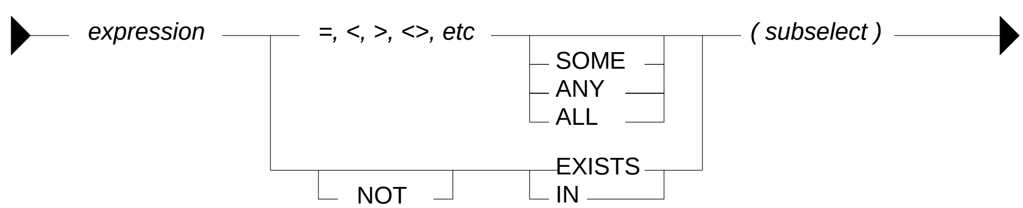

Figure 564, Sub-query syntax diagram

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.

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:

```
SELECT *                                                    ANSWER
FROM   table1                                               =======
WHERE  t1a =                                                T1A T1B
      (SELECT t2a                                           --- --
       FROM   table2                                        A   aa
       WHERE  t2a = 'A');

                                    SUB-Q   TABLE1      TABLE2
                                    RESLT   +-------+   +-----------+
                                    +---+   |T1A|T1B|   |T2A|T2B|T2C|
                                    |T2A|   |---|---|   |---|---|---|
                                    |---|   |A  |AA |   |A  |A  |A  |
                                    |A  |   |B  |BB |   |B  |A  | - |
                                    +---+   |C  |CC |   +-----------+
                                            +-------+   "-" = null
```

Figure 565, No keyword sub-query, works

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:

```
SELECT *                                                    ANSWER
FROM   table1                                               =======
WHERE  t1a =                                                <error>
      (SELECT t2a
       FROM   table2);
                                    SUB-Q   TABLE1      TABLE2
                                    RESLT   +-------+   +-----------+
                                    +---+   |T1A|T1B|   |T2A|T2B|T2C|
                                    |T2A|   |---|---|   |---|---|---|
                                    |---|   |A  |AA |   |A  |A  |A  |
                                    |A  |   |B  |BB |   |B  |A  | - |
                                    |B  |   |C  |CC |   +-----------+
                                    +---+   +-------+   "-" = null
```

Figure 566, No keyword sub-query, fails

NOTE: 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.

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:

```
SELECT *                          ANSWER     SUB-Q  TABLE1     TABLE2
FROM   table1                     =======    RESLT  +-------+  +-----------+
WHERE  t1a > ANY                  T1A T1B    +---+  |T1A|T1B|  |T2A|T2B|T2C|
      (SELECT t2a                 --- --     |T2A|  |---|---|  |---|---|---|
       FROM   table2);            B   BB     |---|  |A  |AA |  |A  |A  |A  |
                                  C   CC     |A  |  |B  |BB |  |B  |A  | - |
                                             |B  |  |C  |CC |  +-----------+
                                             +---+  +-------+  "-" = null
```

Figure 567, ANY sub-query

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:

```
SUB-QUERY CHECK       EQUIVALENT COLUMN FUNCTION
================      ============================
> ANY(sub-qurey)      > 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)
```

Figure 568, ANY and ALL vs. column functions

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:

```
SELECT *                                                   ANSWER     SUB-Q
FROM   table1                                              =======    RESLT
WHERE  t1a = ALL                                           T1A T1B    +---+
      (SELECT t2b                                          --- --     |T2B|
       FROM   table2                                       A   aa     |---|
       WHERE  t2b >= 'A');                                            |A  |
                                                                      |A  |
                                                                      +---+
```

Figure 569, ALL sub-query, with non-empty sub-query result

When the sub-query result consists of zero rows (i.e. an empty set) then all rows processed in TABLE1 are deemed to match:

```
SELECT *                                                   ANSWER     SUB-Q
FROM   table1                                              =======    RESLT
WHERE  t1a = ALL                                           T1A T1B    +---+
      (SELECT t2b                                          --- --     |T2B|
       FROM   table2                                       A   aa     |---|
       WHERE  t2b >= 'X');                                 B   BB     +---+
                                                           C   CC
```

Figure 570, ALL sub-query, with empty sub-query result

The above may seem a little unintuitive, but it actually makes sense, and is in accordance with how the NOT EXISTS sub-query (see page 203) 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:

```
SELECT *                                                       ANSWER
FROM   table1                                                  ======
WHERE  t1a = ALL                                               0 rows
      (SELECT t2b
       FROM   table2            SQ-#1  SQ-#2  TABLE1     TABLE2
       WHERE  t2b >= 'X')       RESLT  RESLT  +-------+  +-----------+
  AND  0 <>                     +---+  +---+  |T1A|T1B|  |T2A|T2B|T2C|
      (SELECT COUNT(*)          |T2B|  |(*)|  |---|---|  |---|---|---|
       FROM   table2            |---|  |---|  |A  |AA |  |A  |A  |A  |
       WHERE  t2b >= 'X');      +---+  |0  |  |B  |BB |  |B  |A  | - |
                                       +---+  |C  |CC |  +-----------+
                                              +-------+  "-" = null
```

Figure 571, ALL sub-query, with extra check for empty set

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.

WARNING: Observe that the ANY sub-query check returns false when used against an empty set, while a similar ALL check returns true.

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:

```
SELECT *                             ANSWER   TABLE1     TABLE2
FROM   table1                        =======  +-------+  +-----------+
WHERE  EXISTS                        T1A T1B  |T1A|T1B|  |T2A|T2B|T2C|
      (SELECT *                      --- --   |---|---|  |---|---|---|
       FROM   table2);               A   aa   |A  |AA |  |A  |A  |A  |
                                     B   BB   |B  |BB |  |B  |A  | - |
                                     C   CC   |C  |CC |  +-----------+
                                              +-------+  "-" = null
```

Figure 572, EXISTS sub-query, always returns a match

Below is an EXISTS check that, given our sample data, always returns false:

```
SELECT *                                                       ANSWER
FROM   table1                                                  ======
WHERE  EXISTS                                                  0 rows
      (SELECT *
       FROM   table2
       WHERE  t2b >= 'X');
```

Figure 573, EXISTS sub-query, always returns a non-match

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 sub-query that will always return true, because even when no matching rows are found in the sub-query, the SELECT COUNT(*) statement will return something (i.e. a zero). Arguably, this query is logically flawed:

```
SELECT *                                      ANSWER   TABLE1      TABLE2
FROM   table1                                 =======  +-------+   +-----------+
WHERE  EXISTS                                 T1A T1B  |T1A|T1B|   |T2A|T2B|T2C|
      (SELECT COUNT(*)                        --- --   |---|---|   |---|---|---|
       FROM   table2                          A   aa   |A  |AA |   |A  |A  |A  |
       WHERE  t2b = 'X');                     B   BB   |B  |BB |   |B  |A  | - |
                                              C   CC   |C  |CC |   +-----------+
                                                       +-------+   "-" = null
```

Figure 574, EXISTS sub-query, always returns a match

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 *                                      ANSWERS  TABLE1      TABLE2
FROM   table1                                 =======  +-------+   +-----------+
WHERE  NOT EXISTS                             T1A T1B  |T1A|T1B|   |T2A|T2B|T2C|
      (SELECT *                               --- ---  |---|---|   |---|---|---|
       FROM   table2                          A   aa   |A  |AA |   |A  |A  |A  |
       WHERE  t2c >= 'A'                               |B  |BB |   |B  |A  | - |
         AND  t2c <> t1a);                             |C  |CC |   +-----------+
                                                       +-------+   "-" = null

SELECT *
FROM   table1
WHERE  t1a = ALL
      (SELECT t2c
       FROM   table2
       WHERE  t2c >= 'A');
```

Figure 575, NOT EXISTS vs. ALL, ignore nulls, find match

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:

```
SELECT *                                      ANSWERS  TABLE1      TABLE2
FROM   table1                                 =======  +-------+   +-----------+
WHERE  NOT EXISTS                             T1A T1B  |T1A|T1B|   |T2A|T2B|T2C|
      (SELECT *                               --- ---  |---|---|   |---|---|---|
       FROM   table2                          A   aa   |A  |AA |   |A  |A  |A  |
       WHERE  t2c >= 'X'                      B   BB   |B  |BB |   |B  |A  | - |
         AND  t2c <> t1a);                    C   CC   |C  |CC |   +-----------+
                                                       +-------+   "-" = null

SELECT *
FROM   table1
WHERE  t1a = ALL
      (SELECT t2c
       FROM   table2
       WHERE  t2c >= 'X');
```

Figure 576, NOT EXISTS vs. ALL, ignore nulls, no match

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:

```
SELECT *                                     ANSWER   TABLE1     TABLE2
FROM   table1                                =======  +-------+  +-----------+
WHERE  NOT EXISTS                            T1A T1B  |T1A|T1B|  |T2A|T2B|T2C|
      (SELECT *                              --- ---  |---|---|  |---|---|---|
       FROM   table2                         A   aa   |A  |AA |  |A  |A  |A  |
       WHERE  t2c <> t1a);                            |B  |BB |  |B  |A  | - |
                                                      |C  |CC |  +-----------+
                                                      +-------+  "-" = null
SELECT *                                     ANSWER
FROM   table1                                =======
WHERE  t1a = ALL                             no rows
      (SELECT t2c
       FROM   table2);
```

Figure 577, NOT EXISTS vs. ALL, process nulls

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:

```
SELECT *                    SELECT *                    SELECT *
FROM   table2               FROM   table2               FROM   table2
WHERE  t2c <> 'A';          WHERE  t2c <> 'B';          WHERE  t2c <> 'C';

ANSWER                      ANSWER                      ANSWER
===========                 ===========                 ===========
T2A T2B T2C                 T2A T2B T2C                 T2A T2B T2C
--- --- ---                 --- --- ---                 --- --- ---
no rows                     A   A   A                   A   A   A
```

Figure 578, List of values in T2C <> T1A value

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:

```
SELECT *                                     ANSWER   TABLE1     TABLE2
FROM   table1                                =======  +-------+  +-----------+
WHERE  NOT EXISTS                            no rows  |T1A|T1B|  |T2A|T2B|T2C|
      (SELECT *                                       |---|---|  |---|---|---|
       FROM   table2                                  |A  |AA |  |A  |A  |A  |
       WHERE  t2c <> t1a                              |B  |BB |  |B  |A  | - |
          OR  t2c IS NULL);                           |C  |CC |  +-----------+
                                                      +-------+  "-" = null
```

Figure 579, NOT EXISTS - same as ALL

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.

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:

```
SELECT *                                      ANSWER    TABLE1     TABLE2
FROM   table1                                 =======   +-------+  +-----------+
WHERE  t1a IN                                 T1A T1B   |T1A|T1B|  |T2A|T2B|T2C|
      (SELECT t2a                             --- --    |---|---|  |---|---|---|
       FROM   table2);                        A   aa    |A  |AA |  |A  |A  |A  |
                                              B   BB    |B  |BB |  |B  |A  | - |
                                                        |C  |CC |  +-----------+
                                                        +-------+  "-" = null
```

Figure 580, IN sub-query example, two matches

In the next example, no rows match because the sub-query result is an empty set:

```
SELECT *                                                       ANSWER
FROM   table1                                                  ======
WHERE  t1a IN                                                  0 rows
      (SELECT t2a
       FROM   table2
       WHERE  t2a >= 'X');
```

Figure 581, IN sub-query example, no matches

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:

```
SELECT *                                           ANSWERS       TABLE2
FROM   table2                                      ===========   +-----------+
WHERE  t2c IN                                      T2A T2B T2C   |T2A|T2B|T2C|
      (SELECT t2c                                  --- --- ---   |---|---|---|
       FROM   table2);                             A   A   A     |A  |A  |A  |
                                                                 |B  |A  | - |
SELECT *                                                         +-----------+
FROM   table2                                                    "-" = null
WHERE  t2c = ANY
      (SELECT t2c
       FROM   table2);
```

Figure 582, IN and = ANY sub-query examples, with nulls

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:

```
SELECT *                                       ANSWER  TABLE1     TABLE2
FROM   table1                                  ======  +-------+  +-----------+
WHERE  t1a NOT IN                              0 rows  |T1A|T1B|  |T2A|T2B|T2C|
      (SELECT t2c                                      |---|---|  |---|---|---|
       FROM   table2);                                 |A  |AA |  |A  |A  |A  |
                                                       |B  |BB |  |B  |A  | - |
                                                       |C  |CC |  +-----------+
                                                       +-------+  "-" = null
```

Figure 583, NOT IN sub-query example, no matches

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:

```
SELECT *                                ANSWER   TABLE1     TABLE2
FROM   table1                           =======  +-------+  +-----------+
WHERE  t1a NOT IN                       T1A T1B  |T1A|T1B|  |T2A|T2B|T2C|
      (SELECT t2c                       --- --   |---|---|  |---|---|---|
       FROM   table2                    B   BB   |A  |AA |  |A  |A  |A  |
       WHERE  t2c IS NOT NULL);         C   CC   |B  |BB |  |B  |A  | - |
                                                 |C  |CC |  +-----------+
                                                 +-------+  "-" = null
```

Figure 584, NOT IN sub-query example, matches

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:

```
SELECT *                                ANSWER   TABLE1     TABLE2
FROM   table1                           =======  +-------+  +-----------+
WHERE  NOT EXISTS                       T1A T1B  |T1A|T1B|  |T2A|T2B|T2C|
      (SELECT *                         --- --   |---|---|  |---|---|---|
       FROM   table2                    B   BB   |A  |AA |  |A  |A  |A  |
       WHERE  t1a = t2c);               C   CC   |B  |BB |  |B  |A  | - |
                                                 |C  |CC |  +-----------+
                                                 +-------+  "-" = null
```

Figure 585, NOT EXISTS sub-query example, matches

Correlated vs. Uncorrelated Sub-Queries

With the exception of the very last example above, all of the sub-queries shown so far have been uncorrelated. 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:

```
SELECT *                                ANSWER   TABLE1     TABLE2
FROM   table1                           =======  +-------+  +-----------+
WHERE  t1a IN                           T1A T1B  |T1A|T1B|  |T2A|T2B|T2C|
      (SELECT t2a                       --- --   |---|---|  |---|---|---|
       FROM   table2);                  A   aa   |A  |AA |  |A  |A  |A  |
                                        B   BB   |B  |BB |  |B  |A  | - |
                                                 |C  |CC |  +-----------+
                                                 +-------+  "-" = null
```

Figure 586, Uncorrelated sub-query

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:

```
SELECT *                                ANSWER   TABLE1     TABLE2
FROM   table1                           =======  +-------+  +-----------+
WHERE  t1a IN                           T1A T1B  |T1A|T1B|  |T2A|T2B|T2C|
      (SELECT t2a                       --- --   |---|---|  |---|---|---|
       FROM   table2                    A   aa   |A  |AA |  |A  |A  |A  |
       WHERE  t1a = t2a);               B   BB   |B  |BB |  |B  |A  | - |
                                                 |C  |CC |  +-----------+
                                                 +-------+  "-" = null
```

Figure 587, Correlated sub-query

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:

```
SELECT *                                        ANSWER       TABLE2
FROM   table2 aa                                ===========  +-----------+
WHERE  EXISTS                                   T2A T2B T2C  |T2A|T2B|T2C|
      (SELECT *                                 --- --- ---  |---|---|---|
       FROM   table2 bb                         A   A   A    |A  |A  |A  |
       WHERE  aa.t2a = bb.t2b);                              |B  |A  | - |
                                                             +-----------+
                                                             "-" = null
```

Figure 588,Correlated sub-query, with correlation names

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.

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:

```
SELECT *                                  ANSWER  TABLE1     TABLE2
FROM   table1                             ======  +-------+  +-----------+
WHERE (t1a,t1b) IN                        0 rows  |T1A|T1B|  |T2A|T2B|T2C|
      (SELECT t2a, t2b                            |---|---|  |---|---|---|
       FROM   table2);                            |A  |AA |  |A  |A  |A  |
                                                  |B  |BB |  |B  |A  | - |
                                                  |C  |CC |  +-----------+
                                                  +-------+  "-" = null
SELECT *                              ANSWER
FROM   table1                         ======
WHERE  EXISTS                         0 rows
      (SELECT   *
       FROM     table2
       WHERE    t1a = t2a
         AND    t1b = t2b);
```

Figure 589, Multi-field sub-queries, equal checks

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:

```
SELECT *                                 ANSWER   TABLE1     TABLE2
FROM   table1                            =======  +-------+  +-----------+
WHERE  EXISTS                            T1A T1B  |T1A|T1B|  |T2A|T2B|T2C|
      (SELECT   *                        --- --   |---|---|  |---|---|---|
       FROM     table2                   A   aa   |A  |AA |  |A  |A  |A  |
       WHERE    t1a  = t2a               B   BB   |B  |BB |  |B  |A  | - |
         AND    t1b >= t2b);                      |C  |CC |  +-----------+
                                                  +-------+  "-" = null
```

Figure 590, Multi-field sub-query, with non-equal check

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'.

```
SELECT empno                                              ANSWER
      ,lastname                                           =========================
      ,salary                                             EMPNO  LASTNAME  SALARY
FROM   employee                                           ------ --------- --------
WHERE  salary >                                           000010 HAAS      52750.00
      (SELECT MAX(salary)                                 000110 LUCCHESSI 46500.00
       FROM   employee
       WHERE  empno NOT IN
             (SELECT empno
              FROM   emp_act
              WHERE  projno LIKE 'MA%'))
ORDER BY 1;
```

Figure 591, Nested Sub-Queries

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.

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:

```
SELECT   COUNT(*)     AS #rows                                ANSWER
        ,MAX(deptno)  AS maxdpt                               =============
FROM     department                                           #ROWS MAXDEPT
WHERE    deptname LIKE 'Z%'                                   ----- -------
ORDER BY 1;                                                       0   null
```

Figure 592, Getting a null value from a not null field

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:

```
SELECT *                                             TABLE1     TABLE2
FROM   table1 t1                                     +-------+  +-----------+
WHERE  0 =                                           |T1A|T1B|  |T2A|T2B|T2C|
      (SELECT COUNT(*)                               |---|---|  |---|---|---|
       FROM   table2 t2                              |A  |AA |  |A  |A  |A  |
       WHERE  t1.t1a = t2.t2c);                      |B  |BB |  |B  |A  | - |
                                                     |C  |CC |  +-----------+
SELECT *                                             +-------+  "-" = null
FROM   table1 t1
WHERE  NOT EXISTS
      (SELECT *                                                       ANSWER
       FROM   table2 t2                                               =======
       WHERE  t1.t1a = t2.t2c);                                       T1A T1B
                                                                      --- ---
SELECT *                                                              B   BB
FROM   table1                                                         C   CC
WHERE  t1a NOT IN
      (SELECT t2c
       FROM   table2
       WHERE  t2c IS NOT NULL);
```

Figure 593, Sub-queries, true if none match

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.

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:

```
SELECT t1.*                                                   ANSWER
FROM   table1 t1                                              =======
LEFT OUTER JOIN                                               T1A T1B
       table2 t2                                              --- ---
ON     t1.t1a  = t2.t2c                                       B   BB
WHERE  t2.t2c IS NULL;                                        C   CC
```

Figure 594, Outer join, true if none match

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:

```
SELECT *                                     TABLE1     TABLE2
FROM   table1 t1                             +-------+  +-----------+
WHERE  EXISTS                                |T1A|T1B|  |T2A|T2B|T2C|
      (SELECT *                              |---|---|  |---|---|---|
       FROM   table2 t2                      |A  |AA |  |A  |A  |A  |
       WHERE  t1.t1a = t2.t2c);              |B  |BB |  |B  |A  | - |
                                             |C  |CC |  +-----------+
SELECT *                                     +-------+  "-" = null
FROM   table1 t1
WHERE  1 <=
      (SELECT COUNT(*)                                        ANSWER
       FROM   table2 t2                                       =======
       WHERE  t1.t1a = t2.t2c);                               T1A T1B
                                                              --- ---
SELECT *                                                      A   aa
FROM   table1
WHERE  t1a = ANY
      (SELECT t2c
       FROM   table2);

SELECT *
FROM   table1
WHERE  t1a = SOME
      (SELECT t2c
       FROM   table2);

SELECT *
FROM   table1
WHERE  t1a IN
      (SELECT t2c
       FROM   table2);
```

Figure 595, Sub-queries, true if any match

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.

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:

```
WITH t2 AS                                         TABLE1     TABLE2
(SELECT DISTINCT t2c                               +-------+  +-----------+
 FROM   table2                                     |T1A|T1B|  |T2A|T2B|T2C|
)                                                  |---|---|  |---|---|---|
SELECT t1.*                                        |A  |AA |  |A  |A  |A  |
FROM   table1 t1                                   |B  |BB |  |B  |A  | - |
      ,t2                                          |C  |CC |  +-----------+
WHERE  t1.t1a = t2.t2c;                            +-------+  "-" = null

SELECT t1.*
FROM   table1 t1                                                    ANSWER
      ,(SELECT DISTINCT t2c                                         =======
        FROM   table2                                               T1A T1B
       )AS t2                                                       --- ---
WHERE   t1.t1a = t2.t2c;                                            A   aa

SELECT t1.*
FROM   table1 t1
INNER JOIN
       (SELECT   DISTINCT t2c
        FROM     table2
       )AS t2
ON      t1.t1a = t2.t2c;
```

Figure 596, Joins, true if any match

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:

```
SELECT *                                           TABLE1     TABLE2
FROM   table1 t1                                   +-------+  +-----------+
WHERE  10 =                                        |T1A|T1B|  |T2A|T2B|T2C|
      (SELECT   COUNT(*)                           |---|---|  |---|---|---|
       FROM     table2 t2                          |A  |AA |  |A  |A  |A  |
       WHERE    t1.t1a = t2.t2b);                  |B  |BB |  |B  |A  | - |
                                                   |C  |CC |  +-----------+
SELECT *                                           +-------+  "-" = null
FROM   table1
WHERE  EXISTS
      (SELECT   t2b                                                  ANSWER
       FROM     table2                                               ======
       WHERE    t1a = t2b                                            0 rows
       GROUP BY t2b
       HAVING   COUNT(*) = 10);

SELECT *
FROM   table1
WHERE  t1a IN
      (SELECT   t2b
       FROM     table2
       GROUP BY t2b
       HAVING   COUNT(*) = 10);
```

Figure 597, Sub-queries, true if ten match (1 of 2)

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 multi-field IN (see page 207 for more notes) to both check T2B and the count at the same time:

```
SELECT *                                                    ANSWER
FROM   table1                                               ======
WHERE (t1a,10) IN                                           0 rows
      (SELECT   t2b, COUNT(*)
       FROM     table2
       GROUP BY t2b);
```

Figure 598, Sub-queries, true if ten match (2 of 2)

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:

```
WITH t2 AS                                      TABLE1      TABLE2
  (SELECT   t2b                                 +-------+   +-----------+
   FROM     table2                              |T1A|T1B|   |T2A|T2B|T2C|
   GROUP BY t2b                                 |---|---|   |---|---|---|
   HAVING   COUNT(*) = 10                       |A  |AA |   |A  |A  |A  |
  )                                             |B  |BB |   |B  |A  | - |
SELECT t1.*                                     |C  |CC |   +-----------+
FROM   table1 t1                                +-------+   "-" = null
      ,t2
WHERE  t1.t1a = t2.t2b;

                                                            ANSWER
SELECT t1.*                                                 ======
FROM   table1 t1                                            0 rows
      ,(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;
```

Figure 599, Joins, true if ten match

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 page 201 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:

```
SELECT *                                          TABLE1     TABLE2
FROM   table1                                     +-------+  +-----------+
WHERE  t1a = ALL                                  |T1A|T1B|  |T2A|T2B|T2C|
      (SELECT t2b                                 |---|---|  |---|---|---|
       FROM   table2);                            |A  |AA |  |A  |A  |A  |
                                                  |B  |BB |  |B  |A  | - |
SELECT *                                          |C  |CC |  +-----------+
FROM   table1                                     +-------+  "-" = null
WHERE  NOT EXISTS
      (SELECT *                                                    ANSWER
       FROM   table2                                               =======
       WHERE  t1a <> t2b);                                         T1A T1B
                                                                   --- ---
                                                                   A   aa
```

Figure 600, Sub-queries, true if all match, find rows

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:

```
SELECT *                                                           ANSWER
FROM   table1                                                      =======
WHERE  t1a = ALL                                                   T1A T1B
      (SELECT t2b                                                  --- ---
       FROM   table2                                               A   aa
       WHERE  t2b >= 'X');                                         B   BB
                                                                   C   CC
SELECT *
FROM   table1
WHERE  NOT EXISTS
      (SELECT *
       FROM   table2
       WHERE  t1a <> t2b
         AND  t2b >= 'X');
```

Figure 601, Sub-queries, true if all match, empty set

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:

```
SELECT *                                          TABLE1     TABLE2
FROM   table1                                     +-------+  +-----------+
WHERE  t1a = ALL                                  |T1A|T1B|  |T2A|T2B|T2C|
      (SELECT t2b                                 |---|---|  |---|---|---|
       FROM   table2                              |A  |AA |  |A  |A  |A  |
       WHERE  t2b >= 'X')                         |B  |BB |  |B  |A  | - |
  AND  0 <>                                       |C  |CC |  +-----------+
      (SELECT COUNT(*)                            +-------+  "-" = null
       FROM   table2
       WHERE  t2b >= 'X');                                          ANSWER
                                                                    ======
SELECT *                                                            0 rows
FROM   table1
WHERE  t1a IN
      (SELECT MAX(t2b)
       FROM   table2
       WHERE  t2b >= 'X'
       HAVING COUNT(DISTINCT t2b) = 1);
```

Figure 602, Sub-queries, true if all match, and at least one value found

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.

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:

```
                R1       R1        R1           R1           R1        R1
                UNION    UNION     INTERSECT    INTERSECT    EXCEPT    EXCEPT
                R2       ALL       R2           ALL          R2        ALL
R1   R2                  R2                     R2                     R2
--   --         -----    -----     ---------    -----        ------    ------
A    A          A        A         A            A            E         A
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
```

Figure 603, Examples of Union, Except, and Intersect

WARNING: 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".

Syntax Diagram

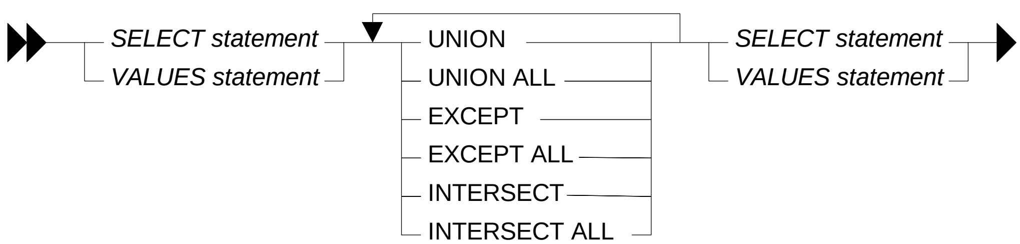

Figure 604, Union, Except, and Intersect syntax

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');           ANSWER
                                                                  ======
SELECT   R1                                                       R1   R2
FROM     R1                                                       --   --
ORDER BY R1;                                                      A    A
                                                                  A    A
SELECT   R2                                                       A    B
FROM     R2                                                       B    B
ORDER BY R2;                                                      B    B
                                                                  C    C
                                                                  C    D
                                                                  C
                                                                  E
```

Figure 605, Query sample views

Usage Notes

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.

```
SELECT   R1                           R1  R2   UNION  UNION ALL
FROM     R1                           --  --   =====  =========
UNION                                 A   A    A      A
SELECT   R2                           A   A    B      A
FROM     R2                           A   B    C      A
ORDER BY 1;                           B   B    D      A
                                      B   B    E      A
                                      C   C           B
SELECT   R1                           C   D           B
FROM     R1                           C               B
UNION ALL                             E               B
SELECT   R2                                           B
FROM     R2                                           C
ORDER BY 1;                                           C
                                                      C
                                                      C
                                                      D
                                                      E
```

Figure 606, Union and Union All SQL

NOTE: 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.

```
SELECT   R1                   R1  R2   INTERSECT  INTERSECT ALL
FROM     R1                   --  --   =========  =============
INTERSECT                     A   A    A          A
SELECT   R2                   A   A    B          A
FROM     R2                   A   B    C          B
ORDER BY 1;                   B   B               B
                              B   B               C
SELECT   R1                   C   C
FROM     R1                   C   D
INTERSECT ALL                 C
SELECT   R2                   E
FROM     R2
ORDER BY 1;
```

Figure 607, Intersect and Intersect All SQL

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.

Except & Except All

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.

```
SELECT   R1                                               R1      R1
FROM     R1                                               EXCEPT  EXCEPT ALL
EXCEPT                                           R1  R2   R2      R2
SELECT   R2                                      --  --   =====   ==========
FROM     R2                                      A   A    E       A
ORDER BY 1;                                      A   A            C
                                                 A   B            C
SELECT   R1                                      B   B            E
FROM     R1                                      B   B
EXCEPT  ALL                                      C   C
SELECT   R2                                      C   D
FROM     R2                                      C
ORDER BY 1;                                      E
```

Figure 608, Except and Except All SQL (R1 on top)

Because the EXCEPT 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:

```
SELECT   R2                                               R2      R2
FROM     R2                                               EXCEPT  EXCEPT ALL
EXCEPT                                           R1  R2   R1      R1
SELECT   R1                                      --  --   =====   ==========
FROM     R1                                      A   A    D       B
ORDER BY 1;                                      A   A            D
                                                 A   B
SELECT   R2                                      B   B
FROM     R2                                      B   B
EXCEPT  ALL                                      C   C
SELECT   R1                                      C   D
FROM     R1                                      C
ORDER BY 1;                                      E
```

Figure 609, Except and Except All SQL (R2 on top)

NOTE: Only the EXCEPT operation is not commutative. Both the UNION and the INTERSECT operations work the same regardless of which table is on top or on bottom.

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       (SELECT   R1        SELECT   R1              R1  R2
FROM     R1        FROM     R1        FROM     R1              --  --
UNION              UNION              UNION                    A   A
SELECT   R2        SELECT   R2       (SELECT   R2              A   A
FROM     R2        FROM     R2        FROM     R2              A   B
EXCEPT            )EXCEPT             EXCEPT                   B   B
SELECT   R2        SELECT   R2        SELECT   R2              B   B
FROM     R2        FROM     R2        FROM     R2              C   C
ORDER BY 1;        ORDER BY 1;       )ORDER BY 1;              C   D
                                                               C
                                                               E
ANSWER             ANSWER             ANSWER
======             ======             ======
E                  E                  A
                                      B
                                      C
                                      E
```

Figure 610, Use of parenthesis in Union

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:

```
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;
```

Figure 611, Define view to combine yearly tables

Below is some SQL that changes the contents of the above 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;
```

Figure 612, Insert, update, and delete using view

Below is the view contents, after the above is run:

```
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
```

Figure 613, View contents after insert, update, delete

Materialized Query Tables

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, or tables, to use instead the materialized query table instead. This can make the query run much faster.

The following statement defines a materialized query table:

```
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;
```

Figure 614, Sample materialized query table DDL

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
==============
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
```

Figure 615, Original and optimized queries

When used appropriately, materialized query tables can result in 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.

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:

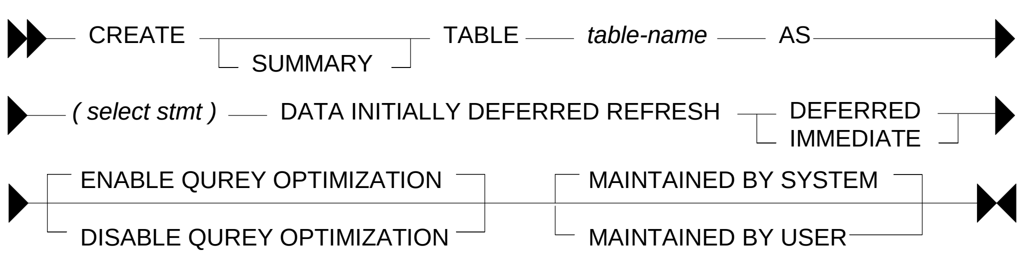

Figure 616, Materialized query table DDL, syntax diagram

Below is a typical materialized query table definition:

```
CREATE TABLE emp_summary AS
  (SELECT   workdept           AS dept
           ,sex                AS sex
           ,COUNT_BIG(*)       AS num_rows
           ,COUNT(salary)      AS num_salary
           ,SUM(salary)        AS sum_salary
           ,GROUPING(workdept) AS fd
           ,GROUPING(sex)      AS fs
   FROM     employee
   WHERE    job         = 'MANAGER'
     AND    lastname LIKE '%S%'
   GROUP BY CUBE(workdept, sex)
)DATA INITIALLY DEFERRED REFRESH IMMEDIATE
ENABLE QUERY OPTIMIZATION
MAINTAINED BY SYSTEM;
```

Figure 617, Typical materialized query table definition

Refresh Options

- 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 the most useful in that the data in them is always current. But they may cost quite a bit to maintain.

Query Optimization Options

- 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.

Maintain Options

- 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.

Options vs. Actions

The following table compares materialized query table options to subsequent 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
```

Figure 618, Materialized query table options vs. allowable actions

Select Statement Restrictions

Various restrictions apply to the select statement used to define the materialized query table:

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).
- The SELECT statement must contain a GROUP BY, unless REPLICATED is specified, in which case a GROUP BY is not allowed.
- The SELECT 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.

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:

```
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;
```

Figure 619, Refresh deferred materialized query table DDL

Using a Refreshed Deferred Table

Unless told otherwise, the DB2 optimizer will not use a materialized query table that is defined refresh deferred, because it cannot guarantee that the data in the table is up to date. If it is desired that such a table be referenced when appropriate, one has to set the REFRESH AGE special register to a non-zero value:

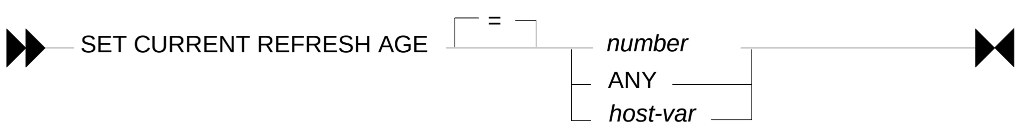

Figure 620, Refresh age command, syntax

The number referred to above is a 26-digit decimal value that is as a timestamp duration, but without the microsecond component. Only two values are allowed:

- 0: Only use those materialized query tables defined refresh immediate.
- 99,999,999,999,999: Use all valid materialized query tables (same as ANY).

Below is the SET command in action:

```
SET CURRENT REFRESH AGE   0;
SET CURRENT REFRESH AGE = ANY;
SET CURRENT REFRESH AGE = 99999999999999;
```

Figure 621, Set refresh age command

One can select the CURRENT REFRESH AGE special register to see what the value is:

```
SELECT  CURRENT REFRESH AGE  AS age_ts
       ,CURRENT TIMESTAMP    AS current_ts
FROM    sysibm.sysdummy1;
```

Figure 622, Selecting refresh age

One can also query the DB2 catalogue to get list of all materialized query tables, and what their refresh option is:

```
SELECT   CHAR(tabschema,10)  AS schema
        ,CHAR(tabname,20)    AS table
        ,type
        ,refresh
        ,refresh_time
        ,card                AS #rows
        ,DATE(create_time)   AS create_dt
        ,DATE(stats_time)    AS stats_dt
FROM     syscat.tables
WHERE    type = 'S'
ORDER BY 1,2;
```

Figure 623, List all materialized query tables

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:

```
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;
```

Figure 624, Refresh immediate materialized query table DDL

Below is a query that can use the above materialized query table in place of the base table:

```
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;
```

Figure 625, Query that uses materialized query table (1 of 3)

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:

```
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;
```

Figure 626, Query that uses materialized query table (2 of 3)

This last example uses the materialized query table in a nested table expression:

```
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;
```

Figure 627, Query that uses materialized query table (3 of 3)

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:

```
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;
```

Figure 628, Query that doesn't use materialized query table (1 of 2)

The following query can't use the materialized query table because of the DISTINCT clause:

```
SELECT   emp.workdept
        ,DEC(SUM(emp.salary),8,2)   AS sum_sal
        ,COUNT(DISTINCT salary)     AS #salaries
FROM     employee emp
GROUP BY emp.workdept;
```

Figure 629, Query that doesn't use materialized query table (2 of 2)

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:

```
REFRESH TABLE emp_summary;
COMMIT;

SET INTEGRITY FOR emp_summary iMMEDIATE CHECKED;
COMMIT;
```

Figure 630, Materialized query table refresh commands

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.

```
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;
```

Figure 631, Multi-table materialized query table DDL

The following query is resolved using the above 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;
```

Figure 632, Query that uses materialized query table

Here is the SQL that DB2 generated internally to get the answer:

```
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;
```

Figure 633, DB2 generated query to use materialized query table

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.

Three-table Example

```
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;
```

Figure 634, Three-table materialized query table DDL

Now for a query that will use the above:

```
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;
```

Figure 635, Query that uses materialized query table

And here is the DB2 generated SQL:

```
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;
```

Figure 636, DB2 generated query to use materialized query table

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:

```
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);
```

Figure 637, Indexes for DPT_EMP_ACT_SUMRY materialized query table table

The next query will use the first index (i.e. on WORKDEPT):

```
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;
```

Figure 638, Sample query that use WORKDEPT index

The next query will use the second index (i.e. on NUM_ROWS):

```
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;
```

Figure 639, Sample query that uses NUM_ROWS index

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:

```
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;
```
Figure 640, Materialized query table organized by dimensions

> WARNING: 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!

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.

NOTE: 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;
```
Figure 641, Sample materialized query table

Here is a staging table for the above:

```
CREATE TABLE emp_sumry_s
  (dept
  ,num_rows
  ,num_sal
  ,sum_sal
  ,GLOBALTRANSID
  ,GLOBALTRANSTIME
)FOR emp_sumry PROPAGATE IMMEDIATE;
```
Figure 642, Staging table for the above materialized query table

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.

```
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;
```
Figure 643, Enabling and the using a staging table

- 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.

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 is almost never a valid business reason for requiring such an unbroken sequence of values. Regardless, some people want this feature, and it can, up to a point, be implemented in DB2. In this chapter we will describe how to do it.

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 a syntax diagram for that part of a CREATE TABLE statement that refers to an identity column definition:

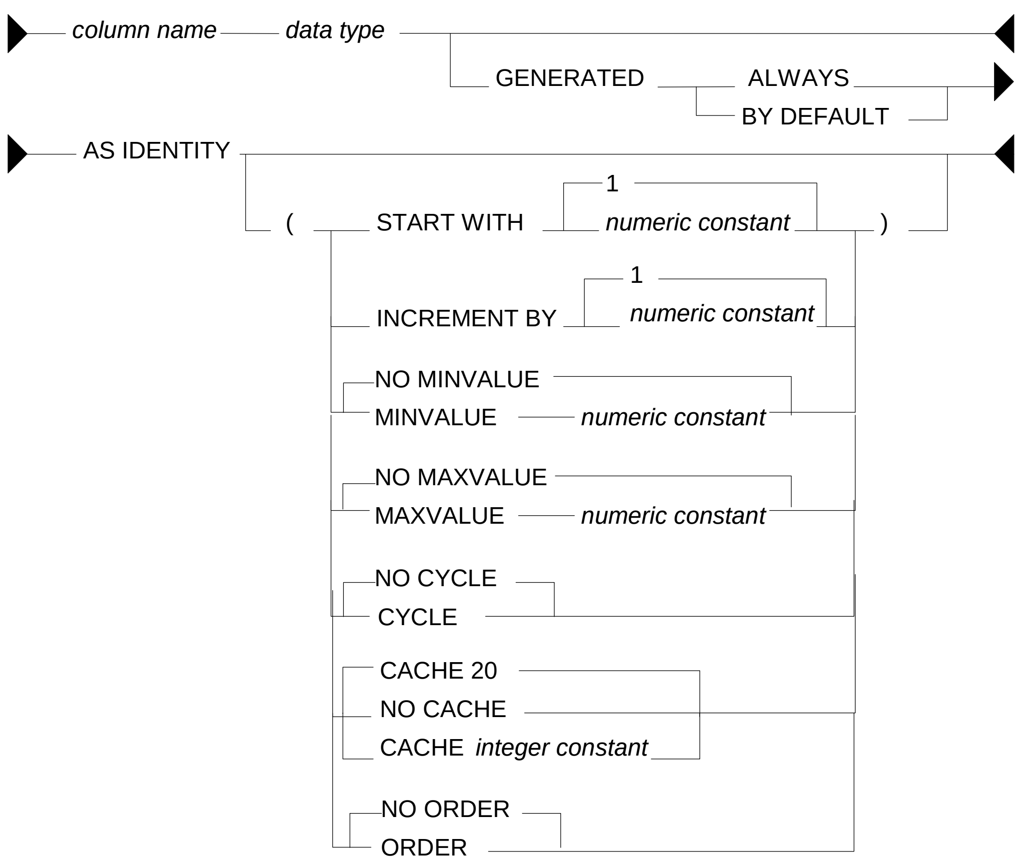

Figure 644, Identity Column syntax

Below is an example of a typical invoice table that uses an identity column that starts at one, and then goes ever upwards:

```
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#));
```

Figure 645, Identity column, sample table

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).

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.

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.

Sequence Examples

The following example uses all of the defaults to start a sequence at one, and then to go up in increments of one. The inserts will finally die when they reach the maximum allowed value for the field type (i.e. for small integer = 32K).

```
CREATE TABLE TEST_DATA                       KEY# FIELD - VALUES ASSIGNED
(KEY#  SMALLINT  NOT NULL                    ============================
       GENERATED ALWAYS AS IDENTITY          1 2 3 4 5 6 7 8 9 10 11 etc.
,DAT1  SMALLINT  NOT NULL
,TS1   TIMESTAMP NOT NULL
,PRIMARY KEY(KEY#));
```

Figure 646, Identity column, ascending sequence

The next example defines a sequence that goes down in increments of -3:

```
CREATE TABLE TEST_DATA                       KEY# FIELD - VALUES ASSIGNED
(KEY#  SMALLINT  NOT NULL                    ============================
       GENERATED ALWAYS AS IDENTITY          6 3 0 -3 -6 -9 -12 -15 etc.
                (START WITH    6
                ,INCREMENT BY -3
                ,NO CYCLE
                ,NO CACHE
                ,ORDER)
,DAT1  SMALLINT  NOT NULL
,TS1   TIMESTAMP NOT NULL
,PRIMARY KEY(KEY#));
```

Figure 647, Identity column, descending sequence

The next example, which is amazingly stupid, goes nowhere fast. A primary key cannot be defined on this table:

```
CREATE TABLE TEST_DATA                        KEY# VALUES ASSIGNED
(KEY#  SMALLINT  NOT NULL                     ============================
       GENERATED ALWAYS AS IDENTITY           123 123 123 123 123 123 etc.
                 (START WITH   123
                 ,MAXVALUE     124
                 ,INCREMENT BY 0
                 ,NO CYCLE
                 ,NO ORDER)
,DAT1  SMALLINT  NOT NULL
,TS1   TIMESTAMP NOT NULL);
```

Figure 648, Identity column, dumb sequence

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:

```
CREATE TABLE TEST_DATA                        KEY# VALUES ASSIGNED
(KEY#  SMALLINT  NOT NULL                     ============================
       GENERATED ALWAYS AS IDENTITY           1 3 5 2 4 6 2 4 6 2 4 6 etc.
                 (START WITH    1
                 ,INCREMENT BY  2
                 ,MAXVALUE      6
                 ,MINVALUE      2
                 ,CYCLE
                 ,NO CACHE
                 ,ORDER)
,DAT1  SMALLINT  NOT NULL
,TS1   TIMESTAMP NOT NULL);
```

Figure 649, Identity column, odd values, then even, then stuck

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:

```
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#));
```

Figure 650, Identity column, definition

One cannot provide an input 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:

```
INSERT INTO INVOICE_DATA
VALUES (DEFAULT,'2001-11-22','ABC',123,100,10);

INSERT INTO INVOICE_DATA
(SALE_DATE,CUSTOMER_ID,PRODUCT_ID,QUANTITY,PRICE)
VALUES ('2001-11-23','DEF',123,100,10);
```

Figure 651, Invoice table, sample inserts

Below is the state of the table after the above two inserts:

```
INVOICE#   SALE_DATE    CUSTOMER_ID   PRODUCT_ID   QUANTITY   PRICE
--------   ----------   -----------   ----------   --------   -----
     100   11/22/2001   ABC                  123        100   10.00
     101   11/23/2001   DEF                  123        100   10.00
```

Figure 652, Invoice table, after inserts

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:

```
ALTER TABLE  INVOICE_DATA
ALTER COLUMN INVOICE#
   RESTART WITH 1000
   SET INCREMENT BY 2;
```

Figure 653, Invoice table, restart identity column value

Now imagine that we insert two more rows thus:

```
INSERT INTO INVOICE_DATA
VALUES (DEFAULT,'2001-11-24','XXX',123,100,10)
      ,(DEFAULT,'2001-11-25','YYY',123,100,10);
```

Figure 654, Invoice table, more sample inserts

Our mindless management will now see this data:

```
INVOICE#   SALE_DATE    CUSTOMER_ID   PRODUCT_ID   QUANTITY   PRICE
--------   ----------   -----------   ----------   --------   -----
     100   11/22/2001   ABC                  123        100   10.00
     101   11/23/2001   DEF                  123        100   10.00
    1000   11/24/2001   XXX                  123        100   10.00
    1002   11/25/2001   YYY                  123        100   10.00
```

Figure 655, Invoice table, after second inserts

Alter Usage Notes

As the following diagram shows, all of the identity column options can be changed using the ALTER TABLE command:

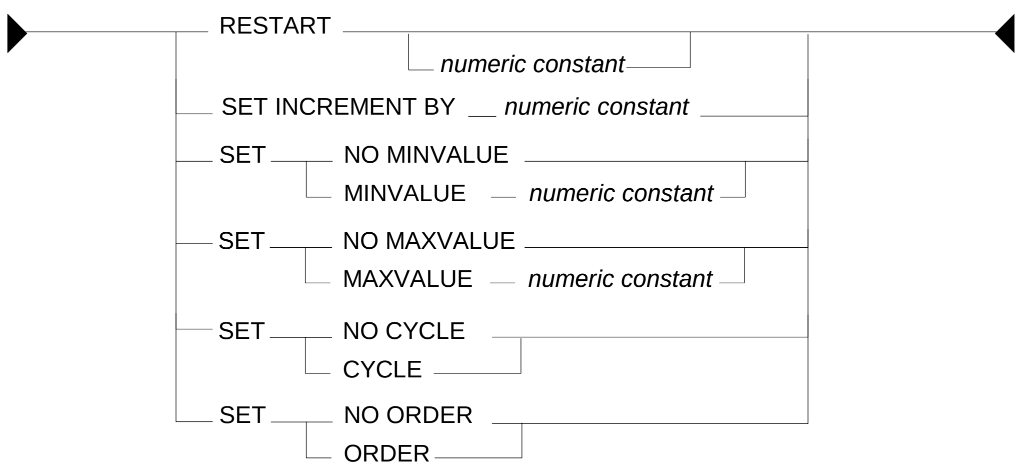

Figure 656, Identity Column alter syntax

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.

Gaps in the Sequence

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 being committed. Below is an illustration of this problem:

```
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;


INSERT INTO CUSTOMERS
VALUES (DEFAULT,'FRED','XXX');

SELECT   *                                     <<< ANSWER
FROM     CUSTOMERS                                 ==================
ORDER BY 1;                                        CUST#  CNAME  CTYPE
                                                   -----  -----  -----
ROLLBACK;                                              1  FRED   XXX


INSERT INTO CUSTOMERS
VALUES (DEFAULT,'FRED','XXX');

SELECT   *                                     <<< ANSWER
FROM     CUSTOMERS                                 ==================
ORDER BY 1;                                        CUST#  CNAME  CTYPE
                                                   -----  -----  -----
COMMIT;                                                2  FRED   XXX
```

Figure 657, Overriding the default identity value

One advantage of DB2's identity column implementation is that the value allocation process is not a point of contention in the table. Subsequent users do not have to wait for the first user to do a commit before they can insert their own rows.

Roll Your Own - no Gaps in Sequence

If one really, really, needs to have a sequence of values with no gaps, then one can do it using a trigger, but there are costs, in processing time, concurrency, and functionality. To illustrate how to do it, consider the following table:

```
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#));
```

Figure 658, Sample table, roll your own sequence#

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:

```
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);
```

Figure 659, Sample trigger, roll your own sequence#

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:

```
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
```

Figure 660, Sample inserts, roll your own sequence#

IDENTITY_VAL_LOCAL Function

Imagine that one has just inserted a row, and one now wants to find out what value DB2 gave the identity column. One calls the IDENTITY_VAL_LOCAL function to find out. The result is a decimal (31.0) field. Certain rules apply:

- 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.

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:

```
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                                            <<< ANSWER
(VALUES (IDENTITY_VAL_LOCAL()))                                  ======
SELECT *                                                             ID
FROM   TEMP;                                                       ----
                                                                      1
COMMIT;

WITH TEMP (ID) AS                                            <<< ANSWER
(VALUES (IDENTITY_VAL_LOCAL()))                                  ======
SELECT *                                                             ID
FROM   TEMP;                                                       ----
                                                                      1
```

Figure 661, IDENTITY_VAL_LOCAL function examples

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:

```
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);     ANSWER
                                                    ==================
SELECT   INVOICE#             AS INV#               INV# SALE_DATE  ID
        ,SALE_DATE                                  ---- ---------- --
        ,IDENTITY_VAL_LOCAL() AS ID                    1 11/22/2000  2
FROM     INVOICE_TABLE                                 2 11/23/2000  2
ORDER BY 1;                                            3 11/24/2000  2
COMMIT;                                                4 11/25/2000  2
```

Figure 662, IDENTITY_VAL_LOCAL function examples

One can also use the function to get the most recently inserted single row:

```
SELECT INVOICE#             AS INV#                 ANSWER
      ,SALE_DATE                                    ==================
      ,IDENTITY_VAL_LOCAL() AS ID                   INV# SALE_DATE  ID
FROM   INVOICE_TABLE                                ---- ---------- --
WHERE  ID = IDENTITY_VAL_LOCAL();                      2 11/23/2000  2
```

Figure 663, IDENTITY_VAL_LOCAL usage in predicate

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 FRED                              SEQ# VALUES ASSIGNED
   AS DECIMAL(31)                                 ====================
   START WITH 100                                 100 102 104 106 etc.
   INCREMENT BY 2
   NO MINVALUE
   NO MAXVALUE
   NO CYCLE
   CACHE 20
   ORDER;
```

Figure 664, Create sequence

The options and defaults for a sequence are exactly the same as those for an identity column (see page 230). Likewise, one can alter a sequence in much the same way as one would alter the status of an identity column:

```
ALTER SEQUENCE FRED                               SEQ# VALUES ASSIGNED
   RESTART WITH -55                               ====================
   INCREMENT BY -5                                -55 -60 -65 -70 etc.
   MINVALUE     -1000
   MAXVALUE     +1000
   NO CACHE
   NO ORDER
   CYCLE;
```

Figure 665, Alter sequence attributes

The only sequence attribute that one cannot change with the ALTER command is the field type that is used to hold the current value.

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:

```
CREATE SEQUENCE FRED;                                              ANSWER
COMMIT;                                                            ======
                                                                     SEQ#
WITH TEMP1 (N1) AS                                                   ----
(VALUES 1                                                               1
 UNION ALL                                                              2
 SELECT N1 + 1                                                          3
 FROM   TEMP1                                                           4
 WHERE  N1 < 5                                                          5
)
SELECT NEXTVAL FOR FRED AS SEQ#
FROM   TEMP1;
```
Figure 666, Selecting the NEXTVAL

Rules and Restrictions

- 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, and 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 prior will result in a SQL error.
- The NEXTVAL and PREVVAL can be used in the following statements:
- SELECT INTO statement (within the select clause), 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.
- The NEXTVAL can be used in a trigger, but the PREVVAL cannot.
- The NEXTVAL and PREVVAL cannot be used in the following statements:
- Join condition of a full outer join.
- Anywhere in a CREATE TABLE or CREATE VIEW statement.
- The NEXTVAL cannot be used in the following statements:
- 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.

There are many more usage restrictions, but you presumably get the picture. See the DB2 SQL Reference for the complete list.

Usage Examples

Below a sequence is defined, then various next and previous values are retrieved:

```
CREATE SEQUENCE FRED;                                          ANSWERS
COMMIT;                                                        =======

WITH TEMP1 (PRV) AS                              ===>              PRV
(VALUES (PREVVAL FOR FRED))                                        ---
SELECT *                                                       <error>
FROM   TEMP1;

WITH TEMP1 (NXT) AS                              ===>              NXT
(VALUES (NEXTVAL FOR FRED))                                        ---
SELECT *                                                             1
FROM   TEMP1;

WITH TEMP1 (PRV) AS                              ===>              PRV
(VALUES (PREVVAL FOR FRED))                                        ---
SELECT *                                                             1
FROM   TEMP1;

WITH TEMP1 (N1) AS                               ===>          NXT PRV
(VALUES 1                                                      --- ---
 UNION ALL                                                       2   1
 SELECT N1 + 1                                                   3   1
 FROM   TEMP1                                                    4   1
 WHERE  N1 < 5                                                   5   1
)                                                                6   1
SELECT NEXTVAL FOR FRED AS NXT
      ,PREVVAL FOR FRED AS PRV
FROM   TEMP1;
```
Figure 667, Use of NEXTVAL and PREVVAL expressions

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):

```
CREATE SEQUENCE FRED;                                          ANSWERS
COMMIT;                                                        =======

WITH TEMP1 AS                                    ===>           ID NXT
(SELECT ID                                                      -- ---
       ,NEXTVAL FOR FRED AS NXT                                 50   5
 FROM   STAFF
 WHERE  ID < 100
)
SELECT *
FROM   TEMP1
WHERE  ID = 50;

WITH TEMP1 (NXT, PRV) AS                         ===>          NXT PRV
(VALUES (NEXTVAL FOR FRED                                      --- ---
        ,PREVVAL FOR FRED))                                     10   9
SELECT *
FROM   TEMP1;
```
Figure 668, NEXTVAL values used but not retrieved

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 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#;
```

Figure 669, Create tables that use a common sequence

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 US_CUSTOMER (CNAME, FRST_SALE, #SALES)
VALUES ('FRED','2002-10-22',1)
      ,('JOHN','2002-10-23',1);

INSERT INTO INTL_CUSTOMER (CNAME, FRST_SALE, #SALES)
VALUES ('SUE','2002-11-12',2)
      ,('DEB','2002-11-13',2);
COMMIT;
                                                                ANSWERS
                                            =============================
SELECT   *                                  CUST# CNAME FRST_SALE  #SALES
FROM     US_CUSTOMER                        ----- ----- ---------- ------
ORDER BY CUST#                                  1 FRED  10/22/2002      1
                                                2 JOHN  10/23/2002      1


SELECT   *                                  CUST# CNAME FRST_SALE  #SALES
FROM     INTL_CUSTOMER                      ----- ----- ---------- ------
ORDER BY CUST#;                                 3 SUE   11/12/2002      2
                                                4 DEB   11/13/2002      2
```

Figure 670, Insert into tables with common sequence

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:

```
WITH TEMP (PREV) AS                                                ANSWER
(VALUES (PREVVAL FOR CUST#))                                       ======
SELECT *                                                             PREV
FROM   TEMP;                                                         ----
                                                                        4
```

Figure 671, Get previous value - select

The following does the same as the above, but puts the result in a host variable:

```
VALUES PREVVAL FOR CUST# INTO :host-var
```

Figure 672, Get previous value - into host-variable

Using the above, we cannot find out how many rows were inserted in the most recent insert, nor to which table the insert was done. And we cannot even be sure that the value is correct, because the insert may have been rolled back after the value was assigned.

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:

```
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;
```

Figure 673, Count deletes done to table

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.

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 columns are not supported in databases with multiple partitions.
- 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.

Now to compare the expressions that get the current status:

- 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.

Temporary Tables

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.

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:

```
SELECT   id
        ,salary
FROM    (SELECT   s.*
                 ,ROW_NUMBER() OVER(ORDER BY salary DESC) AS sorder
         FROM     staff s
         WHERE    id < 200                                       ANSWER
        )AS xxx                                                  =============
WHERE    sorder BETWEEN 2 AND 3                                  ID   SALARY
ORDER BY id;                                                     ---  --------
                                                                  50  20659.80
                                                                 140  21150.00
```

Figure 674, Nested Table Expression

NOTE: 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:

```
WITH xxx (id, salary, sorder) AS
(SELECT  ID
        ,salary
        ,ROW_NUMBER() OVER(ORDER BY salary DESC) AS sorder
 FROM    staff
 WHERE   id < 200
)                                                                ANSWER
SELECT   id                                                      =============
        ,salary                                                  ID   SALARY
FROM     xxx                                                     ---  --------
WHERE    sorder BETWEEN 2 AND 3                                   50  20659.80
ORDER BY id;                                                     140  21150.00
```

Figure 675, Common Table Expression

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:

```
WITH                                      ANSWER
rows_wanted AS                            ================================
   (SELECT  *                             ID NAME    SALARY   SUM_SAL  PCT
    FROM    staff                         -- ------- -------- -------- ---
    WHERE   id            < 100           70 Rothman 16502.83 34504.58  47
      AND   UCASE(name) LIKE '%T%'        90 Koonitz 18001.75 34504.58  52
    ),
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;
```

Figure 676, Common Table Expression

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:

```
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(*)                               ANSWER#1
FROM     staff                                  ========
WHERE    id > 200                               CNT
GROUP BY dept;                                  ---
COMMIT;                                           4

SELECT  COUNT(*) AS cnt
FROM    session.fred;                           ANSWER#2
                                                ==========================
DELETE FROM session.fred                        DEPT  AVG_SALARY  NUM_EMPS
WHERE  dept > 80;                               ----  ----------  --------
                                                  10    20168.08         3
SELECT  *                                         51    15161.43         3
FROM    session.fred;                             66    17215.24         5
```

Figure 677, Declared Global Temporary Table

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 page 251.

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 full-select in the FROM part of the query.
- Define a full-select 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:

```
WITH staff_dept AS                           ANSWER
(SELECT   dept         AS dept#              ==========================
         ,MAX(salary)  AS max_sal            ID  DEPT  SALARY   MAX_SAL
 FROM     staff                              --- ---- -------- --------
 WHERE    dept < 50                           10   20 18357.50 18357.50
 GROUP BY dept                               190   20 14252.75 18357.50
)                                            200   42 11508.60 18352.80
SELECT   id                                  220   51 17654.50        -
        ,dept
        ,salary
        ,max_sal
FROM     staff
LEFT OUTER JOIN
         staff_dept
ON       dept     = dept#
WHERE    name LIKE 'S%'
ORDER BY id;
```

Figure 678, Identical query (1 of 3) - using Common Table Expression

```
SELECT   id                                  ANSWER
        ,dept                                ==========================
        ,salary                              ID  DEPT  SALARY   MAX_SAL
        ,max_sal                             --- ---- -------- --------
FROM     staff                                10   20 18357.50 18357.50
LEFT OUTER JOIN                              190   20 14252.75 18357.50
        (SELECT   dept         AS dept#      200   42 11508.60 18352.80
                 ,MAX(salary)  AS max_sal    220   51 17654.50        -
         FROM     staff
         WHERE    dept < 50
         GROUP BY dept
        )AS STAFF_dept
ON       dept     = dept#
WHERE    name LIKE 'S%'
ORDER BY id;
```

Figure 679, Identical query (2 of 3) - using full-select in FROM

```
SELECT   id                                  ANSWER
        ,dept                                ==========================
        ,salary                              ID  DEPT  SALARY   MAX_SAL
        ,(SELECT   MAX(salary)               --- ---- -------- --------
          FROM     staff s2                   10   20 18357.50 18357.50
          WHERE    s1.dept = s2.dept         190   20 14252.75 18357.50
            AND    s2.dept < 50              200   42 11508.60 18352.80
          GROUP BY dept)                     220   51 17654.50        -
         AS max_sal
FROM     staff s1
WHERE    name LIKE 'S%'
ORDER BY id;
```

Figure 680, Identical query (3 of 3) - using full-select in SELECT

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.

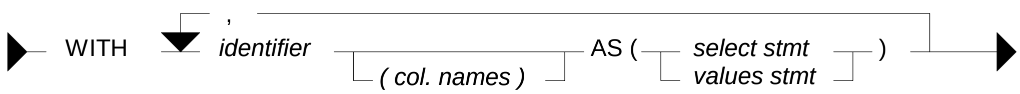

Figure 681, Common Table Expression Syntax

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.

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):

```
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
```

Figure 682, Common Table Expression, using named fields

In this next example, the fields being selected are unnamed, so names have to be specified in the WITH statement:

```
WITH temp1 (max_name,max_dept) AS
(SELECT MAX(name)
       ,MAX(dept)
 FROM   staff
)
SELECT *
FROM   temp1;
```

```
ANSWER
==================
MAX_NAME  MAX_DEPT
--------- --------
Yamaguchi       84
```

Figure 683, Common Table Expression, using unnamed fields

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:

```
WITH                                                        ANSWER
temp1 AS                                                    ==========
  (SELECT   dept                                            MAX_AVG
           ,AVG(salary)  AS avg_sal                         ----------
   FROM     staff                                           20865.8625
   GROUP BY dept),
temp2 AS
  (SELECT   MAX(avg_sal) AS max_avg
   FROM     temp1)
SELECT *
FROM   temp2;
```

Figure 684, Query with two common table expressions

FYI, the exact same query can be written using nested table expressions thus:

```
SELECT *                                                    ANSWER
FROM  (SELECT MAX(avg_sal) AS max_avg                       ==========
       FROM  (SELECT dept                                   MAX_AVG
                    ,AVG(salary) AS avg_sal                 ----------
              FROM   staff                                  20865.8625
              GROUP BY dept
             )AS temp1
      )AS temp2;
```

Figure 685, Same as prior example, but using nested table expressions

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:

```
WITH temp1 AS                                  ANSWER
(SELECT   id                                   ==========================
         ,name                                 ID  DEPT  SALARY   MAX_SAL
         ,dept                                 --- ---- -------- --------
         ,salary                                10   20 18357.50 18357.50
 FROM     staff                                190   20 14252.75 18357.50
 WHERE    id      <  300                       200   42 11508.60 11508.60
   AND    dept   <>  55                        220   51 17654.50 17654.50
   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;
```

Figure 686, Deriving second temporary table from first

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 INTO staff
WITH temp1 (max1) AS
(SELECT MAX(id) + 1
 FROM   staff
)
SELECT max1,'A',1,'B',2,3,4
FROM   temp1;
```

Figure 687, Insert using common table expression

As it happens, the above query can be written equally well in the raw:

```
INSERT INTO staff
SELECT MAX(id) + 1
      ,'A',1,'B',2,3,4
FROM   staff;
```

Figure 688, Equivalent insert (to above) without common table expression

Full-Select

A full-select 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 full-select 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 full-select 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 full-select is a column of data (i.e. in SELECT part of query), each reference to the temporary table must only return a single value.

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:

```
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                            ANSWER
                 ,org                   ==============================
         WHERE    dept = deptnumb       DIVISION  DIV_DEPT #DPTS #EMPS
         GROUP BY division              --------- -------- ----- -----
                 ,dept                  Corporate 20865.86     1     4
        )AS xxx                         Eastern   15670.32     3    13
GROUP BY division;                      Midwest   15905.21     2     9
                                        Western   16875.99     2     9
```

Figure 689, Nested column function usage

The next query illustrates how multiple full-selects can be nested inside each other:

```
SELECT id                                                          ANSWER
FROM  (SELECT *                                                    ======
       FROM  (SELECT id, years, salary                               ID
              FROM  (SELECT *                                        ---
                     FROM   (SELECT *                                170
                             FROM   staff                            180
                             WHERE  dept < 77                        230
                            )AS t1
                     WHERE  id  < 300
                    )AS t2
              WHERE  job LIKE 'C%'
             )AS t3
       WHERE  salary < 18000
      )AS t4
WHERE  years < 5;
```
Figure 690, Nested full-selects

A very common usage of a full-select 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:

```
SELECT   a.id                                    ANSWER
        ,a.dept                                  =========================
        ,a.salary                                ID DEPT  SALARY  AVG_DEPT
        ,DEC(b.avgsal,7,2) AS avg_dept           -- ---- -------- --------
FROM     staff  a                                10   20 18357.50 16071.52
LEFT OUTER JOIN                                  20   20 18171.25 16071.52
        (SELECT   dept         AS dept           30   38 17506.75        -
                 ,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;
```
Figure 691, Join full-select to real table

Table Function Usage

If the full-select query has a reference to a row in a table that is outside of the full-select, then it needs to be written as a TABLE function call. In the next example, the preceding "A" table is referenced in the full-select, and so the TABLE function call is required:

```
SELECT   a.id                                    ANSWER
        ,a.dept                                  =========================
        ,a.salary                                ID DEPT SALARY   DEPTSAL
        ,b.deptsal                               -- ---- -------- --------
FROM     staff  a                                10 20   18357.50 64286.10
        ,TABLE                                   20 20   18171.25 64286.10
        (SELECT   b.dept                         30 38   17506.75 77285.55
                 ,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;
```
Figure 692, Full-select with external table reference

Below is the same query written without the reference to the "A" table in the full-select, and thus without a TABLE function call:

```
SELECT   a.id                               ANSWER
        ,a.dept                             =========================
        ,a.salary                           ID DEPT SALARY   DEPTSAL
        ,b.deptsal                          -- ---- -------- --------
FROM     staff  a                           10 20   18357.50 64286.10
       ,(SELECT   b.dept                    20 20   18171.25 64286.10
                 ,SUM(b.salary) AS deptsal  30 38   17506.75 77285.55
         FROM     staff b
         GROUP BY b.dept
        )AS b
WHERE    a.id   < 40
  AND    b.dept = a.dept
ORDER BY a.id;
```

Figure 693, Full-select without external table reference

Any externally referenced table in a full-select must be defined in the query syntax (starting at the first FROM statement) before the full-select. Thus, in the first example above, if the "A" table had been listed after the "B" table, then the query would have been invalid.

Full-Select in SELECT Phrase

A full-select that returns a single column and row can be used in the SELECT part of a query:

```
SELECT   id                                      ANSWER
        ,salary                                  ====================
        ,(SELECT MAX(salary)                     ID SALARY   MAXSAL
          FROM   staff                           -- -------- --------
         ) AS maxsal                             10 18357.50 22959.20
FROM     staff  a                                20 18171.25 22959.20
WHERE    id  < 60                                30 17506.75 22959.20
ORDER BY id;                                     40 18006.00 22959.20
                                                 50 20659.80 22959.20
```

Figure 694, Use an uncorrelated Full-Select in a SELECT list

A full-select 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.

```
SELECT   id                                      ANSWER
        ,salary                                  ====================
        ,(SELECT MAX(salary)                     ID SALARY   MAXSAL
          FROM   staff  b                        -- -------- --------
          WHERE  a.dept = b.dept                 10 18357.50 18357.50
         ) AS maxsal                             20 18171.25 18357.50
FROM     staff  a                                30 17506.75 18006.00
WHERE    id  < 60                                40 18006.00 18006.00
ORDER BY id;                                     50 20659.80 20659.80
```

Figure 695, Use a correlated Full-Select in a SELECT list

```
SELECT id                            ANSWER
      ,dept                          ==================================
      ,salary                        ID DEPT  SALARY   4        5
      ,(SELECT MAX(salary)           -- ---- -------- -------- --------
        FROM   staff b               10   20 18357.50 18357.50 22959.20
        WHERE  b.dept = a.dept)      20   20 18171.25 18357.50 22959.20
      ,(SELECT MAX(salary)           30   38 17506.75 18006.00 22959.20
        FROM   staff)                40   38 18006.00 18006.00 22959.20
FROM   staff a                       50   15 20659.80 20659.80 22959.20
WHERE  id  < 60
ORDER BY id;
```

Figure 696, Use correlated and uncorrelated Full-Selects in a SELECT list

INSERT Usage

The following query uses both an uncorrelated and correlated full-select in the query that builds the set of rows to be inserted:

```
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);
```

Figure 697, Full-select in INSERT

UPDATE Usage

The following example uses an uncorrelated full-select to assign a set of workers the average salary in the company - plus two thousand dollars.

```
UPDATE staff a                              ANSWER:      SALARY
SET    salary =                             ======= =================
       (SELECT AVG(salary)+ 2000            ID DEPT BEFORE   AFTER
        FROM   staff)                       -- ---- -------- --------
WHERE  id  < 60;                            10   20 18357.50 18675.64
                                            20   20 18171.25 18675.64
                                            30   38 17506.75 18675.64
                                            40   38 18006.00 18675.64
                                            50   15 20659.80 18675.64
```

Figure 698, Use uncorrelated Full-Select to give workers company AVG salary (+$2000)

The next statement uses a correlated full-select 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 full-select is resolved before the first update was done, not after each.

```
UPDATE staff a                              ANSWER:      SALARY
SET    salary =                             ======= =================
       (SELECT AVG(salary) + 2000           ID DEPT BEFORE   AFTER
        FROM   staff  b                     -- ---- -------- --------
        WHERE  a.dept = b.dept )            10   20 18357.50 18071.52
WHERE  id  < 60;                            20   20 18171.25 18071.52
                                            30   38 17506.75 17457.11
                                            40   38 18006.00 17457.11
                                            50   15 20659.80 17482.33
```

Figure 699, Use correlated Full-Select to give workers department AVG salary (+$2000)

NOTE: A full-select 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.

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. The type of 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.

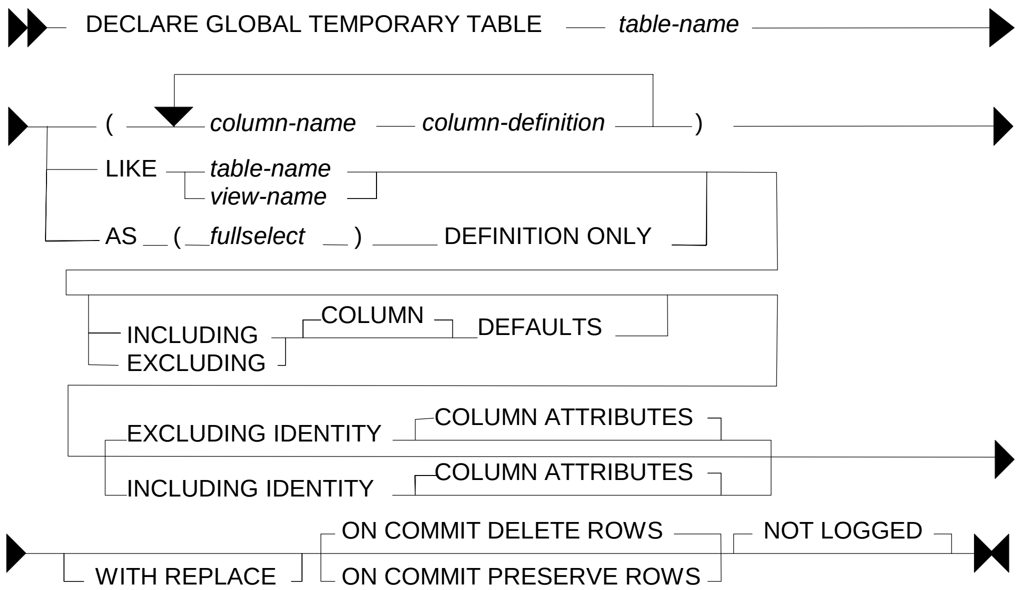

Figure 700, Declared Global Temporary Table syntax

Below is an example of declaring a global temporary table the old fashioned way:

```
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;
```
Figure 701, Declare Global Temporary Table - define columns

In the next example, the temporary table is defined to have exactly the same columns as the existing STAFF table:

```
DECLARE GLOBAL TEMPORARY TABLE session.fred
LIKE staff INCLUDING COLUMN DEFAULTS
WITH REPLACE
ON COMMIT PRESERVE ROWS;
```
Figure 702, Declare Global Temporary Table - like another table

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 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;
```
Figure 703, Declare Global Temporary Table - like query output

Indexes can be added to temporary tables in order to improve performance and/or to enforce uniqueness:

```
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;
                                                            ANSWER
SELECT  COUNT(*)                                            ======
FROM    session.fred;                                           19

COMMIT;
                                                            ANSWER
SELECT  COUNT(*)                                            ======
FROM    session.fred;                                            0
```

Figure 704, Temporary table with index

A temporary table has to be dropped to reuse the same name:

```
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;
                                                            ANSWER
SELECT  COUNT(*)                                            ======
FROM    session.fred;                                            8

DROP TABLE session.fred;

DECLARE GLOBAL TEMPORARY TABLE session.fred
(dept          SMALLINT    NOT NULL)
ON COMMIT DELETE ROWS;
                                                            ANSWER
SELECT  COUNT(*)                                            ======
FROM    session.fred;                                            0
```

Figure 705, Dropping a temporary table

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 qualifier, if provided, must be SESSION. If the qualifier is not provided, it is assumed to be SESSION. If the temporary table already exists, the WITH REPLACE clause must be used to override it.
- 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, except the following: BLOB, CLOB, DBCLOB, LONG VARCHAR, LONG VARGRAPHIC, DATALINK, reference, and structured data types.
- One can choose to preserve or delete (the default) the rows when a commit occurs.
- Standard identity column definitions can be added if desired.

- Changes are not logged.

Before a user can create a declared global temporary table, a USER TEMPORARY table-space that they have access to, has to be created. A typical definition follows:

```
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;
```

Figure 706, Create USER TEMPORARY tablespace

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.

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.

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                                                 AAA
+---------------+                                          |
|PKEY |CKEY |NUM|                                     +-----+-----+
|-----|-----|---|                                     |     |     |
|AAA  |BBB  |  1|                                    BBB   CCC   DDD
|AAA  |CCC  |  5|                                           |     |
|AAA  |DDD  | 20|                                          +-+ +-+--+
|CCC  |EEE  | 33|                                            | |    |
|DDD  |EEE  | 44|                                            EEE   FFF
|DDD  |FFF  |  5|                                                   |
|FFF  |GGG  |  5|                                                   |
+---------------+                                                  GGG
```

Figure 707, Sample Table description - Recursion

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:

```
WITH parent (pkey, ckey) AS                            ANSWER
  (SELECT pkey, ckey                                  =========      PROCESSING
   FROM   hierarchy                                   PKEY CKEY       SEQUENCE
   WHERE  pkey = 'AAA'                                ---- ----      ==========
   UNION ALL                                          AAA  BBB     < 1st pass
   SELECT C.pkey, C.ckey                              AAA  CCC       ""
   FROM   hierarchy C                                 AAA  DDD       ""
         ,parent    P                                 CCC  EEE     < 2nd pass
   WHERE  P.ckey = C.pkey                             DDD  EEE     < 3rd pass
  )                                                   DDD  FFF       ""
SELECT pkey, ckey                                     FFF  GGG     < 4th pass
FROM   parent;
```

Figure 708, SQL that does Recursion

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:

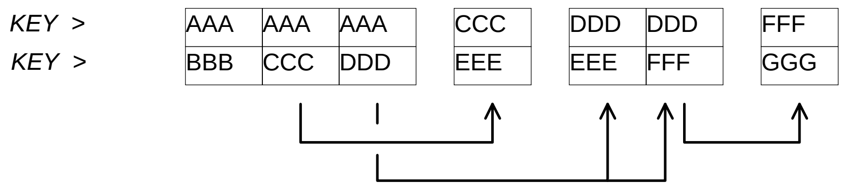

Figure 709, Recursive processing sequence

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.
- Recursive SQL is usually a fairly efficient. When it involves a join similar to the example shown above, it is important to make sure that this join is done efficiently. To this end, suitable indexes should always 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" on page 266.

Sample Table DDL & DML

```
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;
```

Figure 710, Sample Table DDL - Recursion

Introductory Recursion

This section will use recursive SQL statements to answer a series of simple business questions using the sample HIERARCHY table described on page 257. Be warned that things are going to get decidedly more complex as we proceed.

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.

```
WITH parent (ckey) AS                      ANSWER   HIERARCHY
  (SELECT ckey                             ======   +---------------+
   FROM   hierarchy                        CKEY     |PKEY |CKEY |NUM|
   WHERE  pkey = 'AAA'                     ----     |-----|-----|---|
   UNION ALL                               BBB      |AAA  |BBB  |  1|
   SELECT C.ckey                           CCC      |AAA  |CCC  |  5|
   FROM   hierarchy C                      DDD      |AAA  |DDD  | 20|
         ,parent    P                      EEE      |CCC  |EEE  | 33|
   WHERE  P.ckey = C.pkey                  EEE      |DDD  |EEE  | 44|
  )                                        FFF      |DDD  |FFF  |  5|
SELECT ckey                                GGG      |FFF  |GGG  |  5|
FROM   parent;                                      +---------------+
```

Figure 711, List of children of AAA

> WARNING: Much of the SQL shown in this section will loop forever if the target database has a recursive data structure. See page 266 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 top-level parent.

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.

```
WITH parent (ckey) AS                          ANSWER   HIERARCHY
  (SELECT DISTINCT pkey                        ======   +---------------+
   FROM   hierarchy                            CKEY     |PKEY |CKEY |NUM|
   WHERE  pkey = 'AAA'                         ----     |-----|-----|---|
   UNION ALL                                   AAA      |AAA  |BBB  |  1|
   SELECT C.ckey                               BBB      |AAA  |CCC  |  5|
   FROM   hierarchy C                          CCC      |AAA  |DDD  | 20|
         ,parent    P                          DDD      |CCC  |EEE  | 33|
   WHERE  P.ckey = C.pkey                      EEE      |DDD  |EEE  | 44|
  )                                            EEE      |DDD  |FFF  |  5|
SELECT ckey                                    FFF      |FFF  |GGG  |  5|
FROM   parent;                                 GGG      +---------------+
```
Figure 712, List all children of AAA

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.

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.

```
WITH parent (ckey) AS                          ANSWER   HIERARCHY
  (SELECT DISTINCT pkey                        ======   +---------------+
   FROM   hierarchy                            CKEY     |PKEY |CKEY |NUM|
   WHERE  pkey = 'AAA'                         ----     |-----|-----|---|
   UNION ALL                                   AAA      |AAA  |BBB  |  1|
   SELECT C.ckey                               BBB      |AAA  |CCC  |  5|
   FROM   hierarchy C                          CCC      |AAA  |DDD  | 20|
         ,parent    P                          DDD      |CCC  |EEE  | 33|
   WHERE  P.ckey = C.pkey                      EEE      |DDD  |EEE  | 44|
  )                                            FFF      |DDD  |FFF  |  5|
SELECT DISTINCT ckey                           GGG      |FFF  |GGG  |  5|
FROM   parent;                                          +---------------+
```
Figure 713, List distinct children of AAA

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.

```
WITH parent (ckey) AS                          ANSWER   HIERARCHY
  (SELECT DISTINCT pkey                        ======   +---------------+
   FROM   hierarchy                            CKEY     |PKEY |CKEY |NUM|
   WHERE  pkey = 'AAA'                         ----     |-----|-----|---|
   UNION ALL                                   AAA      |AAA  |BBB  |  1|
   SELECT C.ckey                               BBB      |AAA  |CCC  |  5|
   FROM   hierarchy C                          CCC      |AAA  |DDD  | 20|
         ,parent    P                          DDD      |CCC  |EEE  | 33|
   WHERE  P.ckey = C.pkey                      EEE      |DDD  |EEE  | 44|
  ),                                           FFF      |DDD  |FFF  |  5|
distinct_parent (ckey) AS                      GGG      |FFF  |GGG  |  5|
  (SELECT DISTINCT ckey                                 +---------------+
   FROM   parent
  )
SELECT ckey
FROM   distinct_parent;
```
Figure 714, List distinct children of AAA

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.

```
WITH parent (ckey, lvl) AS                        ANSWER          AAA
  (SELECT DISTINCT pkey, 0                        ========         |
   FROM   hierarchy                               CKEY LVL    +-----+-----+
   WHERE  pkey = 'AAA'                            ---- ---    |     |     |
   UNION ALL                                      AAA    0   BBB   CCC   DDD
   SELECT C.ckey, P.lvl +1                        BBB    1          |     |
   FROM   hierarchy C                             CCC    1         +-+ +-+--+
         ,parent    P                             DDD    1           | |    |
   WHERE  P.ckey = C.pkey                         EEE    2           EEE   FFF
  )                                               EEE    2                  |
SELECT ckey, lvl                                  FFF    2                  |
FROM   parent;                                    GGG    3                 GGG
```

Figure 715, Show item level in hierarchy

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.

Select Certain Levels

Get a list of all the children of AAA that are less than three levels below AAA.

```
WITH parent (ckey, lvl) AS                         ANSWER     HIERARCHY
  (SELECT DISTINCT pkey, 0                         ========   +---------------+
   FROM   hierarchy                                CKEY LVL   |PKEY |CKEY |NUM|
   WHERE  pkey = 'AAA'                             ---- ---   |-----|-----|---|
   UNION ALL                                       AAA    0   |AAA  |BBB  |  1|
   SELECT C.ckey, P.lvl +1                         BBB    1   |AAA  |CCC  |  5|
   FROM   hierarchy C                              CCC    1   |AAA  |DDD  | 20|
         ,parent    P                              DDD    1   |CCC  |EEE  | 33|
   WHERE  P.ckey  = C.pkey                         EEE    2   |DDD  |EEE  | 44|
  )                                                EEE    2   |DDD  |FFF  |  5|
SELECT ckey, lvl                                   FFF    2   |FFF  |GGG  |  5|
FROM   parent                                                 +---------------+
WHERE  lvl < 3;
```

Figure 716, Select rows where LEVEL < 3

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:

```
WITH parent (ckey, lvl) AS                        ANSWER          AAA
  (SELECT DISTINCT pkey, 0                        ========         |
   FROM   hierarchy                               CKEY LVL    +-----+-----+
   WHERE  pkey = 'AAA'                            ---- ---    |     |     |
   UNION ALL                                      AAA    0   BBB   CCC   DDD
   SELECT C.ckey, P.lvl +1                        BBB    1          |     |
   FROM   hierarchy C                             CCC    1         +-+ +-+--+
         ,parent    P                             DDD    1           | |    |
   WHERE  P.ckey  = C.pkey                        EEE    2           EEE   FFF
     AND  P.lvl+1 < 3                             EEE    2                  |
  )                                               FFF    2                  |
SELECT ckey, lvl                                                           GGG
FROM   parent;
```

Figure 717, Select rows where LEVEL < 3

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).

Select Explicit Level

Get a list of all the children of AAA that are exactly two levels below AAA.

```
WITH parent (ckey, lvl) AS                    ANSWER      HIERARCHY
  (SELECT DISTINCT pkey, 0                    ========    +---------------+
   FROM   hierarchy                           CKEY LVL    |PKEY |CKEY |NUM|
   WHERE  pkey = 'AAA'                        ---- ---    |-----|-----|---|
   UNION ALL                                  EEE    2    |AAA  |BBB  |  1|
   SELECT C.ckey, P.lvl +1                    EEE    2    |AAA  |CCC  |  5|
   FROM   hierarchy C                         FFF    2    |AAA  |DDD  | 20|
         ,parent    P                                     |CCC  |EEE  | 33|
   WHERE  P.ckey  = C.pkey                                |DDD  |EEE  | 44|
     AND  P.lvl+1 < 3                                     |DDD  |FFF  |  5|
  )                                                       |FFF  |GGG  |  5|
SELECT ckey, lvl                                          +---------------+
FROM   parent
WHERE  lvl = 2;
```

Figure 718, Select rows where LEVEL = 2

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.

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.

```
WITH children (kkey, lvl) AS                 ANSWER          AAA
  (SELECT ckey, 1                             ========         |
   FROM   hierarchy                           KKEY LVL    +-----+-----+
   WHERE  pkey = 'DDD'                        ---- ---    |     |     |
   UNION ALL                                  AAA   -1   BBB   CCC   DDD
   SELECT H.ckey, C.lvl + 1                   EEE    1                |     |
   FROM   hierarchy H                         FFF    1              +-+ +-+--+
         ,children  C                         GGG    2                | |    |
   WHERE  H.pkey = C.kkey                                            EEE   FFF
  )                                                                         |
,parents (kkey, lvl) AS                                                     |
  (SELECT pkey, -1                                                         GGG
   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;
```

Figure 719, Find all children and parents of DDD

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:

```
WITH temp1 (n1) AS                                                    ANSWER
  (SELECT id                                                          ======
   FROM   staff                                                           N1
   WHERE  id = 10                                                         --
   UNION ALL                                                            warn
   SELECT n1 +10                                                          10
   FROM   temp1                                                           20
   WHERE  n1 <  50                                                        30
  )                                                                       40
SELECT *                                                                  50
FROM   temp1;
```

Figure 720, Recursion - with warning message

```
WITH temp1 (n1) AS                                                    ANSWER
  (SELECT INT(id)                                                     ======
   FROM   staff                                                           N1
   WHERE  id = 10                                                         --
   UNION ALL                                                              10
   SELECT n1 +10                                                          20
   FROM   temp1                                                           30
   WHERE  n1 <  50                                                        40
  )                                                                       50
SELECT *
FROM   temp1;
```

Figure 721, Recursion - without warning message

If you know what you are doing, ignore the message.

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.

```
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
```

Figure 722, Hierarchy Flavours

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

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

```
          AAA
           |
     +-----+-----+
     |     |     |
    BBB   CCC   DDD
                 |
              +--+--+
              |     |
             EEE   FFF
                    |
                    |
                   GGG
```

Figure 723, Divergent Hierarchy - Table and Layout

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.

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.

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

```
      AAA
       |
 +-----+-----+
 |     |     |
BBB   CCC   DDD
       |     |
       +-+ +-+--+
         | |    |
        EEE    FFF
                |
                |
               GGG
```

Figure 724, Convergent Hierarchy - Tables and Layout

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.

Recursive Hierarchy

WARNING: Recursive data hierarchies will cause poorly written recursive SQL statements to run forever. See the section titled "Halting Recursive Processing" on page 266 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.

```
OBJECTS          RELATIONSHIPS                        AAA <------+
+-----------+    +---------------+                     |         |
|KEYO |PRICE|    |PKEY |CKEY |NUM|               +-----+-----+   |
|-----|-----|    |-----|-----|---|               |     |     |   |
|AAA  |  $10|    |AAA  |BBB  |  1|              BBB   CCC   DDD>-+
|BBB  |  $21|    |AAA  |CCC  |  5|                     |     |
|CCC  |  $23|    |AAA  |DDD  | 20|                     +-+ +-+--+
|DDD  |  $25|    |CCC  |EEE  | 33|                       | |    |
|EEE  |  $33|    |DDD  |AAA  | 99|                       EEE   FFF
|FFF  |  $34|    |DDD  |FFF  |  5|                              |
|GGG  |  $44|    |DDD  |EEE  | 44|                              |
+-----------+    |FFF  |GGG  |  5|                             GGG
                 +---------------+
```

Figure 725, Recursive Hierarchy - Tables and Layout

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 page 266 for details.

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.

```
      AAA              << Balanced hierarchy                AAA
       |                  Unbalanced hierarchy >>            |
 +-----+-----+                                          +---+----+
 |     |     |                                          |   |    |
BBB   CCC   DDD                                         |  CCC  DDD
 |     |     |                                          |   |    |
 |     |   +-+-+                                        | +-+  +-+-+
 |     |   |   |                                        | |    |   |
EEE   FFF GGG HHH                                       FFF   GGG HHH
                                                                   |
                                                                   |
                                                                  III
```

Figure 726, 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.

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.

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.

```
TROUBLE                                                     AAA <------+
+---------+                                                  |         |
|PKEY|CKEY|                                            +-----+-----+   |
|----|----|                                            |     |     |   |
|AAA |BBB |                                           BBB   CCC   DDD>-+
|AAA |CCC |                                                  |     |
|AAA |DDD |                                                +-+  +-+--+
|CCC |EEE |                                                | |  |    |
|DDD |AAA |    <===  This row                               EEE     FFF
|DDD |FFF |    points back to                                        |
|DDD |EEE |    the hierarchy                                         |
|FFF |GGG |    parent.                                              GGG
+---------+
```
Figure 727, Recursive Hierarchy - Sample Table and Layout

Below is the DDL and DML that was used to create the above table.

```
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');
```

Figure 728, Sample Table DDL - Recursive Hierarchy

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.

```
WITH parent (pkey, ckey, lvl) AS                ANSWER              TROUBLE
  (SELECT DISTINCT                              =============       +---------+
          pkey                                  PKEY CKEY LVL       |PKEY|CKEY|
         ,pkey                                  ---- ---- ---       |----|----|
         ,0                                     AAA  AAA    0       |AAA |BBB |
   FROM   trouble                               AAA  BBB    1       |AAA |CCC |
   WHERE  pkey = 'AAA'                          AAA  CCC    1       |AAA |DDD |
   UNION ALL                                    AAA  DDD    1       |CCC |EEE |
   SELECT C.pkey                                CCC  EEE    2       |DDD |AAA |
         ,C.ckey                                DDD  AAA    2       |DDD |FFF |
         ,P.lvl + 1                             DDD  EEE    2       |DDD |EEE |
   FROM   trouble  C                            DDD  FFF    2       |FFF |GGG |
         ,parent   P                            AAA  BBB    3       +---------+
   WHERE  P.ckey      = C.pkey                  AAA  CCC    3
     AND  P.lvl + 1 < 4                         AAA  DDD    3
  )                                             FFF  GGG    3
SELECT *
FROM   parent;
```

Figure 729, Stop Recursive SQL after "n" levels

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.

```
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
```

Figure 730, LOCATE_BLOCK user defined function

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:

```
SELECT id                                                ANSWER
      ,NAME                                              =================
      ,LOCATE('th',name)       AS L1                     ID  NAME    L1 L2
      ,LOCATE_BLOCK('th',name) AS L2                     --- ------- -- --
FROM   staff                                              70 Rothman  3  2
WHERE  LOCATE('th',name) > 1;                            220 Smith    4  0
```

Figure 731, LOCATE_BLOCK function example

NOTE: 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:

```
WITH parent (pkey, ckey, lvl, path, loop) AS
  (SELECT DISTINCT
          pkey
         ,pkey                           ANSWER
         ,0                              ===============================
         ,VARCHAR(pkey,20)               PKEY CKEY LVL PATH         LOOP
         ,0                              ---- ---- --- ------------ ----
   FROM   trouble                        AAA  AAA    0 AAA             0
   WHERE  pkey = 'AAA'                   AAA  BBB    1 AAABBB          0
   UNION ALL                             AAA  CCC    1 AAACCC          0
   SELECT C.pkey                         AAA  DDD    1 AAADDD          0
         ,C.ckey                         CCC  EEE    2 AAACCCEEE       0
         ,P.lvl + 1                      DDD  AAA    2 AAADDDAAA       1
         ,P.path || C.ckey               DDD  EEE    2 AAADDDEEE       0
         ,LOCATE_BLOCK(C.ckey,P.path)    DDD  FFF    2 AAADDDFFF       0
   FROM   trouble C                      AAA  BBB    3 AAADDDAAABBB    0
         ,parent  P                      AAA  CCC    3 AAADDDAAACCC    0
   WHERE  P.ckey      = C.pkey           AAA  DDD    3 AAADDDAAADDD    2
     AND  P.lvl + 1 < 4                  FFF  GGG    3 AAADDDFFFGGG    0
  )
SELECT *
FROM   parent;                           TROUBLE
                                         +---------+         AAA <------+
                                         |PKEY|CKEY|          |         |
                                         |----|----|    +-----+-----+   |
                                         |AAA |BBB |    |     |     |   |
                                         |AAA |CCC |   BBB   CCC   DDD>-+
                                         |AAA |DDD |          |     |
                                         |CCC |EEE |          +-+ +-+--+
                        This row  ===>   |DDD |AAA |            | |    |
                        points back to   |DDD |FFF |           EEE    FFF
                        the hierarchy    |DDD |EEE |                   |
                        parent.          |FFF |GGG |                   |
                                         +---------+                  GGG
```

Figure 732, Show path, and rows in loop

Now we can get rid of the level check, and instead use the LOCATE_BLOCK function to avoid loops in the data:

```
WITH parent (pkey, ckey, lvl, path) AS      ANSWER
  (SELECT DISTINCT                          ==========================
          pkey                              PKEY CKEY LVL PATH
         ,pkey                              ---- ----- -- ------------
         ,0                                 AAA  AAA    0 AAA
         ,VARCHAR(pkey,20)                  AAA  BBB    1 AAABBB
   FROM   trouble                           AAA  CCC    1 AAACCC
   WHERE  pkey = 'AAA'                      AAA  DDD    1 AAADDD
   UNION ALL                                CCC  EEE    2 AAACCCEEE
   SELECT C.pkey                            DDD  EEE    2 AAADDDEEE
         ,C.ckey                            DDD  FFF    2 AAADDDFFF
         ,P.lvl + 1                         FFF  GGG    3 AAADDDFFFGGG
         ,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;
```

Figure 733, Use LOCATE_BLOCK function to stop recursion

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;

```
WITH parent (pkey, ckey, lvl, path, loop) AS
  (SELECT DISTINCT
          pkey
         ,pkey
         ,0
         ,VARCHAR(pkey,20)             ANSWER
         ,0                            ===============================
   FROM   trouble                      PKEY CKEY LVL PATH         LOOP
   WHERE  pkey = 'AAA'                 ---- ---- --- ------------ ----
   UNION ALL                           AAA  AAA    0 AAA             0
   SELECT C.pkey                       AAA  BBB    1 AAABBB          0
         ,C.ckey                       AAA  CCC    1 AAACCC          0
         ,P.lvl + 1                    AAA  DDD    1 AAADDD          0
         ,P.path || C.ckey             CCC  EEE    2 AAACCCEEE       0
         ,LOCATE_BLOCK(C.ckey,P.path)  DDD  AAA    2 AAADDDAAA       1
   FROM   trouble C                    DDD  EEE    2 AAADDDEEE       0
         ,parent  P                    DDD  FFF    2 AAADDDFFF       0
   WHERE  P.ckey = C.pkey              FFF  GGG    3 AAADDDFFFGGG    0
     AND  P.loop = 0
  )
SELECT *
FROM   parent;
```

Figure 734, Use LOCATE_BLOCK function to stop recursion

The next query tosses in another predicate (in the final select) to only list those rows that point back to a previously processed parent:

```
WITH parent (pkey, ckey, lvl, path, loop) AS                ANSWER
  (SELECT DISTINCT                                          =========
          pkey                                              PKEY CKEY
         ,pkey                                              ---- ----
         ,0                                                 DDD  AAA
         ,VARCHAR(pkey,20)
         ,0
   FROM   trouble
   WHERE  pkey = 'AAA'
   UNION ALL
   SELECT C.pkey
         ,C.ckey                                          TROUBLE
         ,P.lvl + 1                                       +---------+
         ,P.path || C.ckey                                |PKEY|CKEY|
         ,LOCATE_BLOCK(C.ckey,P.path)                     |----|----|
   FROM   trouble C                                       |AAA |BBB |
         ,parent  P                                       |AAA |CCC |
   WHERE  P.ckey = C.pkey                                 |AAA |DDD |
     AND  P.loop = 0                                      |CCC |EEE |
  )                               This row  ===>          |DDD |AAA |
SELECT pkey                       points back to          |DDD |FFF |
      ,ckey                       the hierarchy           |DDD |EEE |
FROM   parent                     parent.                 |FFF |GGG |
WHERE  loop > 0;                                          +---------+
```

Figure 735,List rows that point back to a parent

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.

```
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);
```

```
                     TROUBLE
                     +---------+
                     |PKEY|CKEY|
                     |----|----|
                     |AAA |BBB |
                     |AAA |CCC |
                     |AAA |DDD |
                     |CCC |EEE |
This row  ===>       |DDD |AAA |
points back to       |DDD |FFF |
the hierarchy        |DDD |EEE |
parent.              |FFF |GGG |
                     +---------+

          AAA <------+
           |         |
     +-----+-----+   |
     |     |     |   |
    BBB   CCC   DDD>-+
           |     |
           +-+ +-+--+
             | |    |
             EEE   FFF
                    |
                    |
                   GGG
```

Figure 736, Delete rows that loop back to a parent

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:

```
CREATE TRIGGER TBL_INS                                          TROUBLE
NO CASCADE BEFORE INSERT ON trouble                             +---------+
REFERENCING NEW AS NNN                         This trigger     |PKEY|CKEY|
FOR EACH ROW MODE DB2SQL                       would reject     |----|----|
   WITH temp (pkey, ckey) AS                   insertion of     |AAA |BBB |
     (VALUES (NNN.pkey                         this row.        |AAA |CCC |
             ,NNN.ckey)                               |         |AAA |DDD |
      UNION ALL                                       |         |CCC |EEE |
      SELECT TTT.pkey                                 +--->     |DDD |AAA |
            ,CASE                                               |DDD |FFF |
                WHEN TTT.ckey = TBL.pkey                        |DDD |EEE |
                THEN RAISE_ERROR('70001','LOOP FOUND')          |FFF |GGG |
                ELSE TBL.ckey                                   +---------+
             END
      FROM   trouble TBL
            ,temp    TTT
      WHERE  TTT.ckey  = TBL.pkey
     )
   SELECT *
   FROM   temp;
```

Figure 737, INSERT trigger

Here is the 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;
```

Figure 738, UPDATE trigger

Given the above preexisting TROUBLE data (absent the DDD to AAA row), the following statements would be rejected by the above triggers:

```
INSERT INTO trouble VALUES('GGG','AAA');

UPDATE trouble SET ckey = 'AAA' WHERE pkey = 'FFF';
UPDATE trouble SET pkey = 'GGG' WHERE ckey = 'DDD';
```

Figure 739, Invalid DML statements

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.

Clean Hierarchies and Efficient Joins

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.

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.

```
                        HIERARCHY#1                      EXPLODED#1
 AAA                    +--------------------+           +-------------+
  |                     |KEYY|PKEY|DATA      |           |PKEY|CKEY|LVL|
 BBB                    |----|----|----------|           |----|----|---|
  |                     |AAA |AAA |SOME DATA |           |AAA |AAA |  0|
  +-----+               |BBB |AAA |MORE DATA |           |AAA |BBB |  1|
  |     |               |CCC |BBB |MORE JUNK |           |AAA |CCC |  2|
 CCC   EEE              |DDD |CCC |MORE JUNK |           |AAA |DDD |  3|
  |                     |EEE |BBB |JUNK DATA |           |AAA |EEE |  2|
 DDD                    +--------------------+           |BBB |BBB |  0|
                                                         |BBB |CCC |  1|
                                                         |BBB |DDD |  2|
                                                         |BBB |EEE |  1|
                                                         |CCC |CCC |  0|
                                                         |CCC |DDD |  1|
                                                         |DDD |DDD |  0|
                                                         |EEE |EEE |  0|
                                                         +-------------+
```

Figure 740, Data Hierarchy, with normalized and exploded representations

Below is the CREATE code for the above normalized table and a dependent trigger:

```
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');
```

Figure 741, Hierarchy table that does not allow updates to 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.

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:

```
CREATE TABLE exploded#1
(pkey  CHAR(4)   NOT NULL
,ckey  CHAR(4)   NOT NULL
,lvl  SMALLINT   NOT NULL
,PRIMARY KEY(pkey,ckey));
```

Figure 742, Exploded table CREATE statement

The following trigger deletes all dependent rows from the exploded table whenever a row is deleted from the 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;
```

Figure 743, Trigger to maintain exploded table after delete in hierarchy table

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:

```
CREATE TRIGGER EXP#1_INS                        HIERARCHY#1         EXPLODED#1
AFTER INSERT ON hierarchy#1                     +--------------+    +-------------+
REFERENCING NEW AS NNN                          |KEYY|PKEY|DATA|    |PKEY|CKEY|LVL|
FOR EACH ROW MODE DB2SQL                        |----|----|----|    |----|----|---|
   INSERT                                       |AAA |AAA |S...|    |AAA |AAA |  0|
   INTO exploded#1                              |BBB |AAA |M...|    |AAA |BBB |  1|
   WITH temp(pkey, ckey, lvl) AS                |CCC |BBB |M...|    |AAA |CCC |  2|
     (VALUES (NNN.keyy                          |DDD |CCC |M...|    |AAA |DDD |  3|
             ,NNN.keyy                          |EEE |BBB |J...|    |AAA |EEE |  2|
             ,0)                                +--------------+    |BBB |BBB |  0|
      UNION ALL                                                     |BBB |CCC |  1|
      SELECT  N.pkey                                                |BBB |DDD |  2|
             ,NNN.keyy                                              |BBB |EEE |  1|
             ,T.lvl +1                                              |CCC |CCC |  0|
      FROM    temp       T                                          |CCC |DDD |  1|
             ,hierarchy#1 N                                         |DDD |DDD |  0|
      WHERE   N.keyy  = T.pkey                                      |EEE |EEE |  0|
        AND   N.keyy <> N.pkey                                      +-------------+
     )
   SELECT *
   FROM   temp;
```

Figure 744, Trigger to maintain exploded table after insert in hierarchy table

There is no update trigger because updates are not allowed to the hierarchy table.

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.

```
SELECT   *
FROM     exploded#1
WHERE    pkey = :host-var
ORDER BY pkey
        ,ckey
        ,lvl;
```

Figure 745, Querying the exploded table

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:

```
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);
```

Figure 746, Hierarchy table that allows updates to PKEY

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:

```
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...

Figure 747, Trigger to check for recursive data structures before update of PKEY

NOTE: 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.

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:

```
CREATE TABLE exploded#2
(pkey  CHAR(4)   NOT NULL
,ckey  CHAR(4)   NOT NULL
,lvl   SMALLINT  NOT NULL
,PRIMARY KEY(pkey,ckey));
```

Figure 748, Exploded table CREATE statement

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:

```
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;
```

Figure 749, Trigger to maintain exploded table after delete in hierarchy table

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:

```
CREATE TRIGGER EXP#2_INS                    HIERARCHY#2        EXPLODED#2
AFTER INSERT ON hierarchy#2                 +--------------+   +-------------+
REFERENCING NEW AS NNN                      |KEYY|PKEY|DATA|   |PKEY|CKEY|LVL|
FOR EACH ROW MODE DB2SQL                    |----|----|----|   |----|----|---|
   INSERT                                   |AAA |AAA |S...|   |AAA |AAA |  0|
   INTO   exploded#2                        |BBB |AAA |M...|   |AAA |BBB |  1|
   WITH temp(pkey, ckey, lvl) AS            |CCC |BBB |M...|   |AAA |CCC |  2|
     (SELECT  NNN.keyy                      |DDD |CCC |M...|   |AAA |DDD |  3|
             ,NNN.keyy                      |EEE |BBB |J...|   |AAA |EEE |  2|
             ,0                             +--------------+   |BBB |BBB |  0|
      FROM    hierarchy#2                                      |BBB |CCC |  1|
      WHERE   keyy = NNN.keyy                                  |BBB |DDD |  2|
      UNION ALL                                                |BBB |EEE |  1|
      SELECT  N.pkey                                           |CCC |CCC |  0|
             ,NNN.keyy                                         |CCC |DDD |  1|
             ,T.lvl +1                                         |DDD |DDD |  0|
      FROM    temp        T                                    |EEE |EEE |  0|
             ,hierarchy#2 N                                    +-------------+
      WHERE   N.keyy  = T.pkey
        AND   N.keyy <> N.pkey
     )
   SELECT *
   FROM   temp;
```

Figure 750, Trigger to maintain exploded table after insert in hierarchy table

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.

```
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
     )
```

Figure 751, Trigger to run after update of PKEY in hierarchy table (part 1 of 2)

```
    ,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
```

Figure 752, Trigger to run after update of PKEY in hierarchy table (part 2 of 2)

NOTE: 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.

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.

```
SELECT   *
FROM     exploded#2
WHERE    pkey = :host-var
ORDER BY pkey
        ,ckey
        ,lvl;
```

Figure 753, Querying the exploded table

Below are some suggested indexes:

- PKEY, CKEY (already defined as part of the primary key).
- CKEY, PKEY (useful when joining to this table).

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.

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.

Create a Row of Data

Select a single column/row entity, but do not use a table or view as the data source.

```
WITH TEMP1 (COL1) AS                                          ANSWER
(VALUES     0                                                 ======
)                                                               COL1
SELECT *                                                        ----
FROM   TEMP1;                                                      0
```

Figure 754, Select one row/column using VALUES

The above statement uses the VALUES statement to define a single row/column in the temporary table TEMP1. This table is then selected from.

Create "n" Rows & Columns of Data

Select multiple rows and columns, but do not use a table or view as the data source.

```
WITH TEMP1 (COL1, COL2, COL3) AS                        ANSWER
(VALUES     (   0, 'AA', 0.00)                          ==============
           ,(   1, 'BB', 1.11)                          COL1 COL2 COL3
           ,(   2, 'CC', 2.22)                          ---- ---- ----
)                                                          0 AA   0.00
SELECT *                                                   1 BB   1.11
FROM   TEMP1;                                              2 CC   2.22
```

Figure 755, Select multiple rows/columns using VALUES

This statement places three rows and columns of data into the temporary table TEMP1, which is then selected from. Note that each row of values is surrounded by parenthesis and separated from the others by a comma.

Linear 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.

```
WITH TEMP1 (COL1) AS                                              ANSWER
(VALUES     0                                                     ======
 UNION ALL                                                        COL1
 SELECT COL1 + 1                                                  ----
 FROM   TEMP1                                                        0
 WHERE  COL1 + 1 < 100                                               1
)                                                                    2
SELECT *                                                             3
FROM   TEMP1;                                                      etc
```

Figure 756, Use recursion to get list of one hundred numbers

The first part of the above recursive statement refers to a single row that has the value zero. Note that no table or view is selected from in this part of the query, the row is defined using a VALUES phrase. In the second part of the statement the original row is recursively added to itself ninety nine times.

Tabular Data Generation

Create the complete set of integers between zero and one hundred. Display ten numbers in each line of output.

```
WITH TEMP1 (C0,C1,C2,C3,C4,C5,C6,C7,C8,C9) AS
(VALUES    ( 0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
 UNION ALL
 SELECT C0+10, C1+10, C2+10, C3+10, C4+10
       ,C5+10, C6+10, C7+10, C8+10, C9+10
 FROM   TEMP1
 WHERE  C0+10 < 100
)
SELECT *
FROM   TEMP1;
```

Figure 757, Recursive SQL used to make an array of numbers (1 of 2)

The result follows, it is of no functional use, but it looks cute:

```
  C0    C1    C2    C3    C4    C5    C6    C7    C8    C9
----  ----  ----  ----  ----  ----  ----  ----  ----  ----
   0     1     2     3     4     5     6     7     8     9
  10    11    12    13    14    15    16    17    18    19
  20    21    22    23    24    25    26    27    28    29
  30    31    32    33    34    35    36    37    38    39
  40    41    42    43    44    45    46    47    48    49
  50    51    52    53    54    55    56    57    58    59
  60    61    62    63    64    65    66    67    68    69
  70    71    72    73    74    75    76    77    78    79
  80    81    82    83    84    85    86    87    88    89
  90    91    92    93    94    95    96    97    98    99
```

Figure 758, Answer - array of numbers made using recursive SQL

Another way to get exactly the same answer is shown below. If differs from the prior SQL in that most of the arithmetic is deferred until the final select. Both statements do the job equally well, which one you prefer is mostly a matter of aesthetics.

```
WITH TEMP1 (C0) AS
(VALUES    ( 0)
 UNION ALL
 SELECT C0+10
 FROM   TEMP1
 WHERE  C0+10 < 100
)
SELECT C0
      ,C0+1 AS C1, C0+2 AS C2, C0+3 AS C3, C0+4 AS C4, C0+5 AS C5
      ,C0+6 AS C6, C0+7 AS C7, C0+8 AS C8, C0+9 AS C9
FROM   TEMP1;
```

Figure 759, Recursive SQL used to make an array of numbers (2 of 2)

Cosine vs. Degree - Table of Values

Create a report that shows the cosine of every angle between zero and ninety degrees (accurate to one tenth of a degree).

```
WITH TEMP1 (DEGREE) AS
(VALUES SMALLINT(0)
 UNION ALL
 SELECT SMALLINT(DEGREE + 1)
 FROM   TEMP1
 WHERE  DEGREE < 89
)
SELECT DEGREE
      ,DEC(COS(RADIANS(DEGREE + 0.0)),4,3) AS POINT0
      ,DEC(COS(RADIANS(DEGREE + 0.1)),4,3) AS POINT1
      ,DEC(COS(RADIANS(DEGREE + 0.2)),4,3) AS POINT2
      ,DEC(COS(RADIANS(DEGREE + 0.3)),4,3) AS POINT3
      ,DEC(COS(RADIANS(DEGREE + 0.4)),4,3) AS POINT4
      ,DEC(COS(RADIANS(DEGREE + 0.5)),4,3) AS POINT5
      ,DEC(COS(RADIANS(DEGREE + 0.6)),4,3) AS POINT6
      ,DEC(COS(RADIANS(DEGREE + 0.7)),4,3) AS POINT7
      ,DEC(COS(RADIANS(DEGREE + 0.8)),4,3) AS POINT8
      ,DEC(COS(RADIANS(DEGREE + 0.9)),4,3) AS POINT9
FROM   TEMP1;
```

Figure 760, SQL to make Cosine vs. Degree table

The answer (part of) follows:

```
DEGREE POINT0 POINT1 POINT2 POINT3 POINT4 POINT5 POINT6 POINT7 etc....
------ ------ ------ ------ ------ ------ ------ ------ ------
     0  1.000  0.999  0.999  0.999  0.999  0.999  0.999  0.999
     1  1.000  0.999  0.999  0.999  0.999  0.999  0.999  0.999
     2  0.999  0.999  0.999  0.999  0.999  0.999  0.999  0.999
     3  0.999  0.999  0.999  0.999  0.999  0.999  0.998  0.998
     4  0.998  0.998  0.998  0.998  0.998  0.998  0.998  0.997
     5  0.997  0.997  0.997  0.997  0.997  0.996  0.996  0.996
     6  0.994  0.994  0.994  0.993  0.993  0.993  0.993  0.993
     7  0.992  0.992  0.992  0.991  0.991  0.991  0.991  0.990
     8  0.990  0.990  0.989  0.989  0.989  0.989  0.988  0.988
     .
     .
    88  0.052  0.050  0.048  0.047  0.045  0.043  0.041  0.040
    89  0.034  0.033  0.031  0.029  0.027  0.026  0.024  0.022
```

Figure 761, Cosine vs. Degree SQL output

Make Reproducible Random Data

So far, all we have done is create different sets of fixed data. 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:

```
WITH TEMP1 (S1, R1) AS                                          ANSWER
(VALUES (0, RAND(1))                                            ============
 UNION ALL                                                      SEQ#    RAN1
 SELECT  S1+1, RAND()                                            ----  -----
 FROM    TEMP1                                                      0  0.001
 WHERE   S1+1 < 5                                                   1  0.563
)                                                                   2  0.193
SELECT SMALLINT(S1)    AS SEQ#                                      3  0.808
      ,DECIMAL(R1,5,3) AS RAN1                                      4  0.585
FROM   TEMP1;
```

Figure 762, Use RAND to create pseudo-random numbers

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.

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 page 116) for an example of how to use it to make random data.

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:

```
WITH TEMP1 (S1, R1) AS                          ANSWER
(VALUES (0, RAND(2))                            ========================
 UNION ALL                                      SEQ#  RAN2  RAN1    RAN3
 SELECT  S1+1, RAND()                           ----  ----  ------  ----
 FROM    TEMP1                                     0    13  0.0013     0
 WHERE   S1+1 < 5                                  1  8916  0.8916     8
)                                                  2  7384  0.7384     7
SELECT SMALLINT(S1) AS SEQ#                        3  5430  0.5430     5
      ,SMALLINT(R1*10000) AS RAN2                  4  8998  0.8998     8
      ,DECIMAL(R1,6,4) AS RAN1
      ,SMALLINT(R1*10) AS RAN3
FROM   TEMP1;
```

Figure 763, Make differing ranges of random numbers

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.

```
WITH TEMP1 (S1, R1) AS                                   ANSWER
(VALUES (0, RAND(2))                                     ===================
 UNION ALL                                               SEQ# RAN2 RAN3 RAN4
 SELECT  S1+1, RAND()                                    ---- ---- ---- ----
 FROM    TEMP1                                              0   65 A    65
 WHERE   S1+1 < 5                                           1   88 X    88
)                                                           2   84 T    84
SELECT SMALLINT(S1) AS SEQ#                                 3   79 O    79
      ,SMALLINT(R1*26+65) AS RAN2                           4   88 X    88
      ,CHR(SMALLINT(R1*26+65)) AS RAN3
      ,CHAR(SMALLINT(R1*26)+65) AS RAN4
FROM   TEMP1;
```

Figure 764, Converting RAND output from number to character

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 four independently random fields. The first has the usual 32K distinct values evenly distributed in the range of zero to one. The second is the same, except that it has many more distinct values (approximately 32K squared). The third and fourth have random numbers that are skewed towards the low end of the range with average values of 0.25 and 0.125 respectively.

```
WITH TEMP1 (S1,R1,R2,R3,R4) AS          ANSWER
(VALUES (0                              ==============================
        ,RAND(2)                        S# RAN1   RAN2   RAN3   RAN4
        ,RAND()+(RAND()/1E5)            -- ------ ------ ------ ------
        ,RAND()* RAND()                  0   1373 169599 182618 215387
        ,RAND()* RAND()* RAND())         1 326700 445273 539604 357592
 UNION ALL                               2 909848 981267   7140  81553
 SELECT  S1 + 1                          3 454573 577320 309318 166436
        ,RAND()                          4 875942 257823 207873   9628
        ,RAND()+(RAND()/1E5)
        ,RAND()* RAND()
        ,RAND()* RAND()* RAND()
 FROM   TEMP1
 WHERE  S1 + 1   < 5
)
SELECT SMALLINT(S1)    AS S#
      ,INTEGER(R1*1E6) AS RAN1,  INTEGER(R2*1E6) AS RAN2
      ,INTEGER(R3*1E6) AS RAN3,  INTEGER(R4*1E6) AS RAN4
FROM   TEMP1;
```

Figure 765, Create RAND data with different distributions

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.

```
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'  ));
COMMIT;

CREATE UNIQUE INDEX PEX2 ON PERSONNEL (SOCSEC#);
CREATE UNIQUE INDEX PEX3 ON PERSONNEL (DEPT, EMP#);
COMMIT;
```

Figure 766, Production-like test table DDL

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.

```
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
```

Figure 767, Production-like test table INSERT (part 1 of 2)

```
       ,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;
```

Figure 768, Production-like test table INSERT (part 2 of 2)

Some sample data follows:

```
 EMP#    SOCSEC#     JOB_ DEPT SALARY    DATE_BN    F_NME     L_NME
 ------  ----------- ---- ---- --------- ---------- --------- ---------
 100000  484-10-9999 WKR    47     13.63 01/01/1979 Ammaef    Mimytmbi
 100001  449-38-9998 SEC    53  35758.87 04/10/1962 Ilojff    Liiiemea
 100002  979-90-9997 WKR     1   8155.23 01/03/1975 Xzacaa    Zytaebma
 100003  580-50-9993 WKR    31  16643.50 02/05/1971 Lpiedd    Pimmeeat
 100004  264-87-9994 WKR    21    962.87 01/01/1979 Wgfacc    Geimteei
 100005  661-84-9995 WKR    19   4648.38 01/02/1977 Wrebbc    Rbiybeet
 100006  554-53-9990 WKR     8    375.42 01/01/1979 Mobaaa    Oiiaiaia
 100007  482-23-9991 SEC    36  23170.09 03/07/1968 Emjgdd    Mimtmamb
 100008  536-41-9992 WKR     6  10514.11 02/03/1974 Jnbcaa    Nieebayt
```

Figure 769, Production-like test table, Sample Output

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 page 284).
- 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.

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).

```
CREATE TABLE TIME_SERIES
(KYY       CHAR(03)      NOT NULL
,BGN_DT    DATE          NOT NULL
,END_DT    DATE          NOT NULL
,CONSTRAINT TSX1 PRIMARY KEY(KYY,BGN_DT)
,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');
```

Figure 770, Sample Table DDL - Time Series

Find Overlapping Rows

We want to find any cases where the begin-to-end date range of one row overlaps another with the same key value. In our test database, this query will return no rows.

The following diagram illustrates what we are trying to find. The row at the top (shown as a bold line) is overlapped by each of the four lower rows, but the nature of the overlap differs in each case.

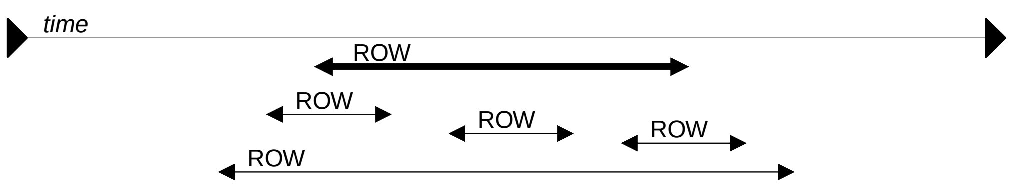

Figure 771, Overlapping Time-Series rows - Definition

> WARNING: When writing SQL to check overlapping data ranges, make sure that all possible types of overlap (see diagram above) are tested. Some simpler SQL statements work with some flavors of overlap, but not others.

The relevant SQL follows. When reading it, think of the "A" table as being the double line above and "B" table as being the four overlapping rows shown as single lines.

```
SELECT KYY                                                      ANSWER
      ,BGN_DT                                                   =========
      ,END_DT                                                   <no rows>
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;
```

Figure 772, Find overlapping rows in time-series

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 page 286. 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 Gaps in Time-Series

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.

```
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

Figure 773, Find gap in Time-Series, SQL

KEYCOL	BGN_DT	END_DT	BGN_DT	END_DT	DIFF
AAA	10/01/1995	10/04/1995	10/06/1995	10/06/1995	2
AAA	10/07/1995	10/07/1995	10/15/1995	10/19/1995	8
BBB	10/01/1995	10/01/1995	10/03/1995	10/03/1995	2

Figure 774, Find gap in Time-Series, Answer

WARNING: 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.

```
SELECT A.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 GAP_SIZE
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

Figure 775, Find gap in Time-Series, SQL

```
KEYCOL  BGN_GAP     END_GAP     GAP_SIZE
------  ----------  ----------  --------
AAA     10/05/1995  10/05/1995         1
AAA     10/08/1995  10/14/1995         7
BBB     10/02/1995  10/02/1995         1
```

Figure 776, Find gap in Time-Series, Answer

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.

```
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;
```

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 GAP_DT     GSIZE
------ ---------- -----
AAA    10/05/1995     1
AAA    10/08/1995     7
AAA    10/09/1995     6
AAA    10/10/1995     5
AAA    10/11/1995     4
AAA    10/12/1995     3
AAA    10/13/1995     2
AAA    10/14/1995     1
BBB    10/02/1995     1
```

Figure 777, Show each day in Time-Series gap

Retaining a Record

In this section, we are going to look at 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.

  NOTE: The key sample code needed to illustrate the above concepts will be described below. A more complete example is available from my website.

Recording Changes

Below is a very simple table that records relevant customer data:

```
CREATE TABLE customer
(cust#        INTEGER       NOT NULL
,cust_name    CHAR(10)
,cust_mgr     CHAR(10)
,PRIMARY KEY(cust#));
```
Figure 778, Customer table

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:

```
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);
```
Figure 779, Customer-history table

NOTE: The secondary index shown above will make the following view processing, which looks for a row that replaces the current, much more efficient.

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.

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:

```
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);
```
Figure 780, Insert trigger

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:

```
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#));
```
Figure 781, Update trigger

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:

```
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#));
```

Figure 782, Delete trigger

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:

```
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));
```

Figure 783, Profile table

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:

```
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);
```

Figure 784, View of Customer history

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.

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.

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.

Version Table

The following table has one row per version created:

```
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));
```
Figure 785, Version table

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.

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):

```
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));
```

Figure 786, Profile table

Customer (data) Table

Below is a typical data table. This one holds customer data:

```
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);
```

Figure 787, Customer table

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).

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:

```
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)));
```
Figure 788, Customer view - 1 of 2

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:

```
CREATE VIEW customer AS
SELECT  cust#
       ,cust_name
       ,cust_mgr
FROM    customer_vw;
```
Figure 789, Customer view - 2 of 2

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.

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:

```
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);
```

Figure 790, Insert trigger

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.

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:

```
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#
```

Figure 791, Update trigger, part 1 of 2

```
    ,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);
```

Figure 792, Update trigger, part 2 of 2

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.

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:

```
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);
```

Figure 793, Delete trigger

In 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.

Sample Code

See my website for more detailed sample code using the above application.

Other Fun Things

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:

```
WITH TEMP1 (C1) AS                      ANSWER (numbers shortened)
(VALUES '123  ',' 345 ','  567')        =================================
SELECT C1                               C1    DBL         DEC   SML  INT
      ,DOUBLE(C1)    AS DBL             ----- ----------- ----- ---- ----
      ,DECIMAL(C1,3) AS DEC             123    +1.2300E+2  123.  123  123
      ,SMALLINT(C1)  AS SML              345   +3.4500E+2  345.  345  345
      ,INTEGER(C1)   AS INT               567  +5.6700E+2  567.  567  567
FROM   TEMP1;
```

Figure 794, Covert Character to Numeric - SQL

Not all numeric functions support all character representations of a number. The following table illustrates what's allowed and what's not:

```
INPUT STRING    COMPATIBLE FUNCTIONS
============    ==========================================
"      1234"    DOUBLE, DECIMAL, INTEGER, SMALLINT, BIGINT
"      12.4"    DOUBLE, DECIMAL
"      12E4"    DOUBLE
```

Figure 795, Acceptable conversion values

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:

```
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
                                                        -    -     -
```

Figure 796, Checking for non-digits

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:

```
--#SET DELIMITER !                                      IMPORTANT
                                                        ============
CREATE FUNCTION isnumeric(instr VARCHAR(40))            This example
RETURNS CHAR(1)                                         uses an "!"
BEGIN ATOMIC                                            as the stmt
   DECLARE is_number CHAR(1)  DEFAULT 'Y';              delimiter.
   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;
```

Figure 797, Check Numeric function, part 1 of 2

```
      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          -
-       -          -
```

Figure 798, Check Numeric function, part 2 of 2

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:

```
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
```

Figure 799, CHAR and DIGITS function usage

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:

```
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);
```

Figure 800, User-defined functions - convert integer to character

Each of the above functions works the same way:

- 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:

```
SELECT   i_sal                                              ANSWER
        ,CHAR_RIGHT(i_sal) AS i_chr                         ===========
FROM    (SELECT  SMALLINT(salary - 11000) AS i_sal          I_SAL I_CHR
         FROM    staff                                      ----- -----
         WHERE   salary > 10000                              -494  -494
           AND   salary < 12200                               -12   -12
        )AS xxx                                               508   508
ORDER BY i_sal;                                              1009  1009
```

Figure 801, Convert SMALLINT to CHAR

Decimal Input

Creating a similar function to handle decimal input is a little more complex. One problem is that the CHAR function adds zeros to decimal data, which we don't want. But a more serious problem is that there are many sizes and scales of decimal input, but we can only make one function (with a given name) that must support all possible lengths and scales. This is impossible, so we will have to comprise as best we can.

Imagine that we have two decimal fields, one of which has a length and scale of (31,0), while the other has a length and scale of (31,31). We cannot create a single function that will handle both input types without either possibly running out of digits (in the first case), or loosing some precision (in the second case).

> NOTE: 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 all the above, below is a function that converts decimal data to character. It compromises by assuming an input of type decimal(31,12), which should work in most situations:

```
CREATE FUNCTION CHAR_RIGHT(inval DECIMAL(31,12))
RETURNS CHAR(33)
RETURN  CHAR_RIGHT(BIGINT(inval))
CONCAT  '.'
CONCAT  SUBSTR(DIGITS(inval - TRUNCATE(inval,0)),20,12);
```

Figure 802, User-defined functions - covert decimal to character

The function works as follows:

- First, convert the input number to integer using the standard BIGINT function.
- Next, use the previously defined CHAR_RIGHT user-function to convert the BIGINT data to a right-justified character value.
- Then, add a period (dot) to the back of the output.
- Finally append the digits (converted to character using the standard DIGITS function) that represent the decimal component of the input.

Below is the function in action:

```
SELECT   d_sal
        ,CHAR_RIGHT(d_sal)   AS d_chr
FROM    (SELECT  DEC(salary - 11000,6,2)  AS d_sal
         FROM    staff
         WHERE   salary > 10000                                ANSWER
           AND   salary < 12200                    =========================
        )AS xxx                                    D_SAL   D_CHR
ORDER BY d_sal;                                    ------- -----------------
                                                   -494.10 -494.100000000000
                                                    -12.00  -12.000000000000
                                                    508.60  508.600000000000
                                                   1009.75 1009.750000000000
```

Figure 803, Convert DECIMAL to CHAR

Floating point data can be processed using the above function, as long as it is first converted to decimal using the standard DECIMAL function.

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:

```
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;
```

Figure 804, Covert Timestamp to number

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:

```
SELECT   EMPNO
        ,FIRSTNME
        ,LASTNAME
        ,JOB
FROM     EMPLOYEE
WHERE    EMPNO < '000100'
ORDER BY EMPNO;
```

Figure 805, Sample query with no column control

Here is the same SQL statement with each character column being checked against a host-variable. If the host-variable is 1, the data is returned, otherwise a zero-length string:

```
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;
```

Figure 806, Sample query with column control

Making Charts Using SQL

Imagine that one had a string of numbers that one wanted to display as a line-bar char. With a little coding, this is easy to do in SQL:

```
WITH TEMP1 (COL1) AS (VALUES 12, 22, 33, 16, 0, 44, 15, 15)
SELECT COL1
      ,SUBSTR(TRANSLATE(CHAR(' ',50),'*',' '),1,COL1)
       AS PRETTY_CHART
FROM   TEMP1;
```

Figure 807, Make chart using SQL

```
COL1   PRETTY_CHART
----   --------------------------------------------------
  12   ************
  22   **********************
  33   *********************************
  16   ****************
   0
  44   ********************************************
  15   ***************
  15   ***************
```

Figure 808, Make charts using SQL, Answer

To create the above graph we first defined a fifty-byte character field. The TRANSLATE function was then used to convert all blanks in this field to asterisks. Lastly, the field was cut down to size using the SUBSTR function.

A CASE statement should be used in those situations where one is not sure what will be highest value returned from the value being charted. This is needed because DB2 will return a SQL error if a SUBSTR truncation-end value is greater than the related column length.

```
WITH TEMP1 (COL1) AS (VALUES 12, 22, 33, 16, 0, 66, 15, 15)
SELECT COL1
      ,CASE
         WHEN COL1 < 48
         THEN SUBSTR(TRANSLATE(CHAR(' ',50),'*',' '),1,COL1)
         ELSE TRANSLATE(CHAR(' ',47),'*',' ')||'>>>'
       END AS PRETTY_CHART
FROM   TEMP1;
```

Figure 809, Make charts using SQL

```
COL1   PRETTY_CHART
----   --------------------------------------------------
  12   ************
  22   **********************
  33   *********************************
  16   ****************
   0
  66   ***********************************************>>>
  15   ***************
  15   ***************
```
Figure 810, Make charts using SQL, Answer

If the above SQL statement looks a bit intimidating, refer to the description of the SUBSTR function given on page 138 for a simpler illustration of the same general process.

Multiple Counts in One Pass

The STATS table that is defined on page 116 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:

```
SELECT   SEX                                        ANSWER >>    SEX NUM
        ,COUNT(*) AS NUM                                          --- ---
FROM     STATS                                                      F 595
GROUP BY SEX                                                        M 405
ORDER BY SEX;
```
Figure 811, Use GROUP BY to get counts

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.

```
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;
```
Figure 812, Use Common Table Expression to get counts

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:

```
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;
```
Figure 813, Use CASE and SUM to get counts

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.

```
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;
```
Figure 814, Use CASE and SUM to get counts

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:

```
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;
```

Figure 815, Multiple counts in one pass, SQL

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
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
```

Figure 816, Multiple counts in one pass, Answer

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:

```
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
```

Figure 817, Count staff joined per year

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:

```
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
```

Figure 818, Count staff joined per year, all years

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:

```
WITH LIST_YEARS (YEAR#) AS                                ANSWER
  (VALUES  SMALLINT(0)                                    ============
   UNION   ALL                                            YEARS #STAFF
   SELECT  YEAR# + 1                                      ----- ------
   FROM    LIST_YEARS                                         0      0
   WHERE   YEAR# < 5)                                         1      1
SELECT   YEAR#             AS YEARS                           2      0
        ,COALESCE(#STFF,0) AS #STAFF                          3      0
FROM     LIST_YEARS                                           4      2
LEFT OUTER JOIN                                               5      3
        (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;
```

Figure 819, Count staff joined per year, all years

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:

```
WITH LIST_YEARS (YEAR#) AS                                      ANSWER
  (VALUES  SMALLINT(0)                                          ======
   UNION   ALL                                                   YEAR#
   SELECT  YEAR# + 1                                             -----
   FROM    LIST_YEARS                                                0
   WHERE   YEAR# < 5)                                                2
SELECT   YEAR#                                                       3
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;
```

Figure 820, List years when no staff joined

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:

```
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;
```

Figure 821, Break text into words - SQL

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.

```
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.
```

Figure 822, Break text into words - Answer

Denormalize Normalized Data

In the next example, we shall use recursion to string together all of the employee NAME fields in the STAFF table (by department):

```
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;
```

Figure 823, Denormalize Normalized Data - SQL

The above statement 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 we progress, we store the current name in the temporary NAME field, maintain a count of names added, and append the same to the end of the ALL_NAMES field. Once we have all of the names, the final SELECT eliminates from the answer-set all rows, except the last for each department.

```
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
```

Figure 824, Denormalize Normalized Data - Answer

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:

```
CREATE FUNCTION list_names(indept SMALLINT)                 IMPORTANT
RETURNS VARCHAR(50)                                          ============
BEGIN ATOMIC                                                 This example
   DECLARE outstr VARCHAR(50) DEFAULT '';                    uses an "!"
   FOR list_names AS                                         as the stmt
      SELECT   name                                          delimiter.
      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!
```
Figure 825, Creating a function to denormalize names

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.

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.

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 124.45 might be 54.123, or 543.12.

Trailing blanks in character values are a similar problem. Obviously, the reverse of "ABC" is "CBA", but what is the reverse of "ABC "? There is no specific 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:

```
--#SET DELIMITER !                                          IMPORTANT
                                                            ============
CREATE FUNCTION reverse(instr VARCHAR(50))                  This example
RETURNS VARCHAR(50)                                         uses an "!"
BEGIN ATOMIC                                                as the stmt
   DECLARE outstr  VARCHAR(50) DEFAULT '';                  delimiter.
   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!
                                                    ANSWER
SELECT   id              AS ID                      ====================
        ,name            AS NAME1                   ID NAME1    NAME2
        ,reverse(name)   AS NAME2                   -- -------- -------
FROM     staff                                      10 Sanders  srednaS
WHERE    id < 40                                    20 Pernal   lanreP
ORDER BY id!                                        30 Marenghi ihgneraM
```

Figure 826, Reversing character field

The same function can be used to reverse numeric values, as long as they are positive:

```
SELECT   id                                AS ID
        ,salary                            AS SALARY1
        ,DEC(reverse(CHAR(salary)),7,4)    AS SALARY2
FROM     staff                                                   ANSWER
WHERE    id < 40                                     ===================
ORDER BY id;                                         ID SALARY1  SALARY2
                                                     -- -------- -------
                                                     10 18357.50  5.7538
                                                     20 18171.25 52.1718
                                                     30 17506.75 57.6057
```

Figure 827, Reversing numeric field

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.

Stripping Characters

If all you want to do is remove leading and trailing blanks, the LTRIM and RTRIM functions can be combined to do the job:

```
WITH TEMP (TXT) AS                                  ANSWER
(VALUES ('   HAS LEADING BLANKS')                   =======================
       ,('HAS TRAILING BLANKS  ')                   TXT2                LEN
       ,(' BLANKS BOTH ENDS    '))                  ------------------- ---
SELECT  LTRIM(RTRIM(TXT))          AS TXT2          HAS LEADING BLANKS   18
       ,LENGTH(LTRIM(RTRIM(TXT)))  AS LEN           HAS TRAILING BLANKS  19
FROM    TEMP;                                       BLANKS BOTH ENDS     16
```

Figure 828, Stripping leading and trailing blanks

Writing Your Own STRIP Function

Stripping leading and trailing non-blank characters is a little harder, and is best done by writing your own function. The following example goes thus:

- Check that a one-byte strip value was provided. Signal an error if not.
- Starting from the left, scan the input string one byte at a time, looking for the character to be stripped. Stop scanning when something else is found.
- Use the SUBSTR function to trim the input-string - up to the first non-target value found.

- Starting from the right, scan the left-stripped input string one byte at a time, looking for the character to be stripped. Stop scanning when something else is found.
- Use the SUBSTR function to trim the right side of the already left-trimmed input string.
- Return the result.

Here is the code:

```
--#SET DELIMITER !

CREATE FUNCTION strip(in_val VARCHAR(20),in_strip VARCHAR(1))
RETURNS VARCHAR(20)
BEGIN ATOMIC
   DECLARE cur_pos SMALLINT;
   DECLARE stp_flg CHAR(1);
   DECLARE out_val VARCHAR(20);
   IF in_strip = '' THEN
      SIGNAL SQLSTATE '75001'
      SET MESSAGE_TEXT = 'Strip char is zero length';
   END IF;
   SET cur_pos = 1;
   SET stp_flg = 'Y';
   WHILE stp_flg = 'Y' AND cur_pos <= length(in_val) DO
      IF SUBSTR(in_val,cur_pos,1) <> in_strip THEN
         SET stp_flg = 'N';
      ELSE
         SET cur_pos = cur_pos + 1;
      END IF;
   END WHILE;
   SET out_val = SUBSTR(in_val,cur_pos);
   SET cur_pos = length(out_val);
   SET stp_flg = 'Y';
   WHILE stp_flg = 'Y' AND cur_pos >= 1 DO
      IF SUBSTR(out_val,cur_pos,1) <> in_strip THEN
         SET stp_flg = 'N';
      ELSE
         SET cur_pos = cur_pos - 1;                          IMPORTANT
      END IF;                                                ============
   END WHILE;                                                This example
   SET out_val = SUBSTR(out_val,1,cur_pos);                  uses an "!"
   RETURN out_val;                                           as the stmt
END!                                                         delimiter.
```

Figure 829, Define strip function

Here is the above function in action:

```
WITH word1 (w#, word_val) AS                    ANSWER
   (VALUES(1,'00 abc 000')                      ========================
         ,(2,'0 0 abc')                         W# WORD_VAL   STP    LEN
         ,(3,' sdbs')                           -- ---------- ------ ---
         ,(4,'000 0')                            1 00 abc 000  abc     5
         ,(5,'0000')                             2 0 0 abc     0 abc   6
         ,(6,'0')                                3  sdbs       sdbs    5
         ,(7,'a')                                4 000 0               1
         ,(8,''))                                5 0000                0
SELECT   w#                                      6 0                   0
        ,word_val                                7 a          a        1
        ,strip(word_val,'0')         AS stp      8                     0
        ,length(strip(word_val,'0')) AS len
FROM     word1
ORDER BY w#;
```

Figure 830, Use strip function

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:

```
--#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';                                    IMPORTANT
       SET cur_pos =  1;                                     ============
       WHILE cur_pos < length(in_val) DO                     This example
          IF  (UCASE(sort_dir)            = 'A'              uses an "!"
          AND  SUBSTR(out_val,cur_pos+1,1) <                 as the stmt
SUBSTR(out_val,cur_pos,1))                                   delimiter.
          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!
```

Figure 831, Define sort-char function

Here is the function in action:

```
WITH word1 (w#, word_val) AS                    ANSWER
   (VALUES(1,'12345678')                        =============================
         ,(2,'ABCDEFG')                         W# WORD_VAL SA       SD
         ,(3,'AaBbCc')                          -- --------- ------- --------
         ,(4,'abccb')                            1 12345678 12345678 87654321
         ,(5,'''%#.')                            2 ABCDEFG  ABCDEFG  GFEDCBA
         ,(6,'bB')                               3 AaBbCc   aAbBcC   CcBbAa
         ,(7,'a')                                4 abccb    abbcc    ccbba
         ,(8,''))                                5 '%#.     .'#%     %#'.
SELECT   w#                                      6 bB       bB       Bb
        ,word_val                                7 a        a        a
        ,sort_char(word_val,'a') sa              8
        ,sort_char(word_val,'D') sd
FROM     word1
ORDER BY w#;
```

Figure 832, Use sort-char function

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:

```
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
```

Figure 833, Run query for four seconds

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.

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.

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:

```
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                           ANSWER
        ,median_row_num  mr                           ==============
WHERE    nn.job       = mr.job                        JOB   MED_SAL
  AND    nn.row# BETWEEN mr.med_lo AND mr.med_hi      ----- --------
GROUP BY nn.job                                       Clerk 13030.50
ORDER BY nn.job;                                      Sales 17432.10
```

Figure 834, Calculating the median

> IMPORTANT: 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:

```
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                ANSWER
WHERE    nn.job        = mr.job            ==========================
  AND    nn.row# BETWEEN mr.med_lo         JOB   MED_SAL  AVG_SAL  #R
                     AND mr.med_hi         ----- -------- -------- --
GROUP BY nn.job                            Clerk 13030.50 12857.56  7
ORDER BY nn.job;                           Sales 17432.10 17460.93  4
```

Figure 835, Get median plus average

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, and the other with four. In this case, when there are 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.

```
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                               ANSWER
FROM     numbered_rows   nn                                 ==============
        ,median_row_num  mr                                 JOB   MED_SAL
WHERE    nn.job  = mr.job                                   ----- --------
  AND    nn.row# = mr.med_row#                              Clerk 13030.50
ORDER BY nn.job;                                            Sales 16858.20
```

Figure 836, Calculating the median

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:

```
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                                        ANSWER
        (SELECT   job                                        ==============
                 ,MAX(row# + 1) / 2                          JOB   MED_SAL
         FROM     numbered_rows                              ----- --------
         GROUP BY job)                                       Clerk 13030.50
ORDER BY job;                                                Sales 16858.20
```

Figure 837, Calculating the median

The next query lists every matching row in the STAFF table (per JOB), and on each line of output, shows the median salary:

```
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;
```

Figure 838, List matching rows and median

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.

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:

```
WITH TEMP1 (C1,T1,T2) AS (VALUES                                   ANSWER
   ('A'                                                            =========
   ,TIMESTAMP('1996-05-01-24.00.00.000000')                        <no rows>
   ,TIMESTAMP('1996-05-02-00.00.00.000000') ))
SELECT C1
FROM   TEMP1
WHERE  T1 = T2;
```

Figure 839, Timestamp comparison - Incorrect

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:

```
WITH TEMP1 (C1,T1,T2) AS (VALUES                                      ANSWER
   ('A'                                                               ======
   ,TIMESTAMP('1996-05-01-24.00.00.000000')                               C1
   ,TIMESTAMP('1996-05-02-00.00.00.000000') ))                            --
SELECT C1                                                                 A
FROM   TEMP1
WHERE  T1 + 0 MICROSECOND = T2 + 0 MICROSECOND;
```

Figure 840, Timestamp comparison - Correct

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.

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:

```
CREATE TABLE SUPERMARKET_SALES
(SALES_TS   TIMESTAMP      NOT NULL
,SALES_VAL  DECIMAL(8,2)   NOT NULL
,PRIMARY KEY(SALES_TS));
```

Figure 841, Sample Table

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 INTO SUPERMARKET_SALES VALUES
('2003-08-01-24.00.00.000000',123.45);
```

Figure 842, Insert row

Now, if we want to select all of the rows that are for a given day, we can write this:

```
SELECT   *
FROM     SUPERMARKET_SALES
WHERE    DATE(SALES_TS) = '2003-08-01'
ORDER BY SALES_TS;
```

Figure 843, Select rows for given date

Or this:

```
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;
```

Figure 844, Select rows for given date

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.

No Rows Match

How many rows to 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:

```
SELECT   CREATOR                                                ANSWER
FROM     SYSIBM.SYSTABLES                                      ========
WHERE    CREATOR = 'ZZZ';                                      <no row>
```

Figure 845, Query with no matching rows (1 of 8)

```
SELECT   MAX(CREATOR)                                            ANSWER
FROM     SYSIBM.SYSTABLES                                        ======
WHERE    CREATOR = 'ZZZ';                                        <null>
```

Figure 846, Query with no matching rows (2 of 8)

```
SELECT   MAX(CREATOR)                                           ANSWER
FROM     SYSIBM.SYSTABLES                                      ========
WHERE    CREATOR = 'ZZZ'                                       <no row>
HAVING   MAX(CREATOR) IS NOT NULL;
```

Figure 847, Query with no matching rows (3 of 8)

```
SELECT   MAX(CREATOR)                                           ANSWER
FROM     SYSIBM.SYSTABLES                                      ========
WHERE    CREATOR      = 'ZZZ'                                  <no row>
HAVING   MAX(CREATOR) = 'ZZZ';
```

Figure 848, Query with no matching rows (4 of 8)

```
SELECT   MAX(CREATOR)                                           ANSWER
FROM     SYSIBM.SYSTABLES                                      ========
WHERE    CREATOR = 'ZZZ'                                       <no row>
GROUP BY CREATOR;
```

Figure 849, Query with no matching rows (5 of 8)

```
SELECT   CREATOR                                                ANSWER
FROM     SYSIBM.SYSTABLES                                      ========
WHERE    CREATOR = 'ZZZ'                                       <no row>
GROUP BY CREATOR;
```

Figure 850, Query with no matching rows (6 of 8)

```
SELECT   COUNT(*)                                               ANSWER
FROM     SYSIBM.SYSTABLES                                      ========
WHERE    CREATOR = 'ZZZ'                                       <no row>
GROUP BY CREATOR;
```

Figure 851, Query with no matching rows (7 of 8)

```
SELECT   COUNT(*)                                                  ANSWER
FROM     SYSIBM.SYSTABLES                                          ======
WHERE    CREATOR = 'ZZZ';                                               0
```

Figure 852, Query with no matching rows (8 of 8)

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:

```
SELECT   COALESCE(NAME,NONAME)  AS NME                       ANSWER
        ,COALESCE(SALARY,NOSAL) AS SAL                       ============
FROM    (SELECT   'NO NAME' AS NONAME                        NME     SAL
                 ,0         AS NOSAL                         ------- ----
         FROM     SYSIBM.SYSDUMMY1                           NO NAME 0.00
        )AS NNN
LEFT OUTER JOIN
        (SELECT   *
         FROM     STAFF
         WHERE    ID  < 5
        )AS XXX
ON       1 = 1
ORDER BY NAME;
```

Figure 853, Always get a row, example 1 of 2

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:

```
WITH NNN (NONAME, NOSAL) AS                                  ANSWER
(VALUES ('NO NAME',0))                                       ============
SELECT   COALESCE(NAME,NONAME)  AS NME                       NME     SAL
        ,COALESCE(SALARY,NOSAL) AS SAL                       ------- ----
FROM     NNN                                                 NO NAME 0.00
LEFT OUTER JOIN
        (SELECT   *
         FROM     STAFF
         WHERE    ID  < 5
        )AS XXX
ON       1 = 1
ORDER BY NAME;
```

Figure 854, Always get a row, example 2 of 2

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:

```
SELECT   DATE('2001-09-22')                                     ANSWER
FROM     SYSIBM.SYSDUMMY1;                                     ==========
                                                               09/22/2001
```

Figure 855, Convert value to DB2 date, right

What happens if you accidentally leave out the quotes in the DATE function? The function still works, but the result is not correct:

```
SELECT   DATE(2001-09-22)                                       ANSWER
FROM     SYSIBM.SYSDUMMY1;                                     ==========
                                                               05/24/0006
```

Figure 856, Convert value to DB2 date, wrong

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.

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:

```
SELECT   ID                                                   ANSWER
        ,NAME                                                 ===========
FROM     STAFF                                                ID NAME
WHERE    ID  <= 100                                           -- --------
  AND    ID   = (INT(RAND()* 10) * 10) + 10                   30 Marenghi
ORDER BY ID;                                                  60 Quigley
```

Figure 857, Get random rows - Incorrect

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:

```
WITH TEMP AS                                          ANSWER
(SELECT   ID                                          ====================
         ,NAME                                        ID  NAME     RAN EQL
         ,(INT(RAND(0)* 10) * 10) + 10 AS RAN         --- -------- --- ---
 FROM     STAFF                                        10 Sanders   10 Y
 WHERE    ID <= 100                                    20 Pernal    30
)                                                      30 Marenghi  70
SELECT   T.*                                           40 O'Brien   10
        ,CASE ID                                       50 Hanes     30
            WHEN RAN THEN 'Y'                          60 Quigley   40
            ELSE          ' '                          70 Rothman   30
         END AS EQL                                    80 James    100
FROM     TEMP T                                        90 Koonitz   40
ORDER BY ID;                                          100 Plotz    100 Y
```

Figure 858, Get random rows - Explanation

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:

```
WITH                                                        ANSWER
STAFF_NUMBERED AS                                           ===========
  (SELECT  S.*                                              ID  NAME
          ,ROW_NUMBER() OVER() AS ROW#                      --- -------
   FROM    STAFF S                                           10 Sanders
   WHERE   ID <= 100                                         20 Pernal
),                                                           90 Koonitz
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;
```

Figure 859, Get random rows - Non-distinct

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:

```
SELECT   ID                                                  ANSWER
        ,NAME                                                ===========
FROM    (SELECT S.*                                          ID NAME
               ,ROW_NUMBER() OVER(ORDER BY RAND()) AS R      -- --------
         FROM   STAFF S                                      10 Sanders
         WHERE  ID <= 100                                    40 O'Brien
        )AS XXX                                              60 Quigley
WHERE    R <= 3
ORDER BY ID;
```

Figure 860, Get random rows - Distinct

In this query, the matching rows are first numbered in random order, and then the three rows with the lowest row number are selected.

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?

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:

```
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
```

Figure 861, Date/Time manipulation - wrong

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:

```
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
```

Figure 862, Date/Time manipulation - right

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.

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, then one has to convert the value into some whole number of a more precise time period. Thus 5.1 days times 82,800 will give one the equivalent number of seconds and 6.2 seconds times 1E6 (i.e. one million) will give one the equivalent number of microseconds.

Use of LIKE 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:

```
WITH TEMP1 (C0,C1,V1) AS (VALUES                                      ANSWER
   ('A',CHAR(' ',1),VARCHAR(' ',1)),                                  ======
   ('B',CHAR(' ',1),VARCHAR('' ,1)))                                      C0
SELECT C0                                                                 --
FROM   TEMP1                                                               A
WHERE  C1 = V1                                                             B
  AND  C1 LIKE ' ';
```

Figure 863, Use LIKE on CHAR field

Look what happens when we change the final predicate from matching on C1 to V1. Now only one row matches our search criteria.

```
WITH TEMP1 (C0,C1,V1) AS (VALUES                                      ANSWER
   ('A',CHAR(' ',1),VARCHAR(' ',1)),                                  ======
   ('B',CHAR(' ',1),VARCHAR('' ,1)))                                      C0
SELECT C0                                                                 --
FROM   TEMP1                                                               A
WHERE  C1 = V1
  AND  V1 LIKE ' ';
```

Figure 864, Use LIKE on VARCHAR field

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:

```
WITH TEMP1 (C0,C1,V1) AS (VALUES                                ANSWER
   ('A',CHAR(' ',1),VARCHAR(' ',1)),                            ======
   ('B',CHAR(' ',1),VARCHAR('' ,1)))                                C0
SELECT C0                                                           --
FROM   TEMP1                                                         A
WHERE  C1 = V1                                                       B
  AND  RTRIM(V1) LIKE '';
```

Figure 865, Use RTRIM to remove trailing blanks

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:

```
WITH TEMP1 (YYMMDD) AS                           ANSWER
(VALUES DATE('2000-01-01')                       ==========================
 UNION  ALL                                      YEAR MIN_DT     MAX_DT
 SELECT YYMMDD + 1 DAY                           ---- ---------- ----------
 FROM   TEMP1                                    2000 2000-08-06 2000-08-12
 WHERE  YYMMDD < '2010-12-31'                    2001 2001-08-12 2001-08-18
)                                                2002 2002-08-11 2002-08-17
SELECT   YY                      AS YEAR         2003 2003-08-10 2003-08-16
        ,CHAR(MIN(YYMMDD),ISO)  AS MIN_DT        2004 2004-08-08 2004-08-14
        ,CHAR(MAX(YYMMDD),ISO)  AS MAX_DT        2005 2005-08-07 2005-08-13
FROM    (SELECT YYMMDD                           2006 2006-08-13 2006-08-19
               ,YEAR(YYMMDD) YY                  2007 2007-08-12 2007-08-18
               ,WEEK(YYMMDD) WK                  2008 2008-08-10 2008-08-16
         FROM   TEMP1                            2009 2009-08-09 2009-08-15
         WHERE  WEEK(YYMMDD) = 33                2010 2010-08-08 2010-08-14
        )AS XXX
GROUP BY YY
        ,WK;
```

Figure 866, Comparing week 33 over 10 years

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:

```
SELECT  SUM(INTEGER(SALARY)) AS S1                            ANSWER
       ,INTEGER(SUM(SALARY)) AS S2                            =============
FROM    STAFF;                                                S1     S2
                                                              ------ ------
                                                              583633 583647
```

Figure 867, DB2 data truncation

If one must do scalar conversions before the column function, use the ROUND function to improve the accuracy of the result:

```
SELECT  SUM(INTEGER(ROUND(SALARY,-1))) AS S1              ANSWER
       ,INTEGER(SUM(SALARY)) AS S2                        =============
FROM    STAFF;                                            S1     S2
                                                          ------ ------
                                                          583640 583647
```

Figure 868, DB2 data rounding

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:

```
SELECT   LASTNAME                                    ANSWER
        ,SEX                                         =================
        ,CASE                                        LASTNAME   SX SXX
           WHEN SEX >= 'F' THEN 'FEM'                ---------- -- ---
           WHEN SEX >= 'M' THEN 'MAL'                JEFFERSON  M  FEM
         END AS SXX                                  JOHNSON    F  FEM
FROM     EMPLOYEE                                    JONES      M  FEM
WHERE    LASTNAME LIKE 'J%'
ORDER BY 1;
```

Figure 869, Case WHEN Processing - Incorrect

By contrast, in the next statement, the SXX value will reflect the related SEX value:

```
SELECT   LASTNAME                                    ANSWER
        ,SEX                                         =================
        ,CASE                                        LASTNAME   SX SXX
           WHEN SEX >= 'M' THEN 'MAL'                ---------- -- ---
           WHEN SEX >= 'F' THEN 'FEM'                JEFFERSON  M  MAL
         END AS SXX                                  JOHNSON    F  FEM
FROM     EMPLOYEE                                    JONES      M  MAL
WHERE    LASTNAME LIKE 'J%'
ORDER BY 1;
```

Figure 870, Case WHEN Processing - Correct

NOTE: See page 32 for more information on this subject.

Division and Average

The following statement gets two results, which is correct?

```
SELECT  AVG(SALARY) / AVG(COMM) AS A1          ANSWER >>>  A1  A2
       ,AVG(SALARY / COMM)      AS A2                      --  -----
FROM    STAFF;                                             32  61.98
```

Figure 871, Division and Average

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).

Date Output Order

DB2 has a bind option (called DATETIME) that specifies the default output format of date-time 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:

```
SELECT   HIREDATE                                              ANSWER
FROM     EMPLOYEE                                              ==========
WHERE    HIREDATE < '1960-01-01'                               05/05/1947
ORDER BY 1;                                                    08/17/1949
                                                               05/16/1958
```
Figure 872, DATE output in year, month, day order

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:

```
SELECT   CHAR(HIREDATE,USA)                                    ANSWER
FROM     EMPLOYEE                                              ==========
WHERE    HIREDATE < '1960-01-01'                               05/05/1947
ORDER BY 1;                                                    05/16/1958
                                                               08/17/1949
```
Figure 873, DATE output in month, day, year order

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.

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.

```
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;
```
Figure 874, Ambiguous Cursor

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.

Floating Point Numbers

The following SQL repetitively multiplies a 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(F1,19)  AS DECIMAL1
      ,BIGINT(F1)  AS BIGINT1
FROM   TEMP;
```

Figure 875, Multiply floating-point number by ten, SQL

After a while, things get interesting:

```
FLOAT1                     DECIMAL1              BIGINT1
------------------------   --------------------  -------------------
  +1.23456789000000E+000                     1.                     1
  +1.23456789000000E+001                    12.                    12
  +1.23456789000000E+002                   123.                   123
  +1.23456789000000E+003                  1234.                  1234
  +1.23456789000000E+004                 12345.                 12345
  +1.23456789000000E+005                123456.                123456
  +1.23456789000000E+006               1234567.               1234567
  +1.23456789000000E+007              12345678.              12345678
  +1.23456789000000E+008             123456789.             123456788
  +1.23456789000000E+009            1234567890.            1234567889
  +1.23456789000000E+010           12345678900.           12345678899
  +1.23456789000000E+011          123456789000.          123456788999
  +1.23456789000000E+012         1234567890000.         1234567889999
  +1.23456789000000E+013        12345678900000.        12345678899999
  +1.23456789000000E+014       123456789000000.       123456788999999
  +1.23456789000000E+015      1234567890000000.      1234567889999999
  +1.23456789000000E+016     12345678900000000.     12345678899999998
  +1.23456789000000E+017    123456789000000000.    123456788999999984
  +1.23456789000000E+018   1234567890000000000.   1234567889999999744
```

Figure 876, Multiply floating-point number by ten, answer

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:

```
WITH TEMP (F1,F2) AS
(VALUES (FLOAT(1.23456789E1 * 10 * 10 * 10 * 10 * 10 * 10 * 10)
        ,FLOAT(1.23456789E8)))
SELECT F1
      ,F2
FROM   TEMP                ANSWER
WHERE  F1 <> F2;           =============================================
                           F1                     F2
                           ---------------------- ----------------------
                           +1.23456789000000E+008 +1.23456789000000E+008
```

Figure 877, Two numbers that look equal, but aren't equal

We can use the HEX function to show that, internally, the two numbers being compared above are not equal:

```
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                         ANSWER
WHERE  F1 <> F2;                    =================================
                                    HEX_F1           HEX_F2
                                    ---------------- ----------------
                                    FFFFFF53346F9D41 00000054346F9D41
```

Figure 878, Two numbers that look equal, but aren't equal, shown in HEX

Now we can explain what is going on in the recursive code shown at the start of this section. The same value is be 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:

```
WITH
 TEMP1 (DEC1, DBL1) AS
   (VALUES (DECIMAL(1),DOUBLE(1)))
,TEMP2 (DEC1, DEC2, DBL1, DBL2) AS
   (SELECT DEC1
          ,DEC1 / 3 AS DEC2
          ,DBL1                       ANSWER (1 row returned)
          ,DBL1 / 3 AS DBL2           ==============================
    FROM   TEMP1)                     DEC1 =  1.0
SELECT *                              DEC2 =  0.33333333333333333333
FROM   TEMP2                          DBL1 =  +1.00000000000000E+000
WHERE  DBL2 <> DEC2;                  DBL2 =  +3.33333333333333E-001
```

Figure 879, Comparing float and decimal division

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:

```
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;
```

Figure 880, Comparing float and decimal multiplication, SQL

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
```

Figure 881, Comparing float and decimal multiplication, answer

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:

```
WITH TEMP (F1) AS               ANSWER
(VALUES FLOAT(0.1))             =======================================
SELECT F1                       F1                     HEX_F1
      ,HEX(F1) AS HEX_F1        ---------------------- ----------------
FROM   TEMP;                    +1.00000000000000E-001 9A9999999999B93F
```

Figure 882, Internal representation of "one tenth" in floating-point

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.

Legally Incorrect SQL

Imagine that we have a cute little view that is defined thus:

```
CREATE VIEW DAMN_LAWYERS (DB2 ,V5) AS
(VALUES  (0001,2)
        ,(1234,2));
```

Figure 883, Sample view definition

Now imagine that we run the following query against this view:

```
SELECT DB2/V5    AS ANSWER                                           ANSWER
FROM   DAMN_LAWYERS;                                                 ------
                                                                          0
                                                                        617
```

Figure 884, Trademark Invalid SQL

Interestingly enough, the above answer is technically correct but, according to IBM, the SQL (actually, they were talking about something else, but it also applies to this SQL) is not quite right. We have been informed (in writing), to quote: "try not to use the slash after 'DB2'. That is an invalid way to use the DB2 trademark - nothing can be attached to 'DB2'." So, as per IBM's trademark requirements, we have changed the SQL thus:

```
SELECT DB2 / V5  AS ANSWER                                           ANSWER
FROM   DAMN_LAWYERS;                                                 ------
                                                                          0
                                                                        617
```

Figure 885, Trademark Valid SQL

Fortunately, we still get the same (correct) answer.

Appendix

DB2 Sample Tables

Class Schedule

```
CREATE TABLE CL_SCHED
(CLASS_CODE          CHARACTER  (00007)
,DAY                 SMALLINT
,STARTING            TIME
,ENDING              TIME);
```

Figure 886, CL_SCHED sample table - DDL

There is no sample data for this table.

Department

```
CREATE TABLE DEPARTMENT
(DEPTNO              CHARACTER  (00003)    NOT NULL
,DEPTNAME            VARCHAR    (00029)    NOT NULL
,MGRNO               CHARACTER  (00006)
,ADMRDEPT            CHARACTER  (00003)    NOT NULL
,LOCATION            CHARACTER  (00016)
,PRIMARY KEY(DEPTNO));
```

Figure 887, DEPARTMENT sample table - DDL

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

Figure 888, DEPARTMENT sample table - Data

Employee

```
CREATE TABLE EMPLOYEE
(EMPNO               CHARACTER  (00006)    NOT NULL
,FIRSTNME            VARCHAR    (00012)    NOT NULL
,MIDINIT             CHARACTER  (00001)    NOT NULL
,LASTNAME            VARCHAR    (00015)    NOT NULL
,WORKDEPT            CHARACTER  (00003)
,PHONENO             CHARACTER  (00004)
,HIREDATE            DATE
,JOB                 CHARACTER  (00008)
,EDLEVEL             SMALLINT              NOT NULL
,SEX                 CHARACTER  (00001)
,BIRTHDATE           DATE
,SALARY              DECIMAL    (09,02)
,BONUS               DECIMAL    (09,02)
,COMM                DECIMAL    (09,02)
,PRIMARY KEY(EMPNO));
```

Figure 889, EMPLOYEE sample table - DDL

EMPNO	FIRSTNME	M	LASTNAME	WKD	HIREDATE	JOB	ED	S	BIRTHDTE	SALRY	BONS	COMM
000010	CHRISTINE	I	HAAS	A00	01/01/1965	PRES	18	F	19330824	52750	1000	4220
000020	MICHAEL	L	THOMPSON	B01	10/10/1973	MANAGER	18	M	19480202	41250	800	3300
000030	SALLY	A	KWAN	C01	04/05/1975	MANAGER	20	F	19410511	38250	800	3060
000050	JOHN	B	GEYER	E01	08/17/1949	MANAGER	16	M	19250915	40175	800	3214
000060	IRVING	F	STERN	D11	09/14/1973	MANAGER	16	M	19450707	32250	500	2580
000070	EVA	D	PULASKI	D21	09/30/1980	MANAGER	16	F	19530526	36170	700	2893
000090	EILEEN	W	HENDERSON	E11	08/15/1970	MANAGER	16	F	19410515	29750	600	2380
000100	THEODORE	Q	SPENSER	E21	06/19/1980	MANAGER	14	M	19561218	26150	500	2092
000110	VINCENZO	G	LUCCHESSI	A00	05/16/1958	SALESREP	19	M	19291105	46500	900	3720
000120	SEAN		O'CONNELL	A00	12/05/1963	CLERK	14	M	19421018	29250	600	2340
000130	DOLORES	M	QUINTANA	C01	07/28/1971	ANALYST	16	F	19250915	23800	500	1904
000140	HEATHER	A	NICHOLLS	C01	12/15/1976	ANALYST	18	F	19460119	28420	600	2274
000150	BRUCE		ADAMSON	D11	02/12/1972	DESIGNER	16	M	19470517	25280	500	2022
000160	ELIZABETH	R	PIANKA	D11	10/11/1977	DESIGNER	17	F	19550412	22250	400	1780
000170	MASATOSHI	J	YOSHIMURA	D11	09/15/1978	DESIGNER	16	M	19510105	24680	500	1974
000180	MARILYN	S	SCOUTTEN	D11	07/07/1973	DESIGNER	17	F	19490221	21340	500	1707
000190	JAMES	H	WALKER	D11	07/26/1974	DESIGNER	16	M	19520625	20450	400	1636
000200	DAVID		BROWN	D11	03/03/1966	DESIGNER	16	M	19410529	27740	600	2217
000210	WILLIAM	T	JONES	D11	04/11/1979	DESIGNER	17	M	19530223	18270	400	1462
000220	JENNIFER	K	LUTZ	D11	08/29/1968	DESIGNER	18	F	19480319	29840	600	2387
000230	JAMES	J	JEFFERSON	D21	11/21/1966	CLERK	14	M	19350530	22180	400	1774
000240	SALVATORE	M	MARINO	D21	12/05/1979	CLERK	17	M	19540331	28760	600	2301
000250	DANIEL	S	SMITH	D21	10/30/1969	CLERK	15	M	19391112	19180	400	1534
000260	SYBIL	P	JOHNSON	D21	09/11/1975	CLERK	16	F	19361005	17250	300	1380
000270	MARIA	L	PEREZ	D21	09/30/1980	CLERK	15	F	19530526	27380	500	2190
000280	ETHEL	R	SCHNEIDER	E11	03/24/1967	OPERATOR	17	F	19360328	26250	500	2100
000290	JOHN	R	PARKER	E11	05/30/1980	OPERATOR	12	M	19460709	15340	300	1227
000300	PHILIP	X	SMITH	E11	06/19/1972	OPERATOR	14	M	19361027	17750	400	1420
000310	MAUDE	F	SETRIGHT	E11	09/12/1964	OPERATOR	12	F	19310421	15900	300	1272
000320	RAMLAL	V	MEHTA	E21	07/07/1965	FIELDREP	16	M	19320811	19950	400	1596
000330	WING		LEE	E21	02/23/1976	FIELDREP	14	M	19410718	25370	500	2030
000340	JASON	R	GOUNOT	E21	05/05/1947	FIELDREP	16	M	19260517	23840	500	1907

Figure 890, EMPLOYEE sample table - Data

Employee Activity

```
CREATE TABLE EMP_ACT
(EMPNO                CHARACTER   (00006)     NOT NULL
,PROJNO               CHARACTER   (00006)     NOT NULL
,ACTNO                SMALLINT                NOT NULL
,EMPTIME              DECIMAL     (05,02)
,EMSTDATE             DATE
,EMENDATE             DATE);
```

Figure 891, EMP_ACT sample table - DDL

EMPNO	PROJNO	ACTNO	EMPTIME	EMSTDATE	EMENDATE
000010	MA2100	10	0.50	01/01/1982	11/01/1982
000010	MA2110	10	1.00	01/01/1982	02/01/1983
000010	AD3100	10	0.50	01/01/1982	07/01/1982
000020	PL2100	30	1.00	01/01/1982	09/15/1982
000030	IF1000	10	0.50	06/01/1982	01/01/1983
000030	IF2000	10	0.50	01/01/1982	01/01/1983
000050	OP1000	10	0.25	01/01/1982	02/01/1983
000050	OP2010	10	0.75	01/01/1982	02/01/1983
000070	AD3110	10	1.00	01/01/1982	02/01/1983
000090	OP1010	10	1.00	01/01/1982	02/01/1983
000100	OP2010	10	1.00	01/01/1982	02/01/1983
000110	MA2100	20	1.00	01/01/1982	03/01/1982
000130	IF1000	90	1.00	01/01/1982	10/01/1982
000130	IF1000	100	0.50	10/01/1982	01/01/1983

Figure 892, EMP_ACT sample table - Data (1 of 2)

EMPNO	PROJNO	ACTNO	EMPTIME	EMSTDATE	EMENDATE
000140	IF1000	90	0.50	10/01/1982	01/01/1983
000140	IF2000	100	1.00	01/01/1982	03/01/1982
000140	IF2000	100	0.50	03/01/1982	07/01/1982
000140	IF2000	110	0.50	03/01/1982	07/01/1982
000140	IF2000	110	0.50	10/01/1982	01/01/1983
000150	MA2112	60	1.00	01/01/1982	07/15/1982
000150	MA2112	180	1.00	07/15/1982	02/01/1983
000160	MA2113	60	1.00	07/15/1982	02/01/1983
000170	MA2112	60	1.00	01/01/1982	06/01/1983
000170	MA2112	70	1.00	06/01/1982	02/01/1983
000170	MA2113	80	1.00	01/01/1982	02/01/1983
000180	MA2113	70	1.00	04/01/1982	06/15/1982
000190	MA2112	70	1.00	02/01/1982	10/01/1982
000190	MA2112	80	1.00	10/01/1982	10/01/1983
000200	MA2111	50	1.00	01/01/1982	06/15/1982
000200	MA2111	60	1.00	06/15/1982	02/01/1983
000210	MA2113	80	0.50	10/01/1982	02/01/1983
000210	MA2113	180	0.50	10/01/1982	02/01/1983
000220	MA2111	40	1.00	01/01/1982	02/01/1983
000230	AD3111	60	1.00	01/01/1982	03/15/1982
000230	AD3111	60	0.50	03/15/1982	04/15/1982
000230	AD3111	70	0.50	03/15/1982	10/15/1982
000230	AD3111	80	0.50	04/15/1982	10/15/1982
000230	AD3111	180	1.00	10/15/1982	01/01/1983
000240	AD3111	70	1.00	02/15/1982	09/15/1982
000240	AD3111	80	1.00	09/15/1982	01/01/1983
000250	AD3112	60	1.00	01/01/1982	02/01/1982
000250	AD3112	60	0.50	02/01/1982	03/15/1982
000250	AD3112	60	0.50	12/01/1982	01/01/1983
000250	AD3112	60	1.00	01/01/1983	02/01/1983
000250	AD3112	70	0.50	02/01/1982	03/15/1982
000250	AD3112	70	1.00	03/15/1982	08/15/1982
000250	AD3112	70	0.25	08/15/1982	10/15/1982
000250	AD3112	80	0.25	08/15/1982	10/15/1982
000250	AD3112	80	0.50	10/15/1982	12/01/1982
000250	AD3112	180	0.50	08/15/1982	01/01/1983
000260	AD3113	70	0.50	06/15/1982	07/01/1982
000260	AD3113	70	1.00	07/01/1982	02/01/1983
000260	AD3113	80	1.00	01/01/1982	03/01/1982
000260	AD3113	80	0.50	03/01/1982	04/15/1982
000260	AD3113	180	0.50	03/01/1982	04/15/1982
000260	AD3113	180	1.00	04/15/1982	06/01/1982
000260	AD3113	180	0.50	06/01/1982	07/01/1982
000270	AD3113	60	0.50	03/01/1982	04/01/1982
000270	AD3113	60	1.00	04/01/1982	09/01/1982
000270	AD3113	60	0.25	09/01/1982	10/15/1982
000270	AD3113	70	0.75	09/01/1982	10/15/1982
000270	AD3113	70	1.00	10/15/1982	02/01/1983
000270	AD3113	80	1.00	01/01/1982	03/01/1982
000270	AD3113	80	0.50	03/01/1982	04/01/1982
000280	OP1010	130	1.00	01/01/1982	02/01/1983
000290	OP1010	130	1.00	01/01/1982	02/01/1983
000300	OP1010	130	1.00	01/01/1982	02/01/1983
000310	OP1010	130	1.00	01/01/1982	02/01/1983
000320	OP2011	140	0.75	01/01/1982	02/01/1983
000320	OP2011	150	0.25	01/01/1982	02/01/1983
000330	OP2012	140	0.25	01/01/1982	02/01/1983
000330	OP2012	160	0.75	01/01/1982	02/01/1983
000340	OP2013	140	0.50	01/01/1982	02/01/1983
000340	OP2013	170	0.50	01/01/1982	02/01/1983

Figure 893, EMP_ACT sample table - Data (2 of 2)

Employee Photo

```
CREATE TABLE EMP_PHOTO
(EMPNO              CHARACTER  (00006)    NOT NULL
,PHOTO_FORMAT       VARCHAR    (00010)    NOT NULL
,PICTURE            BLOB       (0100)K
,PRIMARY KEY(EMPNO,PHOTO_FORMAT));
```

Figure 894, EMP_PHOTO sample table - DDL

```
EMPNO   PHOTO_FORMAT  PICTURE
------  ------------  -------------
000130  bitmap        <<NOT SHOWN>>
000130  gif           <<NOT SHOWN>>
000130  xwd           <<NOT SHOWN>>
000140  bitmap        <<NOT SHOWN>>
000140  gif           <<NOT SHOWN>>
000140  xwd           <<NOT SHOWN>>
000150  bitmap        <<NOT SHOWN>>
000150  gif           <<NOT SHOWN>>
000150  xwd           <<NOT SHOWN>>
000190  bitmap        <<NOT SHOWN>>
000190  gif           <<NOT SHOWN>>
000190  xwd           <<NOT SHOWN>>
```

Figure 895, EMP_PHOTO sample table - Data

Employee Resume

```
CREATE TABLE EMP_RESUME
(EMPNO              CHARACTER  (00006)    NOT NULL
,RESUME_FORMAT      VARCHAR    (00010)    NOT NULL
,RESUME             CLOB       (0005)K
,PRIMARY KEY(EMPNO,RESUME_FORMAT));
```

Figure 896, EMP_RESUME sample table - DDL

```
EMPNO   RESUME_FORMAT  RESUME
------  -------------  -------------
000130  ascii          <<NOT SHOWN>>
000130  script         <<NOT SHOWN>>
000140  ascii          <<NOT SHOWN>>
000140  script         <<NOT SHOWN>>
000150  ascii          <<NOT SHOWN>>
000150  script         <<NOT SHOWN>>
000190  ascii          <<NOT SHOWN>>
000190  script         <<NOT SHOWN>>
```

Figure 897, EMP_RESUME sample table - Data

In Tray

```
CREATE TABLE IN_TRAY
(RECEIVED           TIMESTAMP
,SOURCE             CHARACTER  (00008)
,SUBJECT            CHARACTER  (00064)
,NOTE_TEXT          VARCHAR    (03000));
```

Figure 898, IN_TRAY sample table - DDL

There is no sample data for this table.

Organization

```
CREATE TABLE ORG
(DEPTNUMB           SMALLINT                NOT NULL
,DEPTNAME           VARCHAR    (00014)
,MANAGER            SMALLINT
,DIVISION           VARCHAR    (00010)
,LOCATION           VARCHAR    (00013)
,PRIMARY KEY(DEPTNUMB));
```

Figure 899, ORG sample table - DDL

```
DEPTNUMB  DEPTNAME        MANAGER  DIVISION    LOCATION
--------  --------------  -------  ----------  -------------
      10  Head Office         160  Corporate   New York
      15  New England          50  Eastern     Boston
      20  Mid Atlantic         10  Eastern     Washington
      38  South Atlantic       30  Eastern     Atlanta
      42  Great Lakes         100  Midwest     Chicago
      51  Plains              140  Midwest     Dallas
      66  Pacific             270  Western     San Francisco
      84  Mountain            290  Western     Denver
```

Figure 900, ORG sample table - Data

Project

```
CREATE TABLE PROJECT
(PROJNO             CHARACTER  (00006)     NOT NULL
,PROJNAME           VARCHAR    (00024)     NOT NULL
,DEPTNO             CHARACTER  (00003)     NOT NULL
,RESPEMP            CHARACTER  (00006)     NOT NULL
,PRSTAFF            DECIMAL    (05,02)
,PRSTDATE           DATE
,PRENDATE           DATE
,MAJPROJ            CHARACTER  (00006)
,PRIMARY KEY(PROJNO));
```

Figure 901, PROJECT sample table - DDL

```
PROJNO PROJNAME               DP# RESEMP PRSTAFF PRSTDATE   PRENDATE   MAJPRJ
------ ---------------------- --- ------ ------- ---------- ---------- ------
AD3100 ADMIN SERVICES         D01 000010    6.50 01/01/1982 02/01/1983
AD3110 GENERAL ADMIN SYSTEMS  D21 000070    6.00 01/01/1982 02/01/1983 AD3100
AD3111 PAYROLL PROGRAMMING    D21 000230    2.00 01/01/1982 02/01/1983 AD3110
AD3112 PERSONNEL PROGRAMMING  D21 000250    1.00 01/01/1982 02/01/1983 AD3110
AD3113 ACCOUNT PROGRAMMING    D21 000270    2.00 01/01/1982 02/01/1983 AD3110
IF1000 QUERY SERVICES         C01 000030    2.00 01/01/1982 02/01/1983 -
IF2000 USER EDUCATION         C01 000030    1.00 01/01/1982 02/01/1983 -
MA2100 WELD LINE AUTOMATION   D01 000010   12.00 01/01/1982 02/01/1983 -
MA2110 W L PROGRAMMING        D11 000060    9.00 01/01/1982 02/01/1983 MA2100
MA2111 W L PROGRAM DESIGN     D11 000220    2.00 01/01/1982 12/01/1982 MA2110
MA2112 W L ROBOT DESIGN       D11 000150    3.00 01/01/1982 12/01/1982 MA2110
OP1000 OPERATION SUPPORT      E01 000050    6.00 01/01/1982 02/01/1983 -
OP1010 OPERATION              E11 000090    5.00 01/01/1982 02/01/1983 OP1000
OP2000 GEN SYSTEMS SERVICES   E01 000050    5.00 01/01/1982 02/01/1983 -
MA2113 W L PROD CONT PROGS    D11 000160    3.00 02/15/1982 12/01/1982 MA2110
OP2010 SYSTEMS SUPPORT        E21 000100    4.00 01/01/1982 02/01/1983 OP2000
OP2011 SCP SYSTEMS SUPPORT    E21 000320    1.00 01/01/1982 02/01/1983 OP2010
OP2012 APPLICATIONS SUPPORT   E21 000330    1.00 01/01/1982 02/01/1983 OP2010
OP2013 DB/DC SUPPORT          E21 000340    1.00 01/01/1982 02/01/1983 OP2010
PL2100 WELD LINE PLANNING     B01 000020    1.00 01/01/1982 09/15/1982 MA2100
```

Figure 902, PROJECT sample table - Data

Sales

```
CREATE TABLE SALES
(SALES_DATE          DATE
,SALES_PERSON        VARCHAR     (00015)
,REGION              VARCHAR     (00015)
,SALES               INTEGER);
```

Figure 903, SALES sample table - DDL

```
SALES_DATE SALES_PERSON    REGION          SALES
---------- --------------- --------------- -----
12/31/1995 GOUNOT          Quebec              1
12/31/1995 LEE             Manitoba            2
12/31/1995 LEE             Ontario-South       3
12/31/1995 LEE             Quebec              1
12/31/1995 LUCCHESSI       Ontario-South       1
03/29/1996 GOUNOT          Manitoba            7
03/29/1996 GOUNOT          Ontario-South       3
03/29/1996 GOUNOT          Quebec              1
03/29/1996 LEE             Manitoba            5
03/29/1996 LEE             Ontario-North       2
03/29/1996 LEE             Ontario-South       2
03/29/1996 LEE             Quebec              3
03/29/1996 LUCCHESSI       Ontario-South       3
03/29/1996 LUCCHESSI       Quebec              1
03/30/1996 GOUNOT          Manitoba            1
03/30/1996 GOUNOT          Ontario-South       2
03/30/1996 GOUNOT          Quebec             18
03/30/1996 LEE             Manitoba            4
03/30/1996 LEE             Ontario-North       3
03/30/1996 LEE             Ontario-South       7
03/30/1996 LEE             Quebec              7
03/30/1996 LUCCHESSI       Manitoba            1
03/30/1996 LUCCHESSI       Ontario-South       1
03/30/1996 LUCCHESSI       Quebec              2
03/31/1996 GOUNOT          Ontario-South       2
03/31/1996 GOUNOT          Quebec              1
03/31/1996 LEE             Manitoba            3
03/31/1996 LEE             Ontario-North       3
03/31/1996 LEE             Ontario-South      14
03/31/1996 LEE             Quebec              7
03/31/1996 LUCCHESSI       Manitoba            1
04/01/1996 GOUNOT          Manitoba            7
04/01/1996 GOUNOT          Ontario-North       1
04/01/1996 GOUNOT          Ontario-South       3
04/01/1996 GOUNOT          Quebec              3
04/01/1996 LEE             Manitoba            9
04/01/1996 LEE             Ontario-North       -
04/01/1996 LEE             Ontario-South       8
04/01/1996 LEE             Quebec              8
04/01/1996 LUCCHESSI       Manitoba            1
04/01/1996 LUCCHESSI       Ontario-South       3
```

Figure 904, SALES sample table - Data

Staff

```
CREATE TABLE STAFF
(ID                  SMALLINT                  NOT NULL
,NAME                VARCHAR     (00009)
,DEPT                SMALLINT
,JOB                 CHARACTER   (00005)
,YEARS               SMALLINT
,SALARY              DECIMAL     (07,02)
,COMM                DECIMAL     (07,02)
,PRIMARY KEY(ID));
```

Figure 905, STAFF sample table - DDL

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
10	Sanders	20	Mgr	7	18357.50	-
20	Pernal	20	Sales	8	18171.25	612.45
30	Marenghi	38	Mgr	5	17506.75	-
40	O'Brien	38	Sales	6	18006.00	846.55
50	Hanes	15	Mgr	10	20659.80	-
60	Quigley	38	Sales	-	16808.30	650.25
70	Rothman	15	Sales	7	16502.83	1152.00
80	James	20	Clerk	-	13504.60	128.20
90	Koonitz	42	Sales	6	18001.75	1386.70
100	Plotz	42	Mgr	7	18352.80	-
110	Ngan	15	Clerk	5	12508.20	206.60
120	Naughton	38	Clerk	-	12954.75	180.00
130	Yamaguchi	42	Clerk	6	10505.90	75.60
140	Fraye	51	Mgr	6	21150.00	-
150	Williams	51	Sales	6	19456.50	637.65
160	Molinare	10	Mgr	7	22959.20	-
170	Kermisch	15	Clerk	4	12258.50	110.10
180	Abrahams	38	Clerk	3	12009.75	236.50
190	Sneider	20	Clerk	8	14252.75	126.50
200	Scoutten	42	Clerk	-	11508.60	84.20
210	Lu	10	Mgr	10	20010.00	-
220	Smith	51	Sales	7	17654.50	992.80
230	Lundquist	51	Clerk	3	13369.80	189.65
240	Daniels	10	Mgr	5	19260.25	-
250	Wheeler	51	Clerk	6	14460.00	513.30
260	Jones	10	Mgr	12	21234.00	-
270	Lea	66	Mgr	9	18555.50	-
280	Wilson	66	Sales	9	18674.50	811.50
290	Quill	84	Mgr	10	19818.00	-
300	Davis	84	Sales	5	15454.50	806.10
310	Graham	66	Sales	13	21000.00	200.30
320	Gonzales	66	Sales	4	16858.20	844.00
330	Burke	66	Clerk	1	10988.00	55.50
340	Edwards	84	Sales	7	17844.00	1285.00
350	Gafney	84	Clerk	5	13030.50	188.00

Figure 906, STAFF sample table - Data

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.

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.

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.

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.

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.

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, however, other binding methods that take a little more skill and better gear that can be used to make "store-quality" books. A good reference on the general subject of home publishing is Book-on-Demand Publishing (ISBN 1-881676-02-1) by Rupert Evans. The publisher is BlackLightning Publications Inc. They are on the web (see: www.flashweb.com).

Index

J

L

M

N

O

P