DB2 UDB V7.2

SQL Cookbook

Graeme Birchall

11-Mar-2002

Preface

IBM's DB2 UDB V7.2 comes with a very advanced SQL language that has a lot of powerful features. This book contains numerous examples illustrating how this SQL dialect can be used to answer a variety of every-day business questions.

TRADEMARKS: AIX, DB2, and DB2 MVS, are registered trademarks of the IBM Corporation. Windows, Windows/NT, Windows/2000, Windows/ME, and Word for Windows are registered trademarks of the Microsoft Corporation. Adobe Acrobat is a registered trademark of the Adobe Corporation.

Disclaimer & Copyright

DISCLAIMER: This document is a best effort on my part. However, given the nature of the subject matter covered, it would be extremely unwise to trust the contents in its entirety. I certainly don't. If you do something silly based on what I say, life is tough.

COPYRIGHT: This document may not be copied in whole, or in part, without the explicit written permission of the distributor. Distribution does not imply publication. This copy was printed on 11 March, 2002. It is intended for personal use only. Secondary distribution for gain is not allowed. File name: BOOK08.DOC.

Tools Used

HARDWARE: This book was written on a Dell 400 XPS with 128 MEG of RAM. For printing, I use an HP 4000 double-sided laser printer.

SOFTWARE: DB2 UDB V7.2 Personal Edition, service level WR21270 (i.e. FixPak 4), was used for all testing. Word for Windows was used to write the document. Adobe Acrobat was used to make the PDF file. One thing that all of this software (except DB2 of course) has in common is that it is all bug-ridden junk. I could have written this book in half the time if any of it was even mildly reliable.

Book Binding

A bound book is a lot easier to work with than loose pages, or the same in a three-ring binder. Because few people have access to professional binding machinery, I did some experiments a couple of years ago - binding books using commonly available materials. I came up with what I consider to be a very satisfactory solution that is fully document on page 271.

PRINTING: This book looks best when printed on a double-sided printer.

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 V7.2 SQL Cookbook ©
Print:   11 March, 2002
Fname:   BOOK08.DOC
#pages:  277
```

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, so I decided to tidy everything up and give it away. Over the years, new chapters have been added as DB2 has evolved, and as I have figured out ways to solve new 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.

There are several other reasons why the book is free, as opposed to formally published. I want whatever I put into the public domain to be the best that I can write, and I feel that my current distribution setup results (in this case) in a better quality product than that for a comparable published book. What I lack is an editor and a graphic designer to fix my many errors. But I have a very fast time to print - new editions of this book usually come out within two months of new versions of DB2 becoming available. And corrections and/or enhancements can be included almost immediately. Lastly, 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. The version 6.1 and 5.2 editions of this book are available from my web site. The version 5.0 and V2.1 are also available, via email, upon request.

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. However do not expect to get a prompt response, nor necessarily 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 the DB2 UDB V5 SQL Cookbook was 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 errors. 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 errors 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.

Table of Contents

PREFACE 3
Disclaimer & Copyright 3
Tools Used 3
Book Binding 3
Author / Book 3
Author Notes 4
Book History 4
Why Free 4
Other Free Documents 4
Answering Questions 4
Book Editions 5
Upload Dates 5
TABLE OF CONTENTS 7
INTRODUCTION TO SQL 13
Syntax Diagram Conventions 13
SQL Components 13
DB2 Objects 13
SELECT Statement 15
FETCH FIRST Clause 17
Correlation Name 18
Renaming Fields 19
Working with Nulls 19
SQL Predicates 20
Basic Predicate 20
Quantified Predicate 21
BETWEEN Predicate 22
EXISTS Predicate 22
IN Predicate 22
LIKE Predicate 23
NULL Predicate 24
Precedence Rules 24
Temporary Tables (in Statement) 25
Common Table Expression 26
Full-Select 29
CAST Expression 32
VALUES Clause 33
CASE Expression 35
COLUMN FUNCTIONS 39
Introduction 39
Column Functions, Definitions 39
AVG 39
CORRELATION 41
COUNT 41
COUNT_BIG 42
COVARIANCE 42
GROUPING 43
MAX 43
MIN 44
REGRESSION 44
STDDEV 45
SUM 45
VAR or VARIANCE 46
OLAP FUNCTIONS 47
Introduction 47
The Bad Old Days 47
OLAP Functions, Definitions 50
Ranking Functions 50
Row Numbering Function 56
Aggregation Function 62
SCALAR FUNCTIONS 71
Introduction 71
Sample Data 71

Scalar Functions, Definitions 71
ABS or ABSVAL 71
ACOS 72
ASCII 72
ASIN 72
ATAN 72
ATAN2 72
BIGINT 72
BLOB 73
CEIL or CEILING 73
CHAR 74
CHR 75
CLOB 76
COALESCE 76
CONCAT 77
COS 78
COT 78
DATE 78
DAY 79
DAYNAME 79
DAYOFWEEK 79
DAYOFWEEK_ISO 80
DAYOFYEAR 80
DAYS 80
DBCLOB 81
DEC or DECIMAL 81
DEGREES 81
DEREF 81
DECRYPT_BIN and DECRYPT_CHAR 81
DIFFERENCE 82
DIGITS 82
DLCOMMENT 83
DLLINKTYPE 83
DLURLCOMPLETE 83
DLURLPATH 83
DLURLPATHONLY 83
DLURLSCHEME 83
DLURLSERVER 83
DLVALUE 83
DOUBLE or DOUBLE_PRECISION 83
ENCRYPT 83
EVENT_MON_STATE 84
EXP 84
FLOAT 85
FLOOR 85
GENERATE_UNIQUE 85
GETHINT 86
GRAPHIC 86
HEX 86
HOUR 87
IDENTITY_VAL_LOCAL 87
INSERT 88
INT or INTEGER 88
JULIAN_DAY 88
LCASE or LOWER 90
LEFT 91
LENGTH 91
LN or LOG 92
LOCATE 92
LOG or LN 92
LOG10 92
LONG_VARCHAR 93
LONG_VARGRAPHIC 93
LOWER 93
LTRIM 93
MICROSECOND 93
MIDNIGHT_SECONDS 93
MINUTE 94
MOD 94
MONTH 95
MONTHNAME 95
MULTIPLY_ALT 95
NODENUMBER 96
NULLIF 96
PARTITION 96

POSSTR 97
POWER 97
QUARTER 97
RADIANS 97
RAISE_ERROR 98
RAND 98
REAL 101
REC2XML 102
REPEAT 102
REPLACE 102
RIGHT 103
ROUND 103
RTRIM 103
SECOND 103
SIGN 104
SIN 104
SMALLINT 104
SOUNDEX 104
SPACE 105
SQLCACHE_SNAPSHOT 106
SQRT 106
SUBSTR 107
TABLE 108
TABLE_NAME 108
TABLE_SCHEMA 108
TAN 109
TIME 109
TIMESTAMP 109
TIMESTAMP_ISO 109
TIMESTAMPDIFF 110
TRANSLATE 111
TRUNC or TRUNCATE 112
TYPE_ID 112
TYPE_NAME 112
TYPE_SECHEMA 112
UCASE or UPPER 112
VALUE 113
VARCHAR 113
VARGRAPHIC 113
VEBLOB_CP_LARGE 113
VEBLOB_CP_LARGE 113
WEEK 113
WEEK_ISO 113
YEAR 114
ORDER BY, GROUP BY, AND HAVING 115
Introduction 115
Order By 115
Sample Data 115
Order by Examples 115
Notes 116
Group By and Having 117
GROUP BY Sample Data 117
Simple GROUP BY Statements 117
GROUPING SETS Statement 119
ROLLUP Statement 123
CUBE Statement 127
Complex Grouping Sets - Done Easy 129
Group By and Order By 131
Group By in Join 132
COUNT and No Rows 132
JOINS 133
Why Joins Matter 133
Sample Views 133
Join Syntax 133
Join Types 135
Inner Join 135
Cartesian Product 135
Left Outer Join 137
Right Outer Join 137
Outer Joins 138
Outer Join Notes 139

Using the COALESCE Function 139
ON Predicate Processing 139
WHERE Predicate Processing 140
Listing non-matching rows only 142
Outer Join in SELECT Phrase 144
Predicates and Joins, a lesson 145
Summary 146

SUB-QUERY 147
Sample Tables 147
Sub-query Flavours 147
Sub-query Syntax 147
Correlated vs. Uncorrelated Sub-Queries 154
Multi-Field Sub-Queries 155
Nested Sub-Queries 155
Usage Examples 156
True if NONE Match 156
True if ANY Match 157
True if TEN Match 158
True if ALL match 159

UNION, INTERSECT, AND EXCEPT 161
Syntax Diagram 161
Sample Views 161
Usage Notes 162
Union & Union All 162
Intersect & Intersect All 162
Except & Except All 162
Precedence Rules 163
View Usage 164
Outer Join Usage 164

SUMMARY TABLES 165
Summary Table Types 165
IBM Implementation 166
DDL Restrictions 166
Definition Only Summary Tables 167
Refresh Deferred Summary Tables 167
Refresh Immediate Summary Tables 168
Usage Notes and Restrictions 169
Multi-table Summary Tables 170
Indexes on Summary Tables 172
Roll Your Own 173
Inefficient Triggers 173
Efficient Triggers 176

IDENTITY COLUMN 183
Usage Notes 183
Rules and Restrictions 184
Restart the Identity Start Value 184
Cache Usage 185
Gaps in the Sequence 186
Roll Your Own - no Gaps in Sequence 186
IDENTITY_VAL_LOCAL Function 187

RECURSIVE SQL 191
Use Recursion To 191
When (Not) to Use Recursion 191
How Recursion Works 191
List Dependents of AAA 192
Notes & Restrictions 193
Sample Table DDL & DML 193
Introductory Recursion 194
List all Children #1 194
List all Children #2 194
List Distinct Children 195
Show Item Level 195
Select Certain Levels 196
Select Explicit Level 197
Trace a Path - Use Multiple Recursions 197
Extraneous Warning Message 198

Logical Hierarchy Flavours 199
Divergent Hierarchy........ 199
Convergent Hierarchy 200
Recursive Hierarchy........ 200
Balanced & Unbalanced Hierarchies 201
Data & Pointer Hierarchies........ 201
Halting Recursive Processing........ 202
Sample Database........ 202
Stop After "n" Levels 203
Stop After "n" Levels - Remove Duplicates........ 203
Stop After "n" Levels - Show Data Paths........ 204
Stop After "n" Rows........ 205
Find all Children, Ignore Data Loops 205
Find all Children, Mark Data Loops........ 206
Find all Data Loops - Only........ 206
Stop if Data Loops........ 207
Working with Other Key Types........ 208
Stopping Simple Recursive Statements Using FETCH FIRST code........ 208
Clean Hierarchies and Efficient Joins........ 209
Introduction........ 209
Limited Update Solution 209
Full Update Solution 211
FUN WITH SQL 215
Creating Sample Data 215
Create a Row of Data........ 215
Create "n" Rows & Columns of Data 215
Linear Data Generation 216
Tabular Data Generation........ 216
Cosine vs. Degree - Table of Values 217
Make Reproducible Random Data........ 217
Make Random Data - Different Ranges........ 218
Make Random Data - Different Flavours 218
Make Random Data - Varying Distribution 219
Make Test Table & Data 219
Statistical Analysis using SQL........ 222
Sample Table DDL & DML........ 222
Mean........ 222
Median........ 223
Mode........ 223
Average Deviation........ 223
Variance 223
Standard Deviation........ 224
Data Distribution........ 224
Standard Error........ 225
Pearson Product-Moment Coefficient........ 225
Time-Series Data Usage........ 226
Find Overlapping Rows........ 226
Find Gaps in Time-Series 227
Show Each Day in Gap 229
Other Fun Things 229
Convert Character to Numeric 229
Convert Timestamp to Numeric 231
Selective Column Output 231
Making Charts Using SQL........ 231
Multiple Counts in One Pass........ 232
Multiple Counts from the Same Row 233
Find Missing Rows in Series / Count all Values 234
Normalize Denormalized Data 236
Denormalize Normalized Data 237
Reversing Field Contents........ 238
Stripping Characters........ 240
Query Runs for "n" Seconds 241
QUIRKS IN SQL 243
Trouble with Timestamps........ 243
No Rows Match 243
Dumb Date Usage........ 244
RAND in Predicate 245
Date/Time Manipulation 247
Use of LIKE on VARCHAR 248
Comparing Weeks........ 249
DB2 Truncates, not Rounds........ 249

CASE Checks in Wrong Sequence.....250
Division and Average.....250
Date Output Order.....250
Ambiguous Cursors.....251
Floating Point Numbers.....252
Legally Incorrect SQL.....254

DB2 DISLIKES.....257

Performance Analysis.....257
The DB2 Monitor.....257
Win/NT Problems.....258
Visual Explain.....259

Other Whines.....260
Statement Terminator.....260
Stopping Recursion.....260
RAND Function.....261
SYSFUN Character Functions.....262
STRIP Function.....262
DB2BATCH Reliability.....262

APPENDIX.....263

DB2 Sample Tables.....263
Class Schedule.....263
Department.....263
Employee.....263
Employee Activity.....264
Employee Photo.....266
Employee Resume.....266
In Tray.....266
Organization.....267
Project.....267
Sales.....268
Staff.....268

BOOK BINDING.....271

INDEX.....271

READER COMMENTS.....277

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.

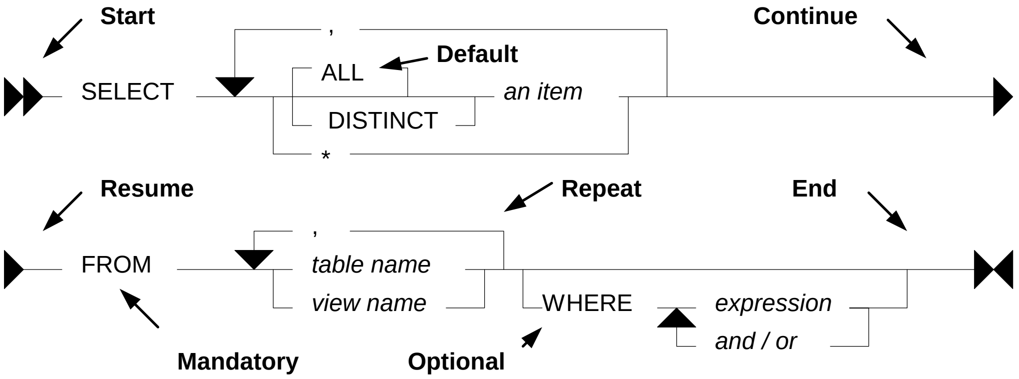


Figure 1, Syntax Diagram Conventions

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.

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 2, 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 3, 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 4, 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
                                                           --  ---  --
                                                           13  AAA  22
                                                           12  BBB  33
                                                           11  CCC   -
```

Figure 5, 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 6, 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 7, 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.

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 26).
- ORDER BY clause. Optionally, order the final output (see page 115).
- FETCH FIRST clause. Optionally, stop the query after "n" rows (see page 17). 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.

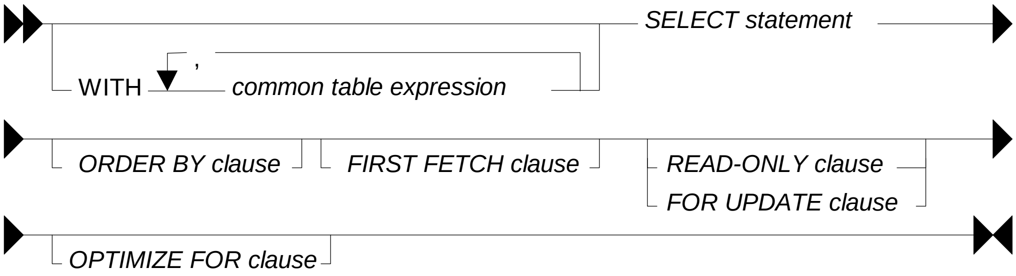


Figure 8, SELECT Statement Syntax (general)

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.

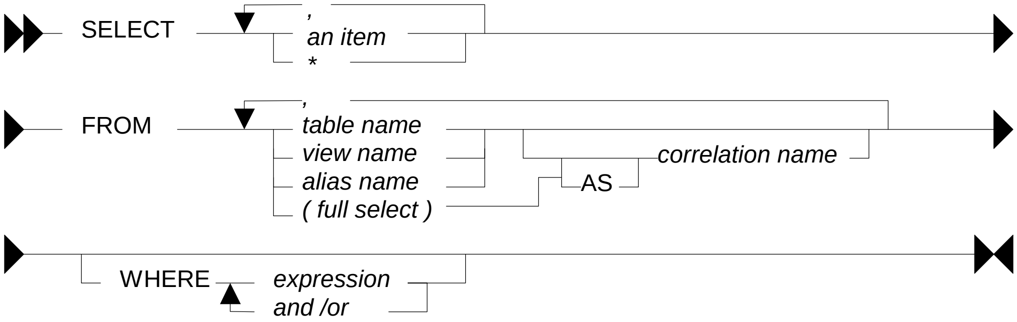


Figure 9, SELECT Statement Syntax

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 10, 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 11, 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 12, 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 13, 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.

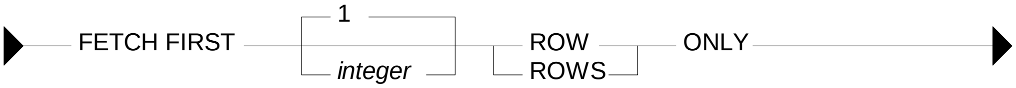


Figure 14, Fetch First clause Syntax

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 15, 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 16, 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 17, 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 58 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 18, 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 19, 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 20, 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 21, 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 22, 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 23, 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 24, AVG of those rows that are not null

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.

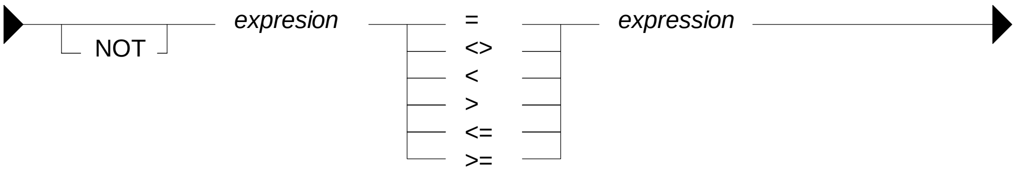


Figure 25, Basic Predicate syntax

```
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 26, Basic Predicate examples

Quantified Predicate

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

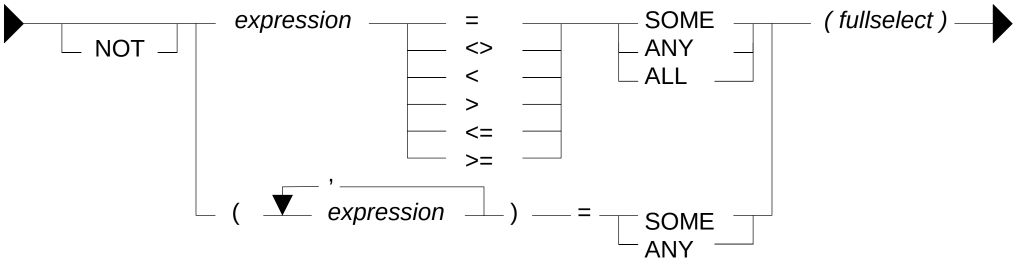


Figure 27, Quantified Predicate syntax,1 of 2

```
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 28, 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 29, Quantified Predicate example, multi-value sub-query

See the sub-query chapter on page 147 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:

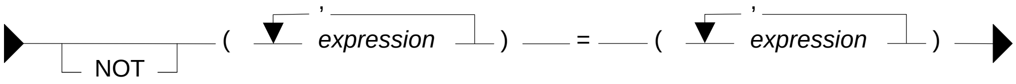


Figure 30, Quantified Predicate syntax, 2 of 2

```
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 31, 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 32, Same query as prior, using individual predicates

BETWEEN Predicate

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

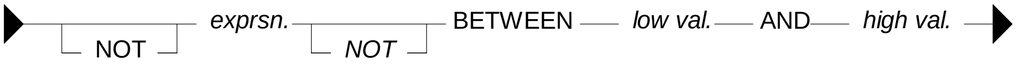


Figure 33, BETWEEN Predicate syntax

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 34, BETWEEN Predicate examples

EXISTS Predicate

An EXISTS predicate tests for the existence of matching rows.

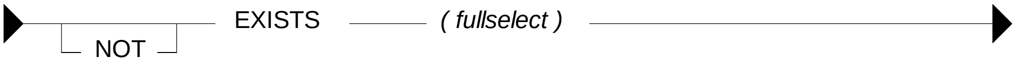


Figure 35, EXISTS Predicate syntax

```
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 36, EXISTS Predicate example

NOTE: See the sub-query chapter on page 147 for more data on this predicate type.

IN Predicate

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

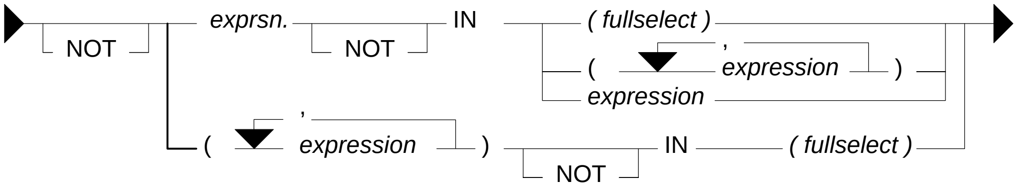


Figure 37, IN Predicate syntax

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 38, 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 147 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 39, IN Predicate example, multi-value

> NOTE: See the sub-query chapter on page 147 for more data on this statement type.

LIKE Predicate

The LIKE predicate does partial checks on character strings.

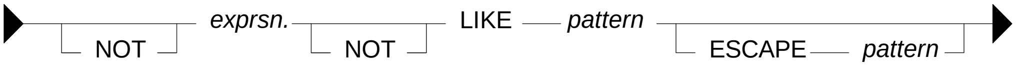


Figure 40, LIKE Predicate syntax

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 41, 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 42, 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 43, 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.

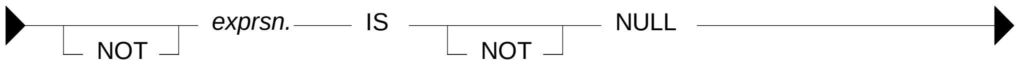


Figure 44, NULL Predicate syntax

```
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 45, 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 46, 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 47, 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 48, 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  |
                                                           |B   |BB  |
SELECT *                                                   |C   |CC  |
FROM   TABLE1                     ANSWER>>    COL1 COL2    +---------+
WHERE  COL1  = 'C'                            ---- ----
  AND (COL1 >= 'A'                               C   CC
   OR  COL2 >= 'AA');
```

Figure 49, 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.

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 50, 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 51, 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 52, 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.

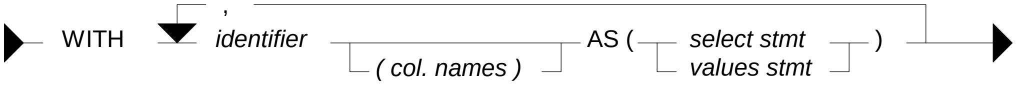


Figure 53, Common Table Expression Syntax

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                                       ANSWER
(SELECT MAX(NAME) AS MAX_NAME                       ==================
       ,MAX(DEPT) AS MAX_DEPT                       MAX_NAME  MAX_DEPT
 FROM   STAFF                                       --------- --------
)                                                   Yamaguchi       84
SELECT *
FROM   TEMP1;
```

Figure 54, 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                   ANSWER
(SELECT MAX(NAME)                                   ==================
       ,MAX(DEPT)                                   MAX_NAME  MAX_DEPT
 FROM   STAFF                                       --------- --------
)                                                   Yamaguchi       84
SELECT *
FROM   TEMP1;
```

Figure 55, 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 56, 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 57, 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 58, 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 59, 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 60, 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 61, 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 62, 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 63, 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 64, 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 65, 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 66, 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 67, 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 68, 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 69, 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 70, 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 71, 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.

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.

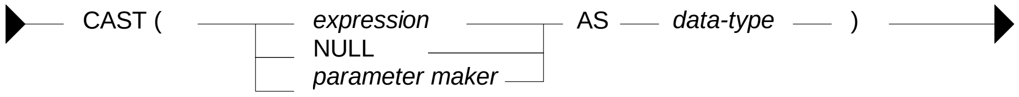


Figure 72, CAST expression syntax

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 73, 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 74, 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 75, Use CAST expression to define SMALLINT field with null values

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.

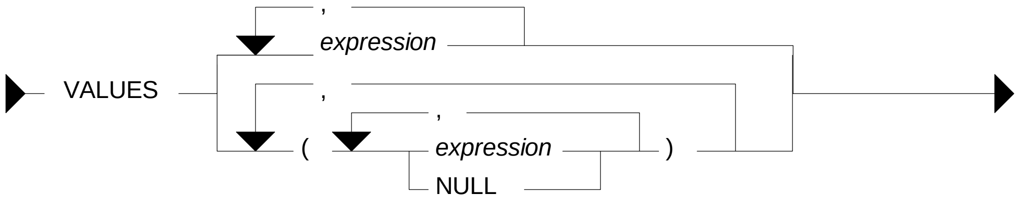


Figure 76, VALUES expression syntax

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 77, 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 78, 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 79, 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 80, 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 81, 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 82, 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 83, 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 84, Use VALUES defined data to seed a recursive SQL statement

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 flavours 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 flavours, the first WHEN that matches is the one chosen.

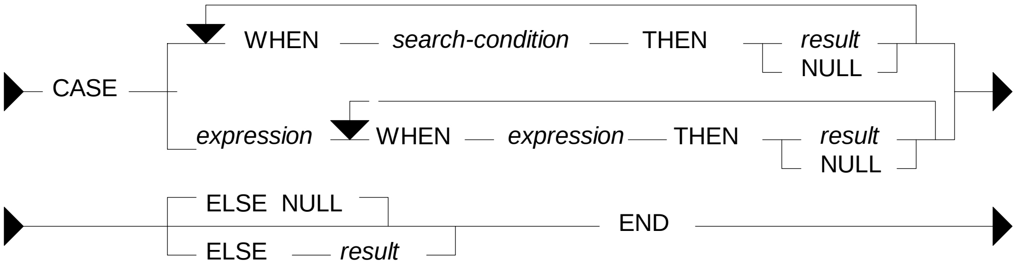


Figure 85, CASE expression syntax

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 86, 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 87, 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 88, 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 89, 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 90, 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 91, 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 92, 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 93, 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 94, 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 95, 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                                    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 96, 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.

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.

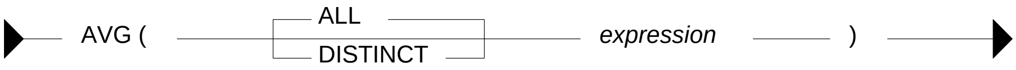


Figure 97, AVG function syntax

```
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 98, 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:

```
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;
```

Figure 99, 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 100, 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 a 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 101, 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 102, 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.

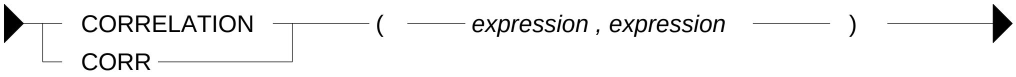


Figure 103, CORRELATION function syntax

```
WITH TEMP1(COL1, COL2, COL3, COL4) AS      ANSWER
(VALUES (0                                 ===========================
        ,0                                 COR11  COR12  COR23  COR34
        ,0                                 ------ ------ ------ ------
        ,RAND(1))                           1.000 -1.000 -0.017 -0.005
 UNION ALL
 SELECT COL1 + 1
       ,COL2 - 1
       ,RAND()
       ,RAND()
 FROM   TEMP1
 WHERE  COL1 <= 1000
)
SELECT DEC(CORRELATION(COL1,COL1),5,3)  AS COR11
      ,DEC(CORRELATION(COL1,COL2),5,3)  AS COR12
      ,DEC(CORRELATION(COL2,COL3),5,3)  AS COR23
      ,DEC(CORRELATION(COL3,COL4),5,3)  AS COR34
FROM   TEMP1;
```

Figure 104, 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.

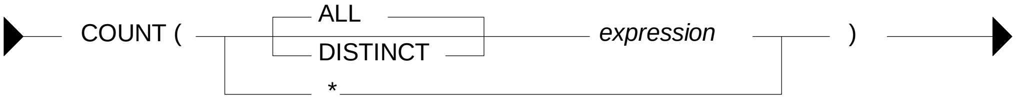


Figure 105, COUNT function syntax

```
SELECT COUNT(*)                     AS C1            ANSWER
      ,COUNT(INT(COMM/10))          AS C2            =================
      ,COUNT(ALL INT(COMM/10))      AS C3            C1 C2 C3 C4 C5 C6
      ,COUNT(DISTINCT INT(COMM/10)) AS C4            -- -- -- -- -- --
      ,COUNT(DISTINCT INT(COMM))    AS C5            35 24 24 19 24  2
      ,COUNT(DISTINCT INT(COMM))/10 AS C6
FROM   STAFF;
```

Figure 106, 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                                ANSWER
        ,COUNT(*)    AS C2                                ============
FROM     STAFF                                            C1        C2
WHERE    ID = -1                                          --------  --
UNION                                                     NO GP-BY   0
SELECT   'GROUP-BY'  AS C1
        ,COUNT(*)    AS C2
FROM     STAFF
WHERE    ID = -1
GROUP BY DEPT;
```

Figure 107, 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.

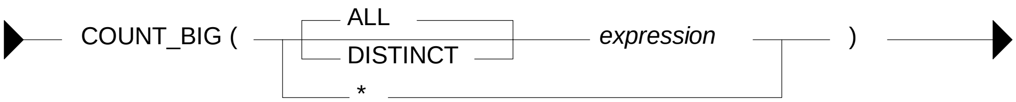


Figure 108, COUNT_BIG function syntax

```
SELECT   COUNT_BIG(*)                 AS C1         ANSWER
        ,COUNT_BIG(DEPT)              AS C2         ===================
        ,COUNT_BIG(DISTINCT DEPT)     AS C3         C1  C2  C3  C4  C5
        ,COUNT_BIG(DISTINCT DEPT/10)  AS C4         --- --- --- --- ---
        ,COUNT_BIG(DISTINCT DEPT)/10  AS C5         35. 35.  8.  7.  0.
FROM     STAFF;
```

Figure 109, COUNT_BIG function examples

COVARIANCE

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

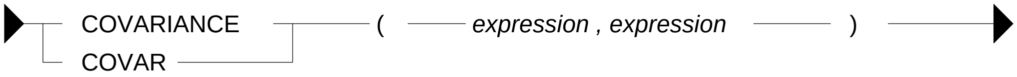


Figure 110, COVARIANCE function syntax

```
WITH TEMP1(C1, C2, C3, C4) AS          ANSWER
(VALUES (0                             ===============================
        ,0                             COV11   COV12   COV23   COV34
        ,0                             ------- ------- ------- -------
        ,RAND(1))                       83666. -83666. -1.4689 -0.0004
 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;
```

Figure 111, 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.

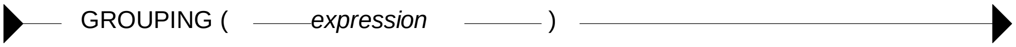


Figure 112, GROUPING function syntax

```
SELECT   DEPT                                              ANSWER
        ,AVG(SALARY)    AS SALARY                          ================
        ,GROUPING(DEPT) AS DF                              DEPT SALARY   DF
FROM     STAFF                                             ---- -------- --
GROUP BY ROLLUP(DEPT)                                        10 20865.86  0
ORDER BY DEPT;                                               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 113, 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.

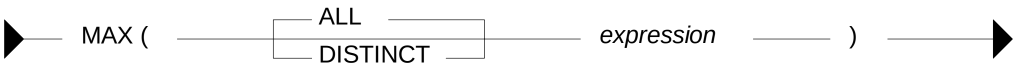


Figure 114, MAX function syntax

```
SELECT   MAX(DEPT)                                          ANSWER
        ,MAX(ALL DEPT)                                      ===============
        ,MAX(DISTINCT DEPT)                                  1   2   3   4
        ,MAX(DISTINCT DEPT/10)                              --- --- --- ---
FROM     STAFF;                                              84  84  84   8
```

Figure 115, 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 116, 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 117, 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 118, 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.

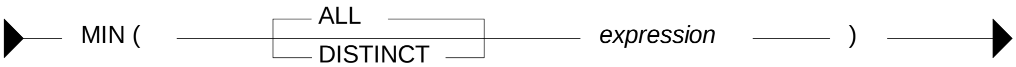


Figure 119, MIN function syntax

```
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 120, 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.

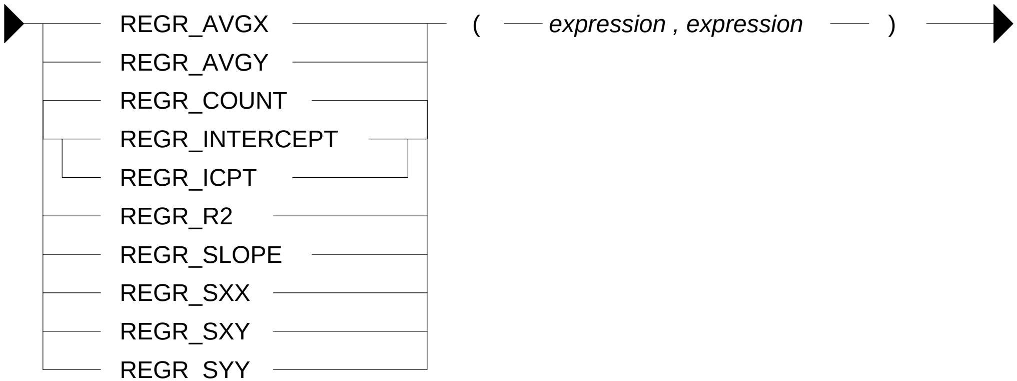


Figure 121, REGRESSION functions syntax

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 122, 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.

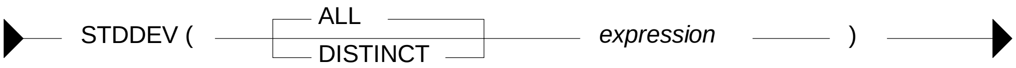


Figure 123, STDDEV function syntax

```
                                       ANSWER
                                       ===============================
                                       A1 S1            S2   S3   S4
                                       -- ------------- ---- ---- ----
SELECT AVG(DEPT) AS A1                 41 +2.3522355E+1 23.5 23.5 24.1
      ,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;
```

Figure 124, 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.

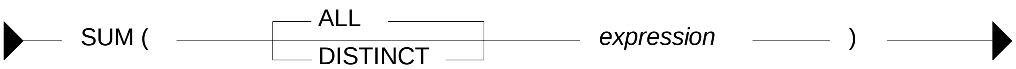


Figure 125, SUM function syntax

```
SELECT   SUM(DEPT)          AS S1             ANSWER
        ,SUM(ALL DEPT)      AS S2             ========================
        ,SUM(DISTINCT DEPT) AS S3              S1   S2   S3   S4   S5
        ,SUM(DEPT/10)       AS S4             ---- ---- ---- ---- ----
        ,SUM(DEPT)/10       AS S5             1459 1459  326  134  145
FROM     STAFF;
```

Figure 126, 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.

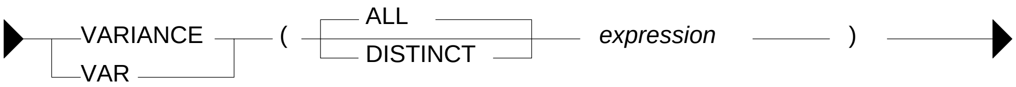


Figure 127, VARIANCE function syntax

```
                                        ANSWER
                                        ==============================
                                        A1 V1              V2  V3  V4
                                        -- --------------- --- --- ---
SELECT AVG(DEPT) AS A1                  41 +5.533012244E+2 553 553 582
      ,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;
```

Figure 128, VARIANCE function examples

NOTE: See the chapter titled "Statistical Analysis using SQL" for a more detailed description of how and when to use the STDDEV and VARIANCE functions.

OLAP Functions

Introduction

The OLAP (Online Analytical Processing) functions enable one sequence and rank query rows. They are especially useful in those environments, like some web servers, where the calling program is unable to do much processing logic.

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 129, 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
FROM     STAFF S1                                               ANSWER
WHERE    S1.NAME LIKE '%s%'               ============================
  AND    S1.ID      <  90                 JOB   ID SALARY   SUMSAL   R
ORDER BY S1.JOB                           ----- -- -------- -------- -
        ,S1.ID;                           Clerk 80 13504.60 13504.60 1
                                          Mgr   10 18357.50 31862.10 2
                                          Mgr   50 20659.80 52521.90 3
```

Figure 130, Using OLAP functions to get additional fields

But imagine that we don't have OLAP functions, or we are too stupid to figure out how to use them, or we are getting paid by the hour. We 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 30 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.#ROWS
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 131, 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 132, 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 133, 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 134, 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.

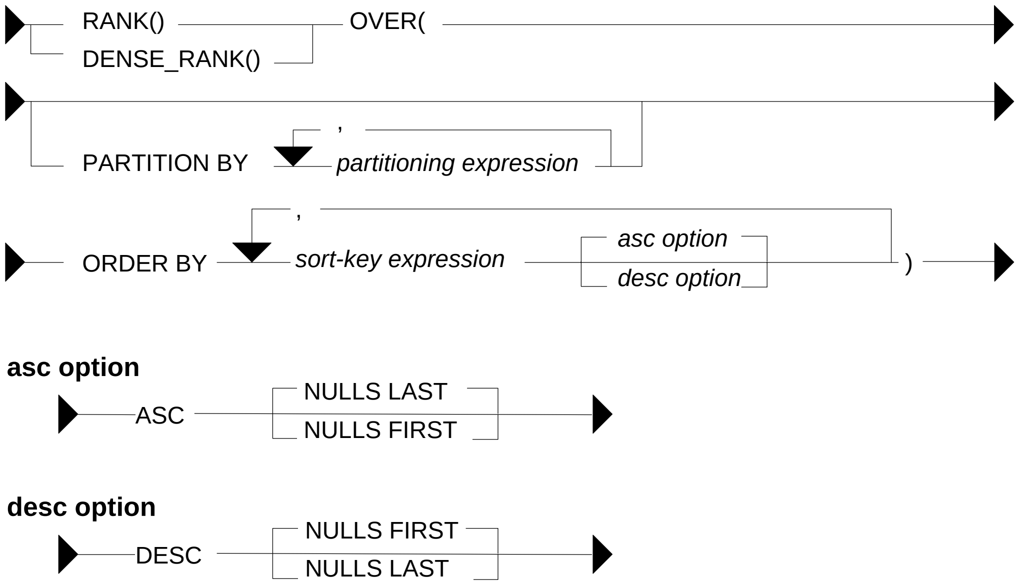


Figure 135, Ranking Functions syntax

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 51 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 136, 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 137, 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 138, 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 139, 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 140, 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 141, 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 142, 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 143, 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 144, 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 145, 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 146, 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 147, 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 148, 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 149, 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:

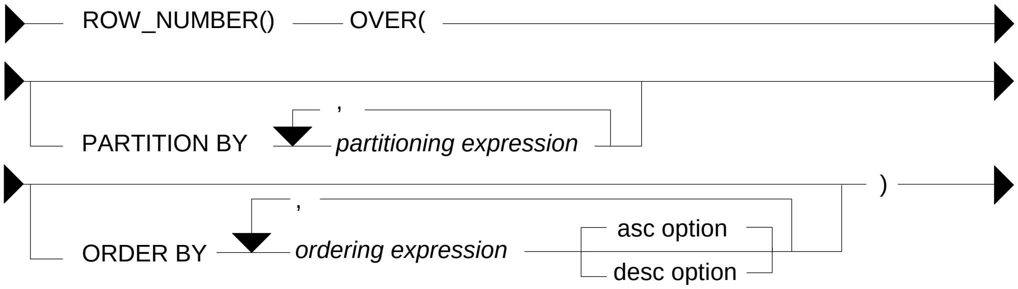


Figure 150, Numbering Function syntax

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 151, 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 152, 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 153, 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 154, 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 155, 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 156, Select first 3 rows, using FETCH FIRST notation

So far, the ROW_NUMBER and the FIRST FETCH 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 157, 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 158, 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 159, 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 160, 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 161, 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 162, 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 163, 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 164, 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 165, 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 166, 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 167, 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.

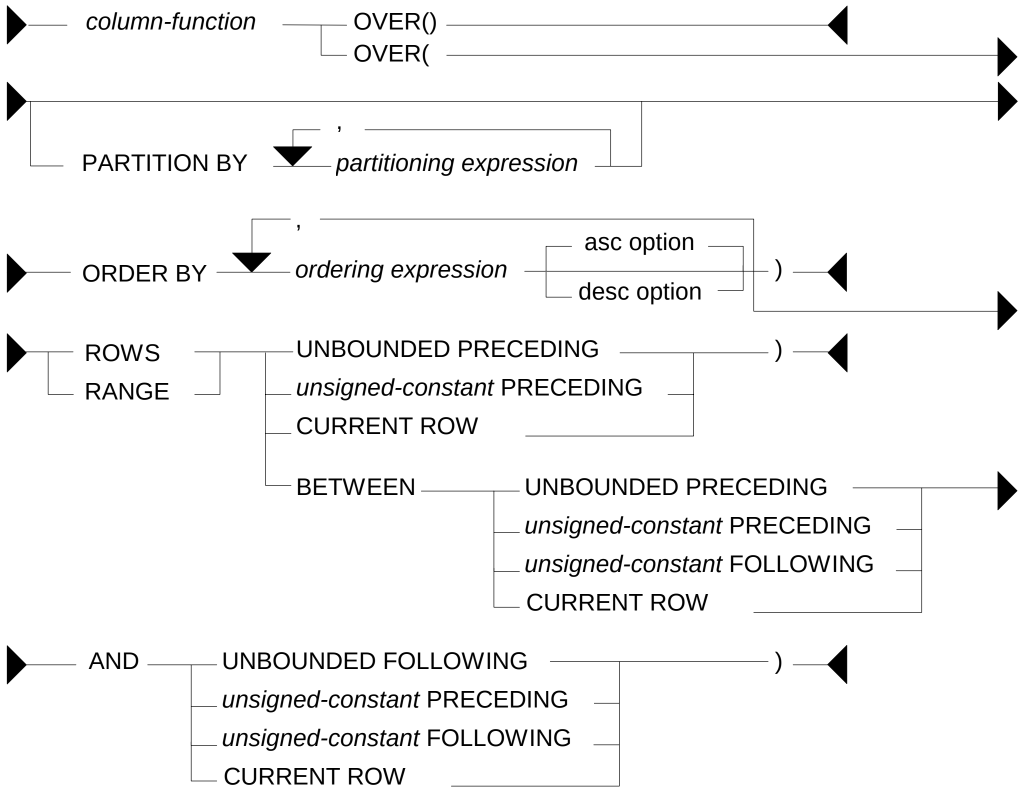


Figure 168, Aggregation Function syntax

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 63).
- 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 64).
- 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 65).
- 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 68).
- 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 169, 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 170, 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 171, 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 172, 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 173, 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 174, 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 175, 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 176, 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 177, 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 178, 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 179, 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 180,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 181, 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 182, 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 183, 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 184, 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 185, 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 186, 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 187, 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 188, 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, except for the addition of two new "ISO'" 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 189, 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 190, 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 191, 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 192, 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.

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                                ANSWER
(VALUES BIGINT(1)                                 ====================
 UNION ALL                                        BIG
 SELECT BIG * 256                                 --------------------
 FROM   TEMP                                                         1
 WHERE  BIG < 1E16                                                 256
)                                                                65536
SELECT BIG                                                    16777216
FROM   TEMP;                                                4294967296
                                                         1099511627776
                                                       281474976710656
                                                     72057594037927936
```

Figure 193, 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 194, 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 195, Convert FLOAT to DECIMAL and BIGINT, answer

See page 252 for a discussion on floating-point number manipulation.

BLOB

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

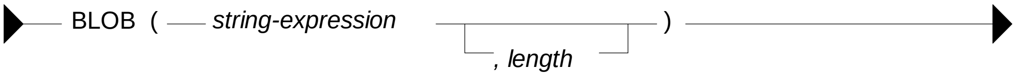


Figure 196, BLOB function syntax

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, except for decimal input, which returns double.

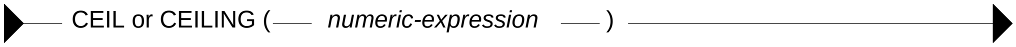


Figure 197, CEILING function syntax

```
SELECT D1                     ANSWER  (float output shortened)
      ,CEIL(D1)               =========================================
      ,F1                     D1     2           F1           4
      ,CEIL(F1)               -----  ----------  ----------  ----------
FROM   SCALAR;                 -2.4   -2.000E+0   -2.400E+0   -2.000E+0
                                0.0   +0.000E+0   +0.000E+0   +0.000E+0
                                1.8   +2.000E+0   +1.800E+0   +2.000E+0
```

Figure 198, CEIL function examples

Decimal values larger than 15 digits do not work correctly with the CEIL function because some rightmost digits are lost when the data is converted to double:

```
WITH TEMP1(N1) AS
(VALUES (CAST(12345678901234567890.1 AS DECIMAL(20))))
SELECT N1
      ,DECIMAL(CEIL(N1),20) AS N2
FROM   TEMP1;
                           ANSWER
                           ===========================================
                           N1                    N2
                           --------------------- ---------------------
                           12345678901234567890. 12345678901234570000.
```

Figure 199, CEIL function, example of incorrect result

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 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.
- 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 left-justified character string. 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.

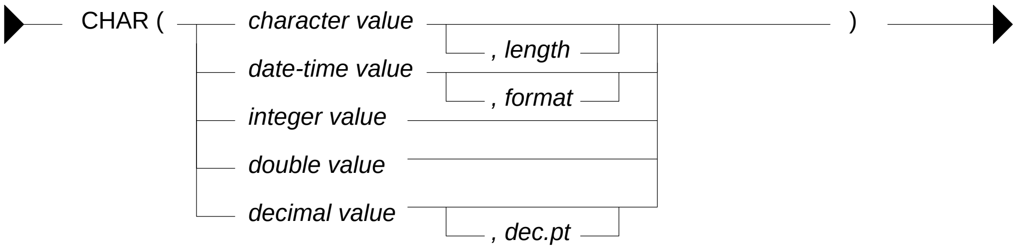


Figure 200, CHAR function syntax

The CHAR function is actually two with the same name. All input types except double are handled by a SYSIBM function, while double input is processed using a SYSFUN function.

```
SELECT C1                   ANSWER
      ,CHAR(C1,3)           ==========================================
      ,S1                   C1     2   S1     4      D1    6     7
      ,CHAR(S1)             ------ --- ------ ------ ----- ----- -----
      ,D1                   ABCDEF ABC     -2 -2      -2.4 -02.4 -02$4
      ,CHAR(D1)             ABCD   ABC      0 0        0.0 00.0  00$0
      ,CHAR(D1,'$')         AB     AB       1 1        1.8 01.8  01$8
FROM   SCALAR;
```

Figure 201, CHAR function examples - characters and numbers

When converting numeric input to character output, the CHAR function does not align the digits in the column (see field 6 above). Additional SQL can address this:

```
SELECT D1                                           ANSWER
      ,CHAR(D1) AS D2                               ==================
      ,CASE                                         D1    D2    D3
          WHEN D1 < 0 THEN CHAR(D1)                 ----- ----- ------
          ELSE      '+' || CHAR(D1)                  -2.4 -02.4 -02.4
       END AS D3                                      0.0 00.0  +00.0
FROM   SCALAR;                                        1.8 01.8  +01.8
```

Figure 202, Align CHAR function output

Observe that the D3 field above has a trailing blank. This was added during the concatenation operation. It can be removed using the SUBSTR function - if you know exactly what length output you desire. The RTRIM function can also be used, but then the output is 4000 bytes long, which is a nuisance to work with.

```
                                      ANSWER (float output shortened)
WITH TEMP1(C1) AS                     ================================
(VALUES ('12345')                     C1     C>D               C>D>C
       ,('-23.5')                     -----  ---------------   --------
       ,('1E+45')                     12345  +1.2345000E+004   1.2345E4
       ,('-2e05'))                    -23.5  -2.3500000E+001   -2.35E1
SELECT C1                             1E+45  +1.0000000E+045   1.0E45
      ,DOUBLE(C1) AS "C>D"            -2e05  -2.0000000E+005   -2.0E5
      ,CHAR(DOUBLE(C1)) AS "C>D>C"
FROM   TEMP1;
```

Figure 203, CHAR function example - double

```
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 204, 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

Both the DIGITS and the CHAR functions convert numeric input (not float) into character strings, but the output from each is quite different - and neither is perfect. 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 D1                                            ANSWER
      ,CHAR(D1)   AS CD1                             =================
      ,DIGITS(D1) AS DD1                             D1     CD1    DD1
FROM   SCALAR;                                       -----  -----  ---
                                                      -2.4  -02.4  024
                                                       0.0  00.0   000
                                                       1.8  01.8   018
```

Figure 205, DIGITS vs. CHAR

CHR

Converts integer input in the range 0 through 255 to the equivalent ASCII character value. An input value outside the valid range returns a null. 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
WHERE  ID = 10;
```

Figure 206, 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 207, 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 208, 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 209, 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 210, 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 211, 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 212, 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 213, 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 214, RADIAN, COS, and SIN functions example

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").

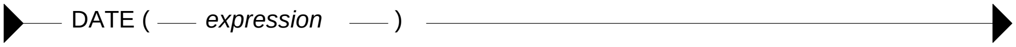


Figure 215, DATE function syntax

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 216, 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 217, 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 218, 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 219, 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 220, 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 221, 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 222, 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 223, 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 224, 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.

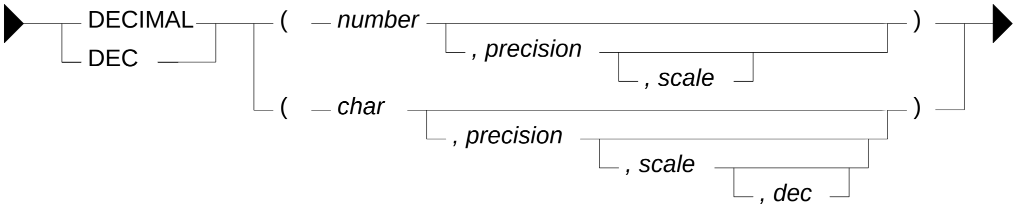


Figure 225, DECIMAL function syntax

```
WITH TEMP1(N1,N2,C1,C2) AS                          ANSWER
(VALUES   (123                                      ==========================
          ,1E2                                      DEC1  DEC2   DEC3   DEC4
          ,'123.4'                                  ----- ------ ------ ------
          ,'567$8'))                                 123.  100.0  123.4  567.8
SELECT DEC(N1,3)       AS DEC1
      ,DEC(N2,4,1)     AS DEC2
      ,DEC(C1,4,1)     AS DEC3
      ,DEC(C2,4,1,'$') AS DEC4
FROM   TEMP1;
```

Figure 226, DECIMAL function examples

> WARNING: Converting a floating-point number to decimal may get different results from converting the same number to integer. See page 252 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.

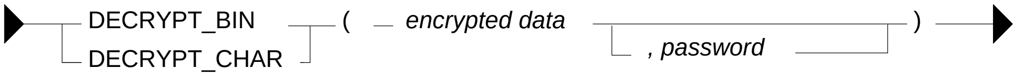


Figure 227, DECRYPT function syntax

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 228, 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 Scoutten  S350  1
  AND    B.ID > 150                     Sanders S536 Molinare  M456  1
  AND    B.ID < 250                     Sanders S536 Lu        L000  0
ORDER BY DF DESC;                       Sanders S536 Abrahams  A165  0
                                        Sanders S536 Kermisch  K652  0
```

Figure 229, 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 230, DIGITS function examples

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

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                ANSWER (output shortened)
(VALUES ('12345',12.4)              ==================================
       ,('-23.5',1234)              C1D               D1D
       ,('1E+45',-234)              ----------------  ----------------
       ,('-2e05',+2.4))             +1.23450000E+004  +1.24000000E+001
SELECT DOUBLE(C1) AS C1D            -2.35000000E+001  +1.23400000E+003
      ,DOUBLE(D1) AS D1D            +1.00000000E+045  -2.34000000E+002
FROM   TEMP1;                       -2.00000000E+005  +2.40000000E+000
```

Figure 231, DOUBLE function examples

See page 252 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.

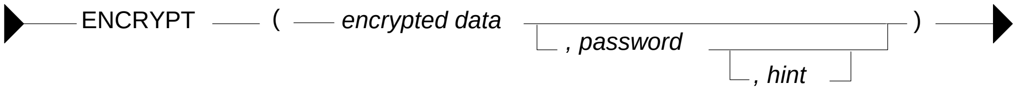


Figure 232, DECRYPT function syntax

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 233, 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 234, 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, except for decimal input, which returns double.

```
SELECT D1                    ANSWER  (float output shortened)
      ,FLOOR(D1)             =========================================
      ,F1                    D1     2           F1          4
      ,FLOOR(F1)             -----  ----------  ----------  ----------
FROM   SCALAR;                -2.4   -3.000E+0   -2.400E+0   -3.000E+0
                               0.0   +0.000E+0   +0.000E+0   +0.000E+0
                               1.8   +1.000E+0   +1.800E+0   +1.000E+0
```

Figure 235, 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 236, 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 237, 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 238, 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 239, 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 240, 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 241, 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 242, 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 187 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 243, 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.

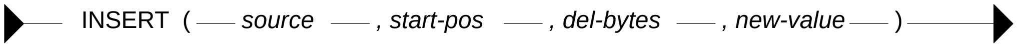


Figure 244, INSERT function syntax

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 245, 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 246, 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 247, 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'               ANSWER
        ) AS XXX                            ==========================
ORDER BY BD;                                BD         2       3
                                            ---------- ------- -------
                                            05/17/1926 2424653 2424653
                                            03/28/1936 2428256 2428256
                                            07/09/1946 2432011 2432011
                                            04/12/1955 2435210 2435210
```

Figure 248, 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 twentieth century, to derive the Julian date one must subtract 15 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):

```
                                         ANSWER
                                         =============================
                                         DT         DJ1        DJ2
WITH TEMP1(DT1) AS                       ---------- ---------- -------
(VALUES ('1997-01-01')                   01/01/1997 12/17/1996 1997001
       ,('1997-01-02')                   01/02/1997 12/18/1996 1997002
       ,('1997-12-31'))                  12/31/1997 12/16/1997 1997365
SELECT DATE(DT1) AS DT
      ,DATE(DT1) - 15 DAYS AS DJ1
      ,YEAR(DT1) * 1000 + DAYOFYEAR(DT1) AS DJ2
FROM   TEMP1;
```

Figure 249, 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 250, LCASE function example

Documentation Comment

According to the DB2 UDB V7.2 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 251, 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 252, 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)                               ANSWER
      ,LENGTH(F1)                               =======================
      ,LENGTH(S1)                               1    2    3    4    5
      ,LENGTH(C1)                               ---  ---  ---  ---  ---
      ,LENGTH(RTRIM(C1))                          2    8    2    6    6
FROM   SCALAR;                                    2    8    2    6    4
                                                  2    8    2    6    2
```

Figure 253, 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                       ANSWER
(VALUES (1),(123),(1234)                ===============================
       ,(12345),(123456))               N1      L1
SELECT N1                               ------  -----------------------
      ,LOG(N1) AS L1                         1   +0.00000000000000E+000
FROM   TEMP1;                              123   +4.81218435537241E+000
                                          1234   +7.11801620446533E+000
                                         12345   +9.42100640177928E+000
                                        123456   +1.17236400962654E+001
```

Figure 254, 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.

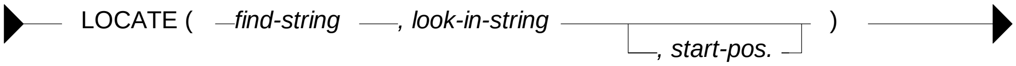


Figure 255, LOCATE function syntax

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                                   ANSWER
      ,LOCATE('D', C1)                      ==========================
      ,LOCATE('D', C1,2)                    C1      2    3    4    5
      ,LOCATE('EF',C1)                      ------  ---  ---  ---  ---
      ,LOCATE('A', C1,2)                    ABCDEF    4    4    5    0
FROM   SCALAR;                              ABCD      4    4    0    0
                                            AB        0    0    0    0
```

Figure 256, 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 257, 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 258, 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 259, 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 260, 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 261, 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 **243** 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 262, 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 263, MOD function example

MONTH

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

MONTHNAME

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

```
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 264, 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 265, 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 266, 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.

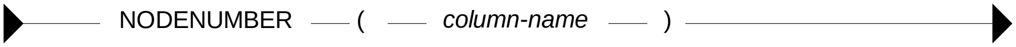


Figure 267, NODENUMBER function syntax

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

Figure 268, 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 269, 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 270, 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 271, 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 272, 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.

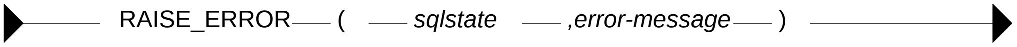


Figure 273, RAISE_ERROR function syntax

```
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 274, RAISE_ERROR function example

RAND

> WARNING: Using the RAND function in a predicate can result in unpredictable results. See page 245 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 275, 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. See page 261 for details.

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 276, 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 277, 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 278, 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 279, 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 85), 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 245 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 280, 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 281, 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 282, 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 283, 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.

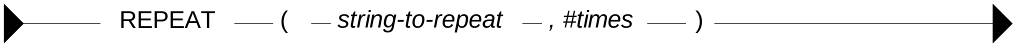


Figure 284, REPEAT function syntax

```
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 285, REPEAT function example

REPLACE

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

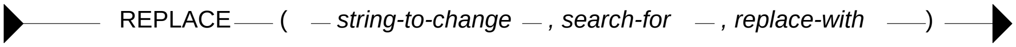


Figure 286, REPLACE function syntax

```
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 287, 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 288, 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 289, 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 290, ROUND function examples

RTRIM

Trims the right-most blanks of a character string.

```
SELECT C1                                     ANSWER
      ,RTRIM(C1)                              ========================
      ,LENGTH(C1)                             C1      2       3    4
      ,LENGTH(RTRIM(C1))                      ------  ------  ---  ---
FROM   SCALAR;                                ABCDEF  ABCDEF    6    6
                                              ABCD    ABCD      6    4
                                              AB      AB        6    2
```

Figure 291, 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 292, 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    COS    TAN
 SELECT  N1 + 10                               --  -----  -----  -----
 FROM    TEMP1                                  0  0.000  0.000  0.000
 WHERE   N1 < 90)                              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 COS        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 293, SIN function example

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 294, SMALLINT function examples

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 Scoutten  S350  1
  AND    B.ID > 150                     Sanders S536 Molinare  M456  1
  AND    B.ID < 250                     Sanders S536 Lu        L000  0
ORDER BY DF DESC;                       Sanders S536 Abrahams  A165  0
                                        Sanders S536 Kermisch  K652  0
```

Figure 295, 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 5).
- 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 item 3 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 296, 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 297, 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 298, 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 299, 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 300, 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.

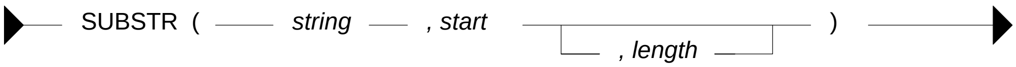


Figure 301, SUBSTR function syntax

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                    ANSWER
(VALUES    (  6,'123456789')                 =========================
          ,(  4,'12345'    )                 LEN DAT1      LDAT SUBDAT
          ,( 16,'123'      )                 --- --------- ---- ------
)                                              6 123456789    9 123456
SELECT    LEN                                  4 12345        5 1234
         ,DAT1                               <error>
         ,LENGTH(DAT1)  AS LDAT
         ,SUBSTR(DAT1,1,LEN) AS SUBDAT
FROM      TEMP1;
```

Figure 302, 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                    ANSWER
(VALUES    (  6,'123456789')                 =========================
          ,(  4,'12345'    )                 LEN DAT1      LDAT SUBDAT
          ,( 16,'123'      )                 --- --------- ---- ------
)                                              6 123456789    9 123456
SELECT    LEN                                  4 12345        5 1234
         ,DAT1                                16 123          3 123
         ,LENGTH(DAT1)  AS LDAT
         ,SUBSTR(DAT1,1,CASE
                        WHEN LEN < LENGTH(DAT1) THEN LEN
                        ELSE LENGTH(DAT1)
                        END ) AS SUBDAT
FROM      TEMP1;
```

Figure 303, 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                                ANSWER
      ,LENGTH(NAME)            AS LEN      ===========================
      ,SUBSTR(NAME,5)          AS S1       NAME     LEN S1   L1 S2  L2
      ,LENGTH(SUBSTR(NAME,5))   AS L1       -------- --- ---- -- --- --
      ,SUBSTR(NAME,5,3)        AS S2       Sanders    7 ers   3 ers  3
      ,LENGTH(SUBSTR(NAME,5,3)) AS L2      Pernal     6 al    2 al   3
FROM   STAFF                               Marenghi   8 nghi  4 ngh  3
WHERE  ID < 60;                            O'Brien    7 ien   3 ien  3
                                           Hanes      5 s     1 s    3
```

Figure 304, 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 305, Full-select with external table reference

See page 30 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 306, 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 307, TABLE_SCHEMA and TABLE_NAME functions example

TAN

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

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 308, TIMESTAMP function examples

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 309, 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 310, 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;
                         ANSWER
                         =============================================
                         MS1               MS2               DIFF
                         ----------------- ----------------- ---------
                         62929699920034000 62929699200000000 720034000
                         62929699954000000 62929699200000000 754000000
```

Figure 311, Difference in microseconds between two timestamps

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.

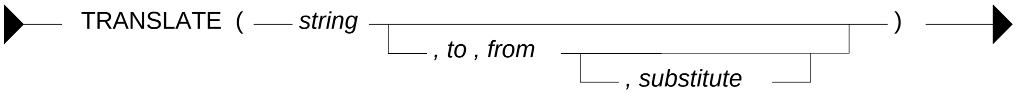


Figure 312, TRANSLATE function syntax

Usage Notes

- The use of the input string alone generates upper case output.
- When "to" and "from" values are provided, each individual "to" character in the input string is replaced by the matching "from" value (if there is one).
- If there is no "from" value for a particular "to" value, those characters in the input string that match the "to" are set to blank (if there is no substitute value).
- A fourth, optional, parameter can be provided that represents a substitute value for those "from" values that have no matching "to" value.

```
                                                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'
FROM   STAFF
WHERE  ID = 10;
```

Figure 313, 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 314, 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, except for decimal which returns double. 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 315, 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 316, 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 317, VARCHAR function examples

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 318, 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 249).

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 319, 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 320, YEAR and WEEK functions example

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

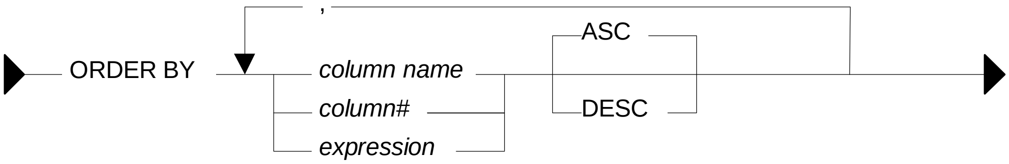


Figure 321, ORDER BY syntax

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 322, 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 323, 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 324, 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 325, 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 326, 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 327, 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 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.

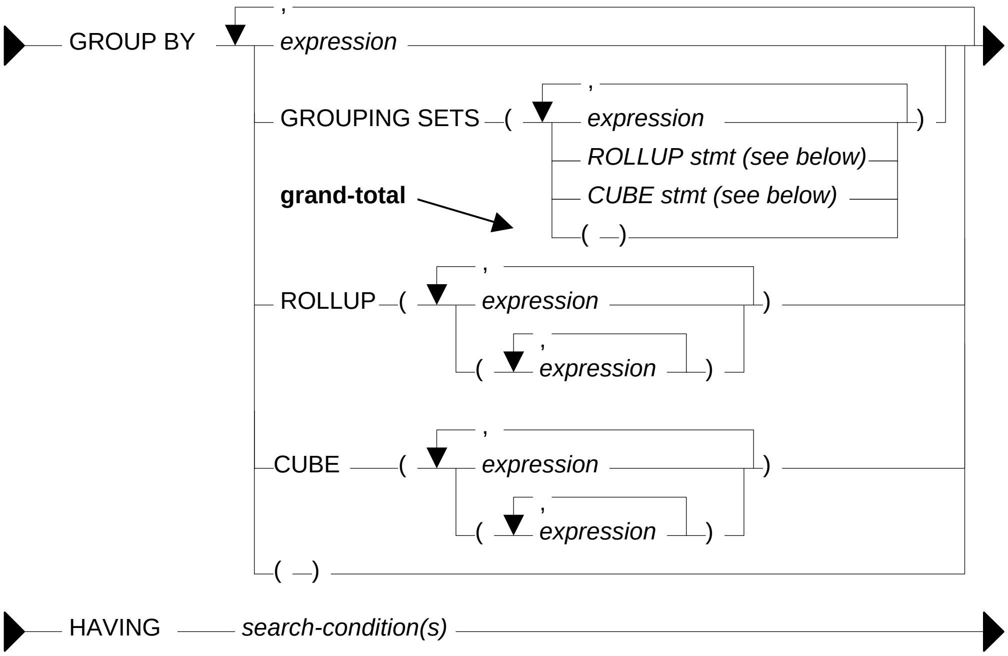


Figure 328, GROUP BY syntax

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 329, 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 330, 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 331, 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 332, 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 333, 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 334, 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 335, 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 336, 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 337, 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 338, 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 339, 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 340, 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 341, 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 342, 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 343, 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 344, 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 345, 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 346, 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 347, 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 348, 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 349, 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 350, 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 351, 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 352, 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 353, 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 354, 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 355, 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 356, 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 357, ROLLUP on SEX, then DEPT

The next example has two independent rollups. These work as follows:

- 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 358, 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 359, 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 360, 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 361, 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 362, 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 363, 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 364, 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 365, 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 366, 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                             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       A  A00  -   128500  3  0  0  1
        ,GROUPING(SEX)      AS FS       A  -    F    52750  1  0  1  0
FROM     EMPLOYEE_VIEW                  A  -    M    75750  2  0  1  0
GROUP BY CUBE(D1, DEPT, SEX)            A  -    -   128500  3  0  1  1
ORDER BY D1                             B  B01  M    41250  1  0  0  0
        ,DEPT                           B  B01  -    41250  1  0  0  1
        ,SEX;                           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 367, CUBE example

Here is the same query expressed as 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 ((D1, DEPT, SEX)
                       ,(D1,DEPT)
                       ,(D1,SEX)
                       ,(DEPT,SEX)
                       ,(D1)
                       ,(DEPT)
                       ,(SEX)
                       ,())
ORDER BY D1
        ,DEPT
        ,SEX;
```

Figure 368, 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 369, 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 370, 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 371, 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 372, 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.                                            ,())

                                 GROUP BY ROLLUP(D1,DEPT)
                                         ,ROLLUP(SEX)
```

Figure 373, 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 374, Get lots of sub-totals, using CUBE

In the above query, the GROUPING function (see page 43) 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 375, 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 376, 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 377, 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 378, 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 379, 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 380, COUNT and No Rows

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 161) the matching rows are joined (in a sense) one-after-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 BEWTEEEN 10 AND 30;              |--|--------|   |--|------|
                                           |10|Sanders |   |20|Sales |
CREATE VIEW STAFF_V2 AS                    |20|Pernal  |   |30|Mgr   |
SELECT ID, JOB                             |30|Marenghi|   |40|Sales |
FROM   STAFF                               +-----------+   |50|Mgr   |
WHERE  ID BETWEEN 20 AND 50;                               +---------+
```

Figure 381, Sample Views used in Join Examples

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.

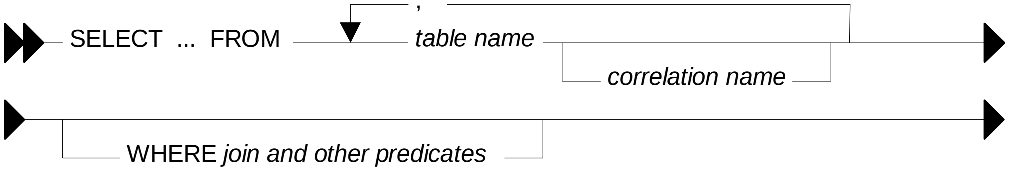


Figure 382, Join Syntax #1

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 Mgr
ORDER BY V1.ID;
```

Figure 383, Sample two-table join

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

Figure 384, 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, while WHERE conditions are applied after the join and constrain the result.

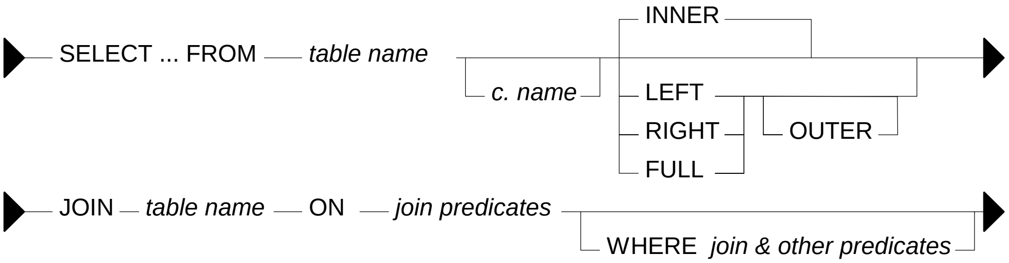


Figure 385, Join Syntax #2

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 Mgr
ON       V1.ID = V2.ID
ORDER BY V1.ID;
```

Figure 386, Sample two-table inner join

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

Figure 387, Sample three-table inner 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   |                              ID NAME     ID JOB
|--|--------|   |--|------|      =========>              -- -------- -- -----
|10|Sanders |   |20|Sales |                              20 Pernal   20 Sales
|20|Pernal  |   |30|Mgr   |                              30 Marenghi 30 Mgr
|30|Marenghi|   |40|Sales |
+-----------+   |50|Mgr   |
                +---------+
```

Figure 388, 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
                                                         30 Marenghi 30 Mgr
```

Figure 389, Inner Join SQL (1 of 2)

```
SELECT   *
FROM     STAFF_V1 V1
INNER JOIN
         STAFF_V2 V2
ON       V1.ID = V2.ID
ORDER BY V1.ID;
```

Figure 390, Inner Join SQL (2 of 2)

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 |                              10 Sanders  20 Sales
|20|Pernal  |   |30|Mgr   |                              10 Sanders  30 Mgr
|30|Marenghi|   |40|Sales |                              10 Sanders  40 Sales
+-----------+   |50|Mgr   |                              10 Sanders  50 Mgr
                +---------+                              20 Pernal   20 Sales
                                                         20 Pernal   30 Mgr
                                                         20 Pernal   40 Sales
                                                         20 Pernal   50 Mgr
                                                         30 Marenghi 20 Sales
                                                         30 Marenghi 30 Mgr
                                                         30 Marenghi 40 Sales
                                                         30 Marenghi 50 Mgr
```

Figure 391, 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;
```

Figure 392, 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;
```

Figure 393, 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 Mgr   30
ORDER BY V2A.ID                                            30 Mgr   50
        ,V2B.ID;
```

Figure 394, Partial Cartesian Product SQL

In the above example we joined the two tables 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                                            Mgr       6
ORDER BY #ROWS                                             Sales     6
        ,V2.JOB;
```

Figure 395, 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.

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|Mgr   |                       20 Pernal   20 Sales
|30|Marenghi|   |40|Sales |                       30 Marenghi 30 Mgr
+-----------+   |50|Mgr   |
                +---------+
```

Figure 396, Example of Left Outer Join

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

Figure 397, 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 1,3;
```

Figure 398, Left Outer Join SQL (2 of 2)

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|Mgr   |                       30 Marenghi 30 Mgr
|30|Marenghi|   |40|Sales |                       -  -        40 Sales
+-----------+   |50|Mgr   |                       -  -        50 Mgr
                +---------+
```

Figure 399, Example of Right Outer Join

```
SELECT   *
FROM     STAFF_V1 V1
RIGHT OUTER JOIN
         STAFF_V2 V2
ON       V1.ID = V2.ID
ORDER BY V2.ID;
```

Figure 400, 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 Mgr
SELECT   CAST(NULL AS SMALLINT)   AS ID           -  -        40 Sales
        ,CAST(NULL AS VARCHAR(9)) AS NAME         -  -        50 Mgr
        ,V2.*
FROM     STAFF_V2 V2
WHERE    V2.ID NOT IN
        (SELECT ID FROM STAFF_V1)
ORDER BY 1,3;
```

Figure 401, Right Outer Join SQL (2 of 2)

Outer Joins

An 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|Mgr   |                       20 Pernal   20 Sales
|30|Marenghi|   |40|Sales |                       30 Marenghi 30 Mgr
+-----------+   |50|Mgr   |                       -  -        40 Sales
                +---------+                       -  -        50 Mgr
```

Figure 402, Example of Full Outer Join

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

Figure 403, Full Outer Join SQL

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

```
SELECT   V1.*
        ,V2.*
FROM     STAFF_V1 V1
        ,STAFF_V2 V2
WHERE    V1.ID = V2.ID
UNION
SELECT   V1.*
        ,CAST(NULL AS SMALLINT) AS ID
        ,CAST(NULL AS CHAR(5))  AS JOB
FROM     STAFF_V1 V1
WHERE    V1.ID NOT IN
        (SELECT ID FROM STAFF_V2)
UNION
SELECT   CAST(NULL AS SMALLINT)   AS ID
        ,CAST(NULL AS VARCHAR(9)) AS NAME
        ,V2.*
FROM     STAFF_V2 V2
WHERE    V2.ID NOT IN
        (SELECT ID FROM STAFF_V1)
ORDER BY 1,3;
```

Figure 404, 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.

Outer Join Notes

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 Mgr
ORDER BY ID;                                         40 ?        Sales
                                                     50 ?        Mgr
```

Figure 405, Use of COALESCE function in outer join

ON Predicate Processing

An ON check is quite unlike a WHERE check in that the former will **never** result in a row being excluded from the answer set (in an outer join). All it does is categorize the input rows into one of two types: matching or non-matching. For example, in the following example the ON matches 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 Mgr
                                                   - -        40 Sales
                                                   - -        50 Mgr
```

Figure 406, 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 Mgr
        ,V2.ID;                                    - -        20 Sales
                                                   - -        40 Sales
                                                   - -        50 Mgr
```

Figure 407, 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 an ON predicate never removes rows, it simply categorizes them as 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 tables:

```
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
                                                   - -        30 Mgr
                                                   - -        40 Sales
                                                   - -        50 Mgr
```

Figure 408, Full Outer Join, match on keys (no rows match)

ON checks are 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. In joins, 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, everything fell into the "outer join" category. So, in this query, 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  - -
                                                   - -        20 Sales
                                                   - -        30 Mgr
                                                   - -        40 Sales
                                                   - -        50 Mgr
```

Figure 409, 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 Mgr
        ,V2.ID;                                   10 Sanders  40 Sales
                                                  10 Sanders  50 Mgr
                                                  20 Pernal   20 Sales
                                                  20 Pernal   30 Mgr
                                                  20 Pernal   40 Sales
                                                  20 Pernal   50 Mgr
                                                  30 Marenghi 20 Sales
                                                  30 Marenghi 30 Mgr
                                                  30 Marenghi 40 Sales
                                                  30 Marenghi 50 Mgr
```

Figure 410, Full Outer Join, don't match on keys (all rows match)

WHERE Predicate Processing

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 Mgr
ORDER BY 1,3;
```

Figure 411, 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 tables (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|Mgr   |         V1.ID < 30
|30|Marenghi|    |40|Sales |
+-----------+    |50|Mgr   |
                 +---------+
```

Figure 412, 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;
```

Figure 413, 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 tables:

```
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;                                      30 Marenghi -  -
                                                   -  -        30 Mgr
                                                   -  -        40 Sales
                                                   -  -        50 Mgr
```

Figure 414, 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 Mgr
ORDER BY 1,3;                                     -  -        40 Sales
                                                  -  -        50 Mgr
```

Figure 415, 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;                                     -  -        50 Mgr
```

Figure 416, 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 Sales
   OR    V1.ID IS NULL                            -  -        40 Sales
ORDER BY 1,3;                                     -  -        50 Mgr
```

Figure 417, 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.

Listing non-matching rows only

Imagine that we wanted to do an outer join on our two test tables, 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|Mgr   |                        -  -       40 Sales
|30|Marenghi|   |40|Sales |                        -  -       50 Mgr
+-----------+   |50|Mgr   |
                +---------+
```

Figure 418, 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;
```

Figure 419, 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;                          -- ------- ---- -- ----- ----
                                         10 Sanders V1    -  -    -
                                         -  -       -    40 Sales V2
                                         -  -       -    50 Mgr   V2
```

Figure 420, 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
FULL OUTER JOIN                                                 ANSWER
       V2 V2                             =============================
ON       V1.ID  =  V2.ID                 ID NAME    FLAG ID JOB   FLAG
WHERE    V1.FLAG IS NULL                 -- ------- ---- -- ----- ----
   OR    V2.FLAG IS NULL                 10 Sanders V1    -  -    -
ORDER BY V1.ID                           -  -       -    40 Sales V2
        ,V2.ID;                          -  -       -    50 Mgr   V2
```

Figure 421, 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|Mgr   |
   OR    V2.ID IS NULL                      |30|Marenghi|   |40|Sales |
ORDER BY V1.ID                              +-----------+   |50|Mgr   |
        ,V2.ID;                                             +---------+
```

Figure 422, Outer Join SQL, getting only non-matching rows

Outer Join in SELECT Phrase

Below is a fairly typical left outer join between the two sample tables:

```
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                                30 Marenghi Mgr
ORDER BY V1.ID;
```

Figure 423, 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     JOB
          FROM    STAFF_V2 V2                         -- -------- -----
          WHERE   V1.ID = V2.ID)                      10 Sanders  -
FROM     STAFF_V1 V1                                  20 Pernal   Sales
ORDER BY V1.ID;                                       30 Marenghi Mgr
```

Figure 424, Outer Join done in SELECT phrase of SQL

Certain rules apply when using the above syntax:

- The nested table expression acts as a left outer join.
- Only one column and one row (at most) can be returned by the expression.
- If no row is returned, the result is null.

Multiple Columns

If you want to retrieve two columns using this syntax style, you need to have two independent nested table expressions:

```
SELECT   V1.ID                                     ANSWER
        ,V1.NAME                                   ====================
        ,(SELECT  V2.JOB                           ID NAME     JOB   J2
          FROM    STAFF_V2 V2                      -- -------- ----- --
          WHERE   V1.ID = V2.ID)                   10 Sanders  -      -
        ,(SELECT  LENGTH(V2.JOB) AS J2             20 Pernal   Sales  5
          FROM    STAFF_V2 V2                      30 Marenghi Mgr    5
          WHERE   V1.ID = V2.ID)
FROM     STAFF_V1 V1
ORDER BY V1.ID;
```

Figure 425, 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   V1.ID                                    ANSWER
        ,V1.NAME                                  ====================
        ,V2.JOB                                   ID NAME     JOB   J2
        ,LENGTH(V2.JOB) AS J2                     -- -------- ----- --
FROM     STAFF_V1 V1                              10 Sanders  -      -
LEFT OUTER JOIN                                   20 Pernal   Sales  5
         STAFF_V2 V2                              30 Marenghi Mgr    5
ON       V1.ID = V2.ID
ORDER BY V1.ID;
```

Figure 426, 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;
```

Figure 427, 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 428, 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|Mgr   |    V2.JOB LIKE 'S%'      20 Pernal   Sales
|30|Marenghi|   |40|Sales |                          30 Marenghi -
+-----------+   |50|Mgr   |
                +---------+
```

Figure 429, 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;
```

Figure 430, 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;
```

Figure 431, 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;
```

Figure 432, Outer Join, WHERE done independently - correct

Summary

- If you don't like working with nulls, but you need to do outer joins, then life is tough. In an outer join, non-matching rows have null values as placeholders.
- From a logical perspective, all WHERE conditions are applied after the join. For performance reasons, DB2 may apply some literal checks before the join, especially in an inner join, where doing this does not affect the result set. But in an outer join, they may have to be applied last, else the result will be wrong.
- The ON predicates in a full outer join never remove rows. They simply determine what rows are matching versus not. To eliminate rows, one must use a WHERE condition.
- All WHERE conditions that join tables act as if they are doing an inner join, even when they are written in an outer join.
- 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.

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 433, 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:

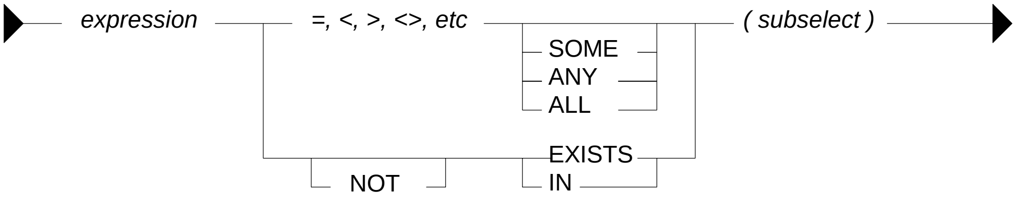


Figure 434, Sub-query syntax diagram

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 435, 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 436, 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 437, 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 438, 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 439, 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 440, 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 151) 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 441, 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 442, 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 443, 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 444, 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 445, 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 446, 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 447, 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 448, List of values in T2C <> T1A value

To make a NOT EXISTS sub-query that is logically equivalent to the ANY 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 449, 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 ANY 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 450, 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 451, 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 452, 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 453, 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 454, 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 455, 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 456, 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 457, 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 458,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 459, 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 460, 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 461, 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 462, 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 463, 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 464, 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  | - |
       WHERE  T1.T1A = T2.T2C);                |B  |BB |  |A  | - | - |
                                               |C  |CC |  +-----------+
SELECT *                                       +-------+    null value
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 465, 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 466, 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 467, 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 155 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 468, 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 469, 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 149 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 470, 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 471, 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 472, 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 473, 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

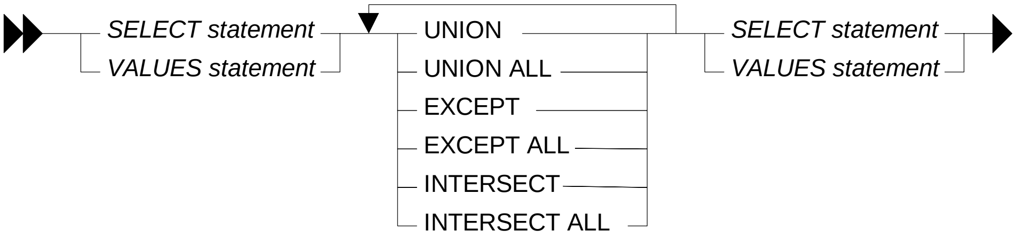


Figure 474, Union, Except, and Intersect syntax

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 475, 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 476, 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 477, 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 478, 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 479, 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 480, Use of parenthesis in Union

View Usage

Views can be defined which include the UNION, INTERSECT, and/or EXCEPT operators. This enables one to hide complex SQL logic from casual users.

```
CREATE VIEW VVVV (COL1, COL2) AS
SELECT COL1, COL2 FROM TABLE1
UNION ALL
SELECT COL1, COL3 FROM TABLE2
EXCEPT
SELECT COL1, COL1 FROM TABLE1;
```

Figure 481, View containing Union All

In some (but not all) cases, views that include UNION (or related) operators can be inefficient to use because external search criteria (on the view itself) may not be applied until after the view is resolved. For example, in the following statement the COL2 predicate will not be applied until after the all the rows in the view have be obtained:

```
SELECT *
FROM   VVVV
WHERE  COL2 > 'BB';
```

Figure 482, Select from view that contains Union All

> WARNING: Moving the COL2 predicate shown above into the view definition may make the SQL run faster, but it may also change the answer set.

Outer Join Usage

The UNION, INTERSECT, and EXCEPT expressions can be used in combination to effect an outer join between two tables. However, the resultant SQL is extremely complicated and, for this reason, is not shown here. It is suggested that the standard outer-join syntax (see page 138) be used instead.

Summary Tables

As their name implies, summary tables maintain a summary of the data in another table. The DB2 optimizer knows about summary tables, and it can use them instead of real tables when working out an access path. To illustrate, imagine the following summary table:

```
CREATE TABLE FRED.STAFF_SUMMARY AS
(SELECT   DEPT
         ,COUNT(*)       AS COUNT_ROWS
         ,SUM(ID)        AS SUM_ID
 FROM     FRED.STAFF
 GROUP BY DEPT
 )DATA INITIALLY DEFERRED REFRESH IMMEDIATE
```

Figure 483, Sample Summary Table DDL

Below on the left is a query that is very similar to the one used in the above CREATE statement. The DB2 optimizer will convert this query into the optimized equivalent shown on the right, which uses the summary table. Because the data in the summary table is maintained in sync with the source table, both statements will return the same answer.

```
ORIGINAL QUERY                       OPTIMIZED QUERY
==============                       =================================
SELECT   DEPT                        SELECT  Q1.DEPT AS "DEPT"
        ,AVG(ID)                            ,Q1.SUM_ID / Q1.COUNT_ROWS
FROM     FRED.STAFF                  FROM    FRED.STAFF_SUMMARY AS Q1
GROUP BY DEPT
```

Figure 484, Original and Optimized queries

When used appropriately, summary tables can give dramatic improvements in query performance. However, there is a cost involved. If the source data changes a lot, or is not summarized (in a query) very often, or does not reduce much when summarized, then summary tables may cost more than what they save.

Summary Table Types

The summary table type is defined using clauses in the CREATE TABLE statement:

- The DEFINITION ONLY clause creates a summary table using a SELECT statement to define the fields. In other words, a table is created that will accept the results of the SELECT, but no SELECT is run.
- The DATA INITIALLY DEFERRED REFRESH DEFERRED clause creates a table that is also based on a SELECT statement. But the difference here is that DB2 stores the SELECT statement away. At some later date, you can say REFRESH TABLE and DB2 will first DELETE all of the existing rows, then run the SELECT statement to (you guessed it) repopulate the table.
- The DATA INITIALLY DEFERRED REFRESH IMMEDIATE clause creates a table that is also based on a SELECT statement. Once created, this type of table has to be refreshed once, and from then on DB2 will maintain the summary table in sync with the source table as changes are made to the latter.

In addition to the above, we shall also describe how one can create and populate your own summary tables using DB2 triggers.

IBM Implementation

A Summary Table is defined using a variation of the standard CREATE TABLE statement. Instead of providing an element list, one supplies a SELECT statement:

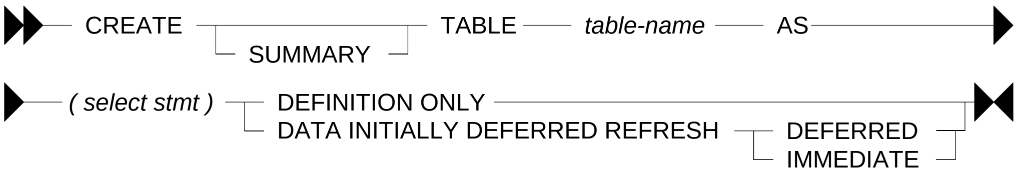


Figure 485, Summary Table DDL, Syntax Diagram

DDL Restrictions

Certain restrictions apply to the SELECT statement that is used to define the summary table. These restrictions get tighter as the summary table becomes more capable:

Definition Only Summary Tables

- Any valid SELECT statement is allowed.
- Every column selected must have a name.
- An ORDER BY is not allowed.
- Reference to a typed table or typed view is not allowed.

Refresh Deferred Summary Tables

All of the above restrictions apply, plus the following:

- Reference to a nickname or summary 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.

Refresh Immediate Summary 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.
- The SELECT must have a COUNT(*) or COUNT_BIG(*) column.
- Besides the COUNT, the only other column function supported is the SUM. 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 (including CUBE and ROLLUP) are not allowed.
- The HAVING clause is not allowed.
- The DISTINCT clause is not allowed.
- Non-deterministic functions are not allowed.
- Special registers are not allowed.

Definition Only Summary Tables

Definition-only summary tables are **not** true summary tables. They can not be refreshed, and once created, DB2 treats them as ordinary tables. Their usefulness comes from how one can use the CREATE SUMMARY TABLE syntax to quickly build a table that will accept the output of a query. Almost any kind of query is allowed.

Below is an example of a definition-only summary table. A simple "SELECT *" is used to define a new table that has the same columns as the source. When used this way, the summary table has the same capability as the "CREATE TABLE LIKE another" syntax that is available in DB2 for OS/390:

```
CREATE SUMMARY TABLE STAFF_COPY AS
(SELECT   *
 FROM     STAFF)
DEFINITION ONLY;
```

Figure 486, Definition-Only Summary Table DDL - simple SQL

Here is another way to write the above:

```
CREATE TABLE STAFF_COPY LIKE STAFF;
```

Figure 487, Create copy of table using LIKE syntax

The next example creates a tables based on the output of a GROUP BY statement:

```
CREATE SUMMARY TABLE STAFF_SUBSET AS
(SELECT   S1.ID
         ,S1.DEPT
         ,S1.COMM / 12 AS MONTY_COMM
         ,(SELECT MAX(S2.SALARY)
           FROM   STAFF S2
           WHERE  S2.DEPT = S1.DEPT)
          AS MAX_SALARY
 FROM     STAFF S1
 WHERE    S1.ID   > 10
   AND    S1.COMM >
         (SELECT MIN(COMM)
          FROM   STAFF))
DEFINITION ONLY
```

Figure 488, Definition-Only Summary Table DDL - complex SQL

Refresh Deferred Summary Tables

A summary table defined REFRESH DEFERRED can be periodically updated using the REFRESH TABLE command. Below is an example of a summary table that has one row per department in the STAFF table:

```
CREATE TABLE STAFF_NAMES AS
(SELECT   NAME
         ,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 NAME
 HAVING   COUNT(*) = 1)
DATA INITIALLY DEFERRED REFRESH DEFERRED
```

Figure 489, Refresh Deferred Summary Table DDL

Refresh Immediate Summary Tables

A summary table defined REFRESH IMMEDIATE is automatically maintained in sync with the source table by DB2. As with any summary table, it is defined by referring to a query. Below is a summary 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 490, Refresh Immediate Summary Table DDL

Below is a query that can use the above summary 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 491, Query that uses summary table (1 of 3)

The next query can also use the summary table. This time, the data returned from the summary table is validated using the 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 492, Query that uses summary table (2 of 3)

This last example uses the summary 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 493, Query that uses summary table (3 of 3)

Queries that don't use Summary 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 494, Query that doesn't use summary table (1 of 2)

The following query can't use the summary 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 495, Query that doesn't use summary table (2 of 2)

Usage Notes and Restrictions

- A summary 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 summary table.
- One can not load data into summary tables.
- One can not directly update summary tables.

To refresh a summary table, use either of the following commands:

```
REFRESH TABLE EMP_SUMMARY;
COMMIT;

SET INTEGRITY FOR EMP_SUMMARY IMMEDIATE CHECKED;
COMMIT;
```

Figure 496, Summary table refresh commands

Multi-table Summary Tables

Single-table summary tables save having to look at individual rows to resolve a GROUP BY. Multi-table summary 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 497, Multi-table Summary Table DDL

The following query is resolved using the above summary 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 498, Query that uses summary 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 499, DB2 generated query to use summary 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 Summary 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 500, Three-table Summary 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 501, Query that uses summary 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 502, DB2 generated query to use summary table

Indexes on Summary Tables

To really make things fly, one can add indexes to the summary table columns. DB2 will then use these indexes to locate the required data. Certain restrictions apply:

- Unique indexes are not allowed.
- The summary 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 503, Indexes for DPT_EMP_ACT_SUMRY summary 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 504, 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 505, Sample query that uses NUM_ROWS index

Don't forget to run RUNSTATS.

Roll Your Own

The REFRESH IMMEDIATE summary table provided by IBM supports the AVG, SUM, and COUNT column functions. In this section we will use triggers to define our own summary tables that will support all of the column functions. However, some of these triggers will be painfully inefficient, so one would not actually want to use them in practice.

> NOTE: Unlike with the IBM defined summary tables, the follow summary tables are not known to the optimizer. To use them, you have to explicitly query them.

Inefficient Triggers

Below are two tables. The one on the left is identical to the sample STAFF table that IBM supplies. The one on the right, called STAFF_DEPT, will contain a summary of the data in the STAFF table (by department).

```
SOURCE TABLE                            SUMMARY TABLE
==============================          ==============================
CREATE TABLE FRED.STAFF                 CREATE TABLE FRED.STAFF_DEPT
(ID       SMALLINT     NOT NULL         (DEPT          SMALLINT
,NAME     VARCHAR(9)                    ,COUNT_ROWS    SMALLINT
,DEPT     SMALLINT                      ,SUM_SALARY    DECIMAL(9,2)
,JOB      CHAR(5)                       ,AVG_SALARY    DECIMAL(7,2)
,YEARS    SMALLINT                      ,MAX_SALARY    DECIMAL(7,2)
,SALARY   DECIMAL(7,2)                  ,MIN_SALARY    DECIMAL(7,2)
,COMM     DECIMAL(7,2)                  ,STD_SALARY    DECIMAL(7,2)
,PRIMARY KEY (ID))                      ,VAR_SALARY    DOUBLE
                                        ,LAST_CHANGE   TIMESTAMP)
```

Figure 506, Source and Summary table DDL

Triggers will be used to automatically update the STAFF_DEPT table as the STAFF data changes. However, these triggers are **not** going to be very efficient. Every time a change is made to the STAFF table, the invoked trigger will delete the existing row (if any) from the STAFF_DEPT table, and then insert a new one (if possible). Three triggers are required, one each for an insert, update, and delete.

```
CREATE TRIGGER FRED.STAFF_INS
AFTER INSERT ON FRED.STAFF
REFERENCING NEW AS NNN
FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
   -----------------------------------------
   ---  DELETE OLD DEPT ROW             ---
   -----------------------------------------
   DELETE
   FROM     FRED.STAFF_DEPT
   WHERE    DEPT      = NNN.DEPT
      OR   (DEPT     IS NULL
     AND    NNN.DEPT IS NULL);
   -----------------------------------------
   ---  INSERT NEW DEPT ROW             ---
   -----------------------------------------
   INSERT
   INTO     FRED.STAFF_DEPT
   SELECT   DEPT
           ,COUNT(*)          AS COUNT_ROWS
           ,SUM(SALARY)       AS SUM_SALARY
           ,AVG(SALARY)       AS AVG_SALARY
```

Figure 507, INSERT Trigger, part 1 of 2

```
           ,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     FRED.STAFF
   WHERE    DEPT      = NNN.DEPT
      OR   (DEPT     IS NULL
     AND    NNN.DEPT IS NULL)
   GROUP BY DEPT;
END
```

Figure 508, INSERT Trigger, part 2 of 2

```
CREATE TRIGGER FRED.STAFF_UPD
AFTER
UPDATE OF SALARY, DEPT, COMM ON FRED.STAFF
REFERENCING OLD AS OOO
            NEW AS NNN
FOR EACH ROW
MODE DB2SQL
BEGIN ATOMIC
   -----------------------------------------
   ---  DELETE OLD DEPT ROW(S)           ---
   -----------------------------------------
   DELETE
   FROM     FRED.STAFF_DEPT
   WHERE    DEPT      = NNN.DEPT
      OR    DEPT      = OOO.DEPT
      OR   (DEPT     IS NULL
     AND   (NNN.DEPT IS NULL
      OR    OOO.DEPT IS NULL));
   -----------------------------------------
   ---  INSERT NEW DEPT ROW(S)           ---
   -----------------------------------------
   INSERT
   INTO     FRED.STAFF_DEPT
   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     FRED.STAFF
   WHERE    DEPT      = NNN.DEPT
      OR    DEPT      = OOO.DEPT
      OR   (DEPT     IS NULL
     AND   (NNN.DEPT IS NULL
      OR    OOO.DEPT IS NULL))
   GROUP BY DEPT;
END
```

Figure 509, UPDATE Trigger

```
CREATE TRIGGER FRED.STAFF_DEL
AFTER
DELETE ON FRED.STAFF
REFERENCING OLD AS OOO
FOR EACH ROW
MODE DB2SQL
BEGIN ATOMIC
   -----------------------------------------
   ---  DELETE OLD DEPT ROW              ---
   -----------------------------------------
```

Figure 510, DELETE Trigger, part 1 of 2

```
   DELETE
   FROM     FRED.STAFF_DEPT
   WHERE    DEPT      = OOO.DEPT
      OR   (DEPT     IS NULL
     AND    OOO.DEPT IS NULL);
   -----------------------------------------
   ---  INSERT NEW DEPT ROW              ---
   -----------------------------------------
   INSERT
   INTO     FRED.STAFF_DEPT
   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     FRED.STAFF
   WHERE    DEPT      = OOO.DEPT
      OR   (DEPT     IS NULL
     AND    OOO.DEPT IS NULL)
   GROUP BY DEPT;
END
```

Figure 511, DELETE Trigger, part 2 of 2

Efficiency (not)

The above triggers can be extremely expensive to run, especially when there are many rows in the STAFF table for a particular department. This is because every time a row is changed, all of the rows in the STAFF table in the same department are scanned in order to make the new row in STAFF_DEPT.

Also, the above triggers are invoked once per row changed, so a mass update to the STAFF table (e.g. add 10% to all salaries in a given department) will result in the invoked trigger scanning all of the rows in the impacted department(s) multiple times.

Notes

- The above CREATE TRIGGER statements are shown without a statement terminator. The "!" (exclamation mark) was used, but because this is non-standard, it was removed from the sample code. The semi-colon can not be used, because these triggers contain ATOMIC SQL statements.
- One advantage of the above trigger design is that one can include just about anything that comes to mind in the summary table. More efficient triggers (see below), and the IBM summary tables are more restrictive.
- The above STAFF_DEPT table lacks a primary key because the DEPT value can be null. It might be prudent to create a unique index on this column.
- Unlike with the IBM summary tables, there is no way to prevent users from directly updating the STAFF_DEPT table, except by using the standard DB2 security setup.

Initial Data Population

If the STAFF table already has data when the triggers are defined, then the following INSERT has to be run to populate the STAFF_DATA table:

```
INSERT
INTO     FRED.STAFF_DEPT
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     FRED.STAFF
GROUP BY DEPT;
```

Figure 512, Initial population of STAFF_DATA table

Efficient Triggers

In this section we will use triggers that, unlike those shown above, do **not** scan all of the rows in the department, every time a STAFF table row is changed. In order to make the code used easier to understand, both the DEPT and SALARY fields will be defined as not null.

```
SOURCE TABLE                      SUMMARY TABLE
==============================    ====================================
CREATE TABLE FRED.STAFF           CREATE TABLE FRED.STAFF_DEPT
(ID      SMALLINT     NOT NULL    (DEPT          SMALLINT     NOT NULL
,NAME    VARCHAR(9)               ,COUNT_ROWS    SMALLINT     NOT NULL
,DEPT    SMALLINT     NOT NULL    ,SUM_SALARY    DECIMAL(9,2) NOT NULL
,JOB     CHAR(5)                  ,AVG_SALARY    DECIMAL(7,2) NOT NULL
,YEARS   SMALLINT                 ,MAX_SALARY    DECIMAL(7,2) NOT NULL
,SALARY  DECIMAL(7,2) NOT NULL    ,MIN_SALARY    DECIMAL(7,2) NOT NULL
,COMM    DECIMAL(7,2)             ,#ROWS_CHANGED INTEGER      NOT NULL
,PRIMARY KEY (ID))                ,#CHANGING_SQL INTEGER      NOT NULL
                                  ,LAST_CHANGE   TIMESTAMP    NOT NULL
                                  ,CREATE_DEPT   TIMESTAMP    NOT NULL
                                  ,PRIMARY KEY (DEPT))
```

Figure 513, Source and Summary table DDL

> NOTE: Having an index on DEPT (above) will enable some of the triggered SQL statements to work much faster.

The above STAFF_DEPT table differs from the previous in the following ways:

- There are no Standard Deviation and Variance fields. These would require scanning all of the rows in the department in order to recalculate, so have been removed.
- There is a #ROWS_CHANGED field that keeps a count of the number of times that rows in a given department have been insert, updated, or deleted.
- There is a #CHANGING_SQL field that keeps a count of the number of SQL statements that have impacted rows in a given department. A SQL statement that updates multiple rows only gets counted once.
- There is a CREATE_DEPT field that records when the department was first created.

The above three new fields can exist in this summary table, but not the previous, because the triggers defined below do not remove a row (for a department) until there are no more related rows in the STAFF table. Therefore, historical information can be maintained over time.

Insert Trigger

This trigger, which is invoked for each row inserted, does two things:

- If no row exists (for the department) in STAFF_DEPT, insert a placeholder row. Some field values will be over-ridden in the following update, but others are important because they begin a sequence.
- Update the STAFF_DEPT row for the department, even if the row was just inserted. All fields are updated except CREATE_DEPT. Also, if the update stmt has the same CURRENT TIMESTAMP as the prior insert, the #CHANGING_SQL field is not changed.

```
CREATE TRIGGER FRED.STAFF_INS
AFTER INSERT ON FRED.STAFF
REFERENCING NEW AS NNN
FOR EACH ROW
MODE DB2SQL
BEGIN ATOMIC
   -----------------------------------------
   ---  INSERT IF NEW DEPT               ---
   -----------------------------------------
   INSERT
   INTO   FRED.STAFF_DEPT
   SELECT NNN.DEPT
         ,CAST(0 AS SMALLINT) AS COUNT_ROWS
         ,CAST(0 AS SMALLINT) AS SUM_SALARY
         ,CAST(0 AS SMALLINT) AS AVG_SALARY
         ,NNN.SALARY          AS MAX_SALARY
         ,NNN.SALARY          AS MIN_SALARY
         ,CAST(0 AS INTEGER)  AS #ROWS_CHANGED
         ,CAST(1 AS INTEGER)  AS #CHANGING_SQL
         ,CURRENT TIMESTAMP   AS LAST_CHANGE
         ,CURRENT TIMESTAMP   AS CREATE_DEPT
   FROM   FRED.STAFF
   WHERE  ID = NNN.ID
     AND  NNN.DEPT NOT IN
         (SELECT DEPT
          FROM   FRED.STAFF_DEPT);
   -----------------------------------------
   ---  UPDATE DEPT ROW, INCREMENT       ---
   -----------------------------------------
   UPDATE FRED.STAFF_DEPT
   SET    COUNT_ROWS    = COUNT_ROWS + 1
         ,SUM_SALARY    = SUM_SALARY + NNN.SALARY
         ,AVG_SALARY    =(SUM_SALARY + NNN.SALARY)
                         /(COUNT_ROWS + 1)
         ,MAX_SALARY    = CASE
                             WHEN NNN.SALARY <= MAX_SALARY
                             THEN MAX_SALARY
                             ELSE NNN.SALARY
                          END
         ,MIN_SALARY    = CASE
                             WHEN NNN.SALARY >= MIN_SALARY
                             THEN MIN_SALARY
                             ELSE NNN.SALARY
                          END
         ,#ROWS_CHANGED = #ROWS_CHANGED + 1
         ,#CHANGING_SQL = CASE
                             WHEN LAST_CHANGE = CURRENT TIMESTAMP
                             THEN #CHANGING_SQL
                             ELSE #CHANGING_SQL + 1
                          END
         ,LAST_CHANGE   = CURRENT TIMESTAMP
   WHERE DEPT           = NNN.DEPT;
END
```

Figure 514, INSERT Trigger

Doing Multiple Actions per Row

All of the triggers in this chapter are defined FOR EACH ROW, which means that they are invoked once per row altered (as opposed to once per statement). All of the triggers can do more than one action (e.g. an INSERT, and then an UPDATE).

When one wants multiple actions to be done after a row is changed, then one **must** put all of the actions into a single trigger. If the actions are defined in separate triggers, then first one will be done, for all rows, then the action other, for all rows. If the initiating action was a multi-row UPDATE (or INSERT, or DELETE) statement, then doing the two triggered actions, one after the other (on a row by row basis) is quite different from doing one (for all rows) then the other (for all rows).

Update Trigger

This trigger, which is invoked for each row updated, does five things:

- If no row exists (for the department) in STAFF_DEPT, insert a placeholder row. Some field values will be over-ridden in the following update, but others are important because they begin a sequence.
- If the department value is being changed, and there is only one row currently in the department, delete the row. This delete must be done before the update of the old row, because otherwise this update will get a divide-by-zero error, when trying to update the AVG salary.
- If the department value is not being changed, update the existing department row.
- If the department value is being changed, update the new department row.
- If the department value is being changed, update the old department row. If the MAX or MIN values were max or min (for the old department), but no longer are, then do a sub-query to find the new max or min. These sub-queries can return a null value if a multi-row update has changed all department values, so the COALESCE function is used to convert the null to zero. The user will never see the (invalid) zero, as the row will get deleted later in the trigger.

```
CREATE TRIGGER FRED.STAFF_UPD
AFTER UPDATE OF SALARY, DEPT, COMM ON FRED.STAFF
REFERENCING OLD AS OOO
            NEW AS NNN
FOR EACH ROW
MODE DB2SQL
BEGIN ATOMIC
   -----------------------------------------
   ---  INSERT IF NEW DEPT               ---
   -----------------------------------------
   INSERT
   INTO   FRED.STAFF_DEPT
   SELECT NNN.DEPT
         ,CAST(0 AS SMALLINT) AS COUNT_ROWS
         ,CAST(0 AS SMALLINT) AS SUM_SALARY
         ,CAST(0 AS SMALLINT) AS AVG_SALARY
         ,NNN.SALARY          AS MAX_SALARY
         ,NNN.SALARY          AS MIN_SALARY
         ,CAST(0 AS INTEGER)  AS #ROWS_CHANGED
         ,CAST(1 AS INTEGER)  AS #CHANGING_SQL
         ,CURRENT TIMESTAMP   AS LAST_CHANGE
         ,CURRENT TIMESTAMP   AS CREATE_DEPT
```

Figure 515, UPDATE Trigger, Part 1 of 3

```
FROM   FRED.STAFF
WHERE  ID = NNN.ID
  AND  NNN.DEPT NOT IN
      (SELECT DEPT
       FROM   FRED.STAFF_DEPT);
-----------------------------------------
---  DELETE IF LAST ROW IN DEPT       ---
-----------------------------------------
DELETE
FROM   FRED.STAFF_DEPT
WHERE  NNN.DEPT     <> OOO.DEPT
  AND  DEPT          = OOO.DEPT
  AND  COUNT_ROWS    = 1;
-----------------------------------------
---  UPDATE DEPT, WHEN NOT CHANGED    ---
-----------------------------------------
UPDATE FRED.STAFF_DEPT
SET    SUM_SALARY    = SUM_SALARY + NNN.SALARY - OOO.SALARY
      ,AVG_SALARY    =(SUM_SALARY + NNN.SALARY - OOO.SALARY)
                     / COUNT_ROWS
      ,MAX_SALARY    = CASE
                          WHEN NNN.SALARY >= MAX_SALARY
                          THEN NNN.SALARY
                          ELSE (SELECT MAX(SALARY)
                                FROM   FRED.STAFF
                                WHERE  DEPT = NNN.DEPT)
                       END
      ,MIN_SALARY    = CASE
                          WHEN NNN.SALARY <= MIN_SALARY
                          THEN NNN.SALARY
                          ELSE (SELECT MIN(SALARY)
                                FROM   FRED.STAFF
                                WHERE  DEPT = NNN.DEPT)
                       END
      ,#ROWS_CHANGED = #ROWS_CHANGED + 1
      ,#CHANGING_SQL = CASE
                          WHEN LAST_CHANGE = CURRENT TIMESTAMP
                          THEN #CHANGING_SQL
                          ELSE #CHANGING_SQL + 1
                       END
      ,LAST_CHANGE   = CURRENT TIMESTAMP
WHERE  NNN.DEPT      = OOO.DEPT
  AND  NNN.DEPT      = DEPT;
-----------------------------------------
---  UPDATE NEW, WHEN DEPT CHANGED    ---
-----------------------------------------
UPDATE FRED.STAFF_DEPT
SET    COUNT_ROWS    = COUNT_ROWS + 1
      ,SUM_SALARY    = SUM_SALARY + NNN.SALARY
      ,AVG_SALARY    =(SUM_SALARY + NNN.SALARY)
                     /(COUNT_ROWS + 1)
      ,MAX_SALARY    = CASE
                          WHEN NNN.SALARY <= MAX_SALARY
                          THEN MAX_SALARY
                          ELSE NNN.SALARY
                       END
      ,MIN_SALARY    = CASE
                          WHEN NNN.SALARY >= MIN_SALARY
                          THEN MIN_SALARY
                          ELSE NNN.SALARY
                       END
      ,#ROWS_CHANGED = #ROWS_CHANGED + 1
      ,#CHANGING_SQL = CASE
                          WHEN LAST_CHANGE = CURRENT TIMESTAMP
                          THEN #CHANGING_SQL
                          ELSE #CHANGING_SQL + 1
```

Figure 516, UPDATE Trigger, Part 2 of 3

```
                          END
         ,LAST_CHANGE   = CURRENT TIMESTAMP
   WHERE  NNN.DEPT     <> OOO.DEPT
     AND  NNN.DEPT      = DEPT;
   -----------------------------------------
   ---  UPDATE OLD, WHEN DEPT CHANGED    ---
   -----------------------------------------
   UPDATE FRED.STAFF_DEPT
   SET    COUNT_ROWS    = COUNT_ROWS - 1
         ,SUM_SALARY    = SUM_SALARY - OOO.SALARY
         ,AVG_SALARY    = (SUM_SALARY - OOO.SALARY)
                         /(COUNT_ROWS - 1)
         ,MAX_SALARY    = CASE
                             WHEN OOO.SALARY < MAX_SALARY
                             THEN MAX_SALARY
                             ELSE (SELECT COALESCE(MAX(SALARY),0)
                                   FROM   FRED.STAFF
                                   WHERE  DEPT = OOO.DEPT)
                          END
         ,MIN_SALARY    = CASE
                             WHEN OOO.SALARY > MIN_SALARY
                             THEN MIN_SALARY
                             ELSE (SELECT COALESCE(MIN(SALARY),0)
                                   FROM   FRED.STAFF
                                   WHERE  DEPT = OOO.DEPT)
                          END
         ,#ROWS_CHANGED = #ROWS_CHANGED + 1
         ,#CHANGING_SQL = CASE
                             WHEN LAST_CHANGE = CURRENT TIMESTAMP
                             THEN #CHANGING_SQL
                             ELSE #CHANGING_SQL + 1
                          END
         ,LAST_CHANGE   = CURRENT TIMESTAMP
   WHERE  NNN.DEPT     <> OOO.DEPT
     AND  OOO.DEPT      = DEPT;
END
```

Figure 517, UPDATE Trigger, Part 3 of 3

Delete Trigger

This trigger, which is invoked for each row deleted, does two things:

- If there is current only one row for the department, then delete this row. This delete must be done before the following update, because otherwise this update would get a divide-by-zero error when changing the AVG value.
- Update the department row (if it is still there). If the MAX or MIN values were max or min (for the old department), but no longer are, then do a sub-query to find the new max or min.

```
CREATE TRIGGER FRED.STAFF_DEL
AFTER DELETE ON FRED.STAFF
REFERENCING OLD AS OOO
FOR EACH ROW
MODE DB2SQL
BEGIN ATOMIC
   -----------------------------------------
   ---  DELETE IF LAST ROW IN DEPT       ---
   -----------------------------------------
   DELETE
   FROM   FRED.STAFF_DEPT
   WHERE  DEPT         = OOO.DEPT
     AND  COUNT_ROWS   = 1;
```

Figure 518, DELETE Trigger, part 1 of 2

```
   -----------------------------------------
   ---  UPDATE OLD DEPT ROW, DECREMENT   ---
   -----------------------------------------
   UPDATE FRED.STAFF_DEPT
   SET    COUNT_ROWS   = COUNT_ROWS - 1
         ,SUM_SALARY   = SUM_SALARY - OOO.SALARY
         ,AVG_SALARY   = (SUM_SALARY - OOO.SALARY)
                        /(COUNT_ROWS - 1)
         ,MAX_SALARY   = CASE
                            WHEN OOO.SALARY < MAX_SALARY
                            THEN MAX_SALARY
                            ELSE (SELECT COALESCE(MAX(SALARY),0)
                                  FROM   FRED.STAFF
                                  WHERE  DEPT = OOO.DEPT)
                         END
         ,MIN_SALARY   = CASE
                            WHEN OOO.SALARY > MIN_SALARY
                            THEN MIN_SALARY
                            ELSE (SELECT COALESCE(MIN(SALARY),0)
                                  FROM   FRED.STAFF
                                  WHERE  DEPT = OOO.DEPT)
                         END
         ,#ROWS_CHANGED = #ROWS_CHANGED + 1
         ,#CHANGING_SQL = CASE
                             WHEN LAST_CHANGE = CURRENT TIMESTAMP
                             THEN #CHANGING_SQL
                             ELSE #CHANGING_SQL + 1
                          END
         ,LAST_CHANGE   = CURRENT TIMESTAMP
   WHERE  OOO.DEPT      = DEPT;
END
```

Figure 519, DELETE Trigger, part 2 of 2

Efficiency

The above triggers are all efficient in that, for almost all situations, a change to the STAFF table requires nothing more than a corresponding change to the STAFF_DEPT table. However, if a row is being removed from a department (either because of an update or a delete), and the row has the current MAX or MIN salary value, then the STAFF table has to be queried to get a new max or min. Having an index on the DEPT column will enable this query to run quite quickly.

Notes

- Unlike with the IBM summary tables, there is no way (above) to stop users directly updating the STAFF_DEPT table, except by using the standard DB2 security setup.
- Also, unlike an IBM summary table, the STAFF_DEPT table can not maintained by a LOAD, nor updated using the refresh statement.
- 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 Column

Imagine that one an INVOICE table, and 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 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, can be implemented in DB2. In this chapter we will describe how to do it.

Usage Notes

One can define a column in a DB2 table as an "identity column". This column, which must be numeric, 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:

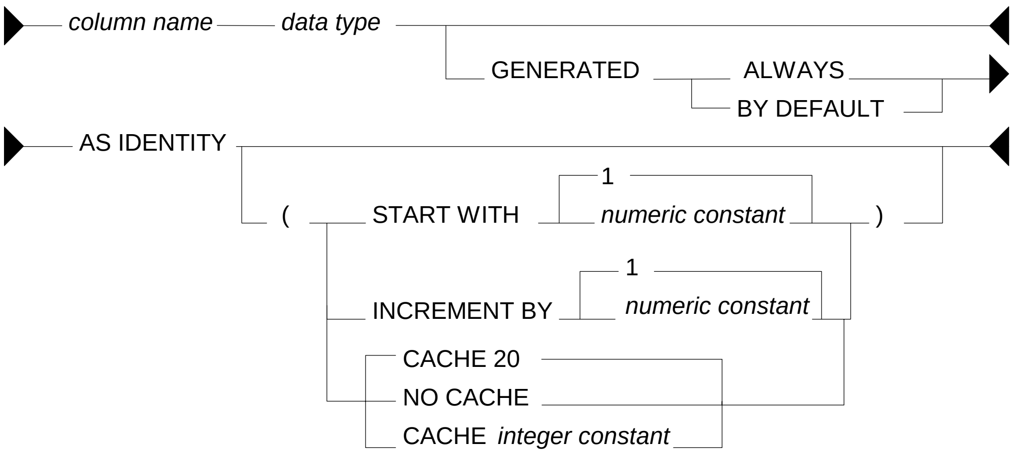


Figure 520, Identity Column syntax

Below is the DDL for a 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 CACHE)
,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 521, 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 522, 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 523, Invoice table, after inserts

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).

Various rules and restrictions apply to identity columns:

- There can only be one 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.
- The identity column value is generated before any BEFORE triggers are applied. Use a trigger transition variable to see the value.
- Unless otherwise specified when the table is created, the first row inserted will have the value one, the next row the value two, and so on.
- 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.

Restart the Identity Start Value

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. One way to improve things without actually having to fix any difficult business problems is to simply set the invoice# identity column value to a higher number. This is done using the ALTER TABLE ... RESTART command:

```
ALTER TABLE  INVOICE_DATA
ALTER COLUMN INVOICE# RESTART WITH 1000;
```

Figure 524, 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 525, 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
    1001   11/25/2001   YYY                  123        100   10.00
```

Figure 526, Invoice table, after second inserts

Restart Usage Notes

- One can only alter the start number with the restart command. One cannot change either the direction or the increment value. To fix these, the table has to drop and recreated.
- Resetting 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 table getting duplicate values - if there are existent rows already in the table, and no suitable unique index.

Cache Usage

As identity column values are assigned to users, their usage is recorded in the DB2 log. One way to improve performance is to give each person doing an insert a block of values, which will be used in subsequent inserts - if any. By default, each user doing inserts gets a block of twenty contiguous values whenever they do their first insert. After they have inserted twenty rows, DB2 then assigns them another twenty values.

Optionally, one can define the identity column as having no cache, which means that each individual insert gets given another (logged) value by DB2. One can also set the cache size to any value that you wish, but you cannot alter it once the table is defined.

Cache Usage Notes

- If a cache is used, there will almost certainly be gaps in the sequence of identity values. This will occur because not every user will always use all of the values they are given.
- If a cache is used, then the sequence of identity column values may not represent the sequence with which rows were actually inserted into the table. To illustrate, imagine that the cache size is ten, and that there are two concurrent users. The first user to do an insert will be given ten values, while the second user will get the next ten values. If the first user pauses between their first and subsequent inserts, while the second user starts doing inserts, then subsequent inserts done by the first user (the next nine only) will be assigned identity values that are lower than what are already in the table.
- The cache is strictly a performance feature. If one does not use a cache, then each insert has to write a record to the DB2 log, which will probably add about five milliseconds to each insert. Use it if you cannot afford the delay, and if you don't desire the sequence of identity column values to represent the order in which the rows were inserted.

Gaps in the Sequence

If an identity column is generated "always", there is absolutely no way to avoid getting gaps in the sequence of numbers, even when no cache is used, **-** if an insert can subsequently be rolled out instead of being committed. An identity value is assigned to each individual row as it is inserted, and it cannot be reused if the insert is subsequently undone. So if the application aborts before the insert is committed, the value is gone for good.

In theory, one can redo a failed insert, if the identity column was defined as being generated "by default" instead of being generated "always". Then, assuming that one knows what identity column value to use, one can reinsert the lost row. Below is an example of doing this:

```
CREATE TABLE CUSTOMERS
(CUST#          INTEGER                   NOT NULL
                GENERATED BY DEFAULT 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 (1,'FRED','XXX');

SELECT   *                                     <<< ANSWER
FROM     CUSTOMERS                                 ===================
ORDER BY 1;                                        CUST#  CNAME  CTYPE
                                                   -----  -----  -----
COMMIT;                                                1  FRED   XXX
```

Figure 527, Overriding the default identity value

> NOTE: Explicitly inserting or updating an identity column value that is generated by default has absolutely no impact on the sequence of values being handed out by DB2.

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 528, 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 529, 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 530, 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 531, 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 532, 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 533, IDENTITY_VAL_LOCAL usage in SET statement

Roll Your Own - Supporting Multi-row Inserts

Imagine that one needed to be able to find out what identity column values were generated when the user inserted multiple rows in a single statement. The IDENTITY_VAL_LOCAL function cannot be used to do this, but one can define a trigger and an additional table to do the job. To illustrate, below is an ordinary data table with an identity column:

```
CREATE TABLE INVOICE_DATA
(INVOICE#       INTEGER                   NOT NULL
                GENERATED ALWAYS AS IDENTITY
                   (START WITH 100
                   ,INCREMENT BY 1
                   ,CACHE 5)
,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 534, Record multi-row inserts - data table

The next table will be used to record every insert done to the above data table:

```
CREATE TABLE CURRENT_INVOICE
(INVOICE#       INTEGER                   NOT NULL
,CURRENT_TS     TIMESTAMP                 NOT NULL
,USERID         VARCHAR(128)              NOT NULL
,PRIMARY KEY   (USERID, INVOICE#));
```

Figure 535, Record multi-row inserts - record table

The following trigger is run once per insert into the above data table. In effect, it will record the invoice# values that were most recently inserted into the data table - for a given user. In particular, it does two things:

- It first deletes all existent rows in the second, record table - for the current user.
- It then inserts a new row for each row that was inserted into the data table.

Note that because the trigger definition uses atomic SQL, the create statement requires a non-standard statement delimiter, in this case, an exclamation mark:

```
--#SET DELIMITER !
CREATE TRIGGER INVOICE_INSERT
AFTER  INSERT
ON     INVOICE_DATA
REFERENCING NEW_TABLE AS NEW_ROWS
FOR EACH STATEMENT
MODE DB2SQL
BEGIN ATOMIC
   DELETE
   FROM   CURRENT_INVOICE
   WHERE  USERID      = USER;
   INSERT
   INTO   CURRENT_INVOICE
   SELECT INVOICE#
         ,CURRENT TIMESTAMP
         ,USER
   FROM   NEW_ROWS;
END!
--#SET DELIMITER ;
COMMIT;
```

Figure 536, Record multi-row inserts - record trigger

Below is an example of the above tables and trigger in action. Three rows are inserted into the data table, which results three rows also being inserted into the second table:

```
INSERT INTO INVOICE_DATA
(SALE_DATE,CUSTOMER_ID,PRODUCT_ID,QUANTITY,PRICE)
VALUES ('2000-11-23','DEF',123,100,10)
      ,('2000-11-23','XXX',123,100,10)
      ,('2000-11-23','ZZZ',123,100,10);
COMMIT;

SELECT   MIN(INVOICE#) AS MIN_INV
        ,MAX(INVOICE#) AS MAX_INV
        ,COUNT(*)      AS #ROWS       ANSWER
        ,CURRENT_TS                   ================================
FROM     CURRENT_INVOICE              MIN_INV MAX_INV #ROWS CURRENT_TS
WHERE    USERID = USER                ------- ------- ----- ----------
GROUP BY CURRENT_TS;                      101     103     3 2001-etc..
```

Figure 537, Record multi-row inserts - usage example

Given the cost of the extra inserts, the above solution is far from ideal. Another option would be to add the user and current timestamp fields to the INVOICE table. Then, to get the most recent rows for a given user, one could run the following query:

```
SELECT   MIN(INVOICE#) AS MIN_INV
        ,MAX(INVOICE#) AS MAX_INV
        ,COUNT(*)      AS #ROWS
        ,CURRENT_TS
FROM     INVOICE_DATA
WHERE    USERID      = USER
  AND    CURRENT_TS >= ANY
                      (SELECT MAX(CURRENT_TS)
                       FROM   INVOICE_DATA
                       WHERE  USERID = USER)
GROUP BY CURRENT_TS;
```

Figure 538, Track multi-row inserts - by adding fields to data table

None of the above SQL will work if the USER special register is not unique to each user on the system. One needs some way to know one user from another.

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 539, 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:

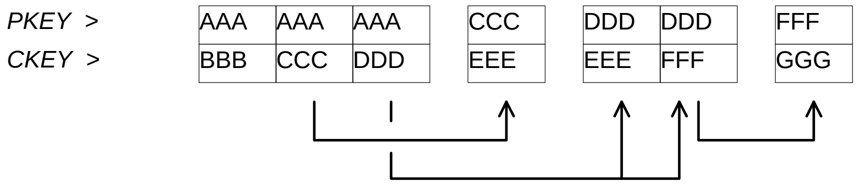
```
WITH PARENT (PKEY, CKEY) AS                   ANSWER
(SELECT PKEY, CKEY                           =========      PROCESSING
 FROM   HIERARCHY                            PKEY CKEY       SEQUENCE
 WHERE  PKEY = 'AAA'                         ---- ----      ==========
 UNION ALL                                   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 540, 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:



PKEY >	AAA	AAA	AAA	CCC	DDD	DDD	FFF
CKEY >	BBB	CCC	DDD	EEE	EEE	FFF	GGG

Figure 541, 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 202.

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 542, 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 193. 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 543, 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 202 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 544, 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 545, 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 546, 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 547, 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 548, 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 549, 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 to 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 550, 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

The output from one recursive expression can be used as input to second recursive expression in the same SQL statement. This means that one can expand multiple hierarchies in a single statement. 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.

To illustrate this trick, we shall get a list of all the values in the HIERARCHY table that are in a path (of items) that is at least four levels deep. This will require two recursive passes of the table. In the first we shall work our way down to the fourth level. In the second pass we will use the fourth-level value(s) as a seed and then work our way back up the hierarchy to find the relevant parents (direct and indirect).

```
WITH TEMP1 (CKEY, LVL) AS                ANSWER           AAA
(SELECT DISTINCT PKEY, 1                 ========          |
 FROM   HIERARCHY                        CKEY LVL    +-----+-----+
 WHERE  PKEY = 'AAA'                     ---- ---    |     |     |
 UNION ALL                               GGG    4   BBB   CCC   DDD
 SELECT C.CKEY, P.LVL +1                 FFF    3          |     |
 FROM   HIERARCHY C                      DDD    2          +-+ +-+--+
       ,TEMP1     P                      AAA    1            | |    |
 WHERE  P.CKEY = C.PKEY                                      EEE   FFF
   AND  P.LVL  < 4                                                  |
)                                                                   |
,TEMP2 (CKEY, LVL) AS                                              GGG
(SELECT CKEY, LVL
 FROM   TEMP1
 WHERE  LVL = 4
 UNION ALL
 SELECT C.PKEY, D.LVL -1
 FROM   HIERARCHY C
       ,TEMP2     D
 WHERE  D.CKEY = C.CKEY
)
SELECT *
FROM   TEMP2;
```

Figure 551, Find all values in paths that are at least four deep

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 552, 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 553, 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 554, 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 555, 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 resolve a hierarchy out 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          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  |EEE  | 44|                           EEE   FFF
|FFF  |  $34|    |DDD  |FFF  |  5|                                  |
|GGG  |  $44|    |FFF  |GGG  |  5|                                  |
+-----------+    +---------------+                                 GGG
```

Figure 556, 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 202 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 557, 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 202 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 558, 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

For better or worse, one occasionally encounters recursive hierarchical data structures. This section describes how to write recursive SQL statements that can process such systems 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 Database

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 559, Recursive Hierarchy - Sample Table and Layout

Below is the DML that was used to create the above system. Note that the ">" character is not allowed in either key column. This was done because this character will be used as a delimiter in some of the following SQL.

```
CREATE TABLE TROUBLE
(PKEY     CHAR(03)     NOT NULL
,CKEY     CHAR(03)     NOT NULL
,CONSTRAINT TBX1 PRIMARY KEY(PKEY, CKEY)
,CONSTRAINT TBC1 CHECK (PKEY <> CKEY)
,CONSTRAINT TBC2 CHECK (LOCATE('>',PKEY)=0)
,CONSTRAINT TBC3 CHECK (LOCATE('>',CKEY)=0));

CREATE UNIQUE INDEX TBLE_X1 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 560, 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 found using simpler SQL than what is given below. But a loop that does not include the beginning object (e.g. AAA points to BBB, which points to CCC, which points to BBB) requires the somewhat complicated SQL that is used here.

Stop After "n" Levels

Find all the children of AAA. In order to avoid running forever, stop after four levels.

```
WITH PARENT (CKEY, LVL) AS                      ANSWER     TROUBLE
(SELECT DISTINCT PKEY, 0                        ========   +---------+
 FROM   TROUBLE                                 CKEY LVL   |PKEY|CKEY|
 WHERE  PKEY = 'AAA'                            ---- ---   |----|----|
 UNION ALL                                      AAA    0   |AAA |BBB |
 SELECT C.CKEY, P.LVL +1                        BBB    1   |AAA |CCC |
 FROM   TROUBLE   C                             CCC    1   |AAA |DDD |
       ,PARENT    P                             DDD    1   |CCC |EEE |
 WHERE  P.CKEY  = C.PKEY                        EEE    2   |DDD |AAA |
   AND  P.LVL+1 < 4                             AAA    2   |DDD |FFF |
)                                               EEE    2   |DDD |EEE |
SELECT CKEY, LVL                                FFF    2   |FFF |GGG |
FROM   PARENT;                                  BBB    3   +---------+
                                                CCC    3
                                                DDD    3
                                                GGG    3
```

Figure 561, Stop Recursive SQL after "n" levels

In order for the above statement to get the right answer, we need to know before beginning how many valid dependent levels there are in the hierarchy. This information is then incorporated into the recursive part of the statement (see: P.LVI+1 < 4). If this information is not known, and we guess wrong, we may not find all children of AAA.

A more specific disadvantage of the above statement is that the list of children contains duplicates. These duplicates include not just those specific values that compose the infinite loop (i.e. AAA and DDD), but also any children of either of the above.

Stop After "n" Levels - Remove Duplicates

Get a distinct list of the children of AAA. Stop searching after four levels.

```
 WITH PARENT (CKEY, LVL) AS                                ANSWER
 (SELECT DISTINCT PKEY, 0                                  ============
  FROM   TROUBLE                                           CKEY LVL NUM
  WHERE  PKEY = 'AAA'                                      ---- --- ---
  UNION ALL                                                AAA    0   2
  SELECT C.CKEY, P.LVL +1                                  BBB    1   2
  FROM   TROUBLE   C                                       CCC    1   2
        ,PARENT    P                                       DDD    1   2
  WHERE  P.CKEY  = C.PKEY                                  EEE    2   2
    AND  P.LVL+1 < 4                                       AAA    2   1
 )                                                         GGG    3   1
 ,NO_DUPS (CKEY, LVL, NUM) AS
 (SELECT   CKEY, MIN(LVL), COUNT(*)
  FROM     PARENT
  GROUP BY CKEY
 )
 SELECT CKEY, LVL, NUM
 FROM   NO_DUPS;                        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 562, Stop Recursive SQL after "n" levels, Remove duplicates

The recursive part of the above statement is the same as the prior. What differs is the use of a second temporary table called NO_DUPS in which the duplicate rows found in the PARENT table are removed. The true level of a duplicated item is deemed to be lowest value found.

Note that two temporary tables are used above. The use of the second is arguably unnecessary because the grouping that it does could equally well be done in the final select. The code is written the way it is because it enables one to use the output from the GROUP BY in a join with another table (instead of just doing a plain select).

Stop After "n" Levels - Show Data Paths

Find all the children of AAA. For each child, display its relationship to AAA. In order to avoid running forever, stop after four levels.

```
 WITH PARENT (CKEY,LVL,PATH,LOC) AS         ANSWER
 (SELECT DISTINCT PKEY, 0                   ===========================
        ,VARCHAR(PKEY,20)                   CKEY LVL PATH           LOC
        ,0                                  ---- --- -------------- ---
  FROM   TROUBLE                            AAA    0 AAA              0
  WHERE  PKEY = 'AAA'                       BBB    1 AAA>BBB          0
  UNION ALL                                 CCC    1 AAA>CCC          0
  SELECT C.CKEY, P.LVL +1                   DDD    1 AAA>DDD          0
        ,P.PATH||'>'||C.CKEY                EEE    2 AAA>CCC>EEE      0
        ,LOCATE(C.CKEY||'>',P.PATH)         AAA    2 AAA>DDD>AAA      1
  FROM   TROUBLE   C                        EEE    2 AAA>DDD>EEE      0
        ,PARENT    P                        FFF    2 AAA>DDD>FFF      0
  WHERE  P.CKEY  = C.PKEY                   BBB    3 AAA>DDD>AAA>BBB  0
    AND  P.LVL+1 < 4                        CCC    3 AAA>DDD>AAA>CCC  0
 )                                          DDD    3 AAA>DDD>AAA>DDD  5
 SELECT CKEY, LVL, PATH, LOC                GGG    3 AAA>DDD>FFF>GGG  0
 FROM   PARENT;
```

Figure 563, Stop Recursive SQL after "n" levels, Show data paths

The PATH field above shows the relationship between each child object and the top-level parent of the hierarchy. In the top part of the select statement, this field is given the value AAA and is defined as VARCHAR. During subsequent recursive processing, the current child key is appended on the right. A ">" symbol is placed between each key value to enhance readability.

The LOC field indicates if the current child key value has been looked at before. It does this by using the LOCATE function to search the PATH field for the existence of the child key string. There are several important things to note here:

- The PATH value searched by the LOCATE function is not the one that you see displayed above. The above PATH field shows the result after the current child key has been appended. The PATH field searched is the one that was there before the append occurred. This difference does not affect the answer but it is worth remembering.
- One has to define the PATH field to be long enough to support the maximum number of levels in the hierarchy. For example, it is twenty bytes long above, which is enough to go down five layers. If you're not sure how many levels there are - make the field very long. Because it is a VARCHAR field, it only uses space when needed.
- The TROUBLE table was defined such that neither key field can contain the ">" character. This is not an issue here, but it will be in the next statement.

Stop After "n" Rows

This can be done using recursive SQL, but it is neither easy nor very efficient (unless a lot of care is taken). Use instead the ROW_NUMBER function (see page 58).

Find all Children, Ignore Data Loops

Find all the children of AAA. This time, don't use levels to avoid going forever.

```
WITH PARENT (CKEY,LVL,PATH) AS                ANSWER
(SELECT DISTINCT PKEY, 0                      ========================
       ,VARCHAR(PKEY,20)                      CKEY LVL PATH
 FROM   TROUBLE                               ---- --- --------------
 WHERE  PKEY = 'AAA'                          AAA    0 AAA
 UNION ALL                                    BBB    1 AAA>BBB
 SELECT C.CKEY, P.LVL +1                      CCC    1 AAA>CCC
       ,P.PATH||'>'||C.CKEY                   DDD    1 AAA>DDD
 FROM   TROUBLE   C                           EEE    2 AAA>CCC>EEE
       ,PARENT    P                           EEE    2 AAA>DDD>EEE
 WHERE  P.CKEY  = C.PKEY                      FFF    2 AAA>DDD>FFF
   AND  LOCATE(C.CKEY||'>',P.PATH)=0          GGG    3 AAA>DDD>FFF>GGG
)
SELECT CKEY, LVL, PATH
FROM   PARENT;
```

Figure 564, Recursive SQL, Ignore data loops (don't retrieve)

Observe the LOCATE predicate in the recursive part of the above SQL statement. This check will reject any row where the current child key value can be found in the related PATH field (i.e. the current row has already been processed). In the PATH field, the '>' is used as a delimiter to ensure that no two concatenated key values can have a combined string value that equates to the current child key. This trick only works because the table was defined as not allowing a '>' in either key field (see page 203 for DDL).

For most hierarchies that contain infinite loops, the above SQL is likely to be more efficient than something similar that uses a level-stop check. This is because the above statement will reject an unwanted row immediately it is encountered whereas the level-stop technique gets the bad along with the good then removes the former in subsequent processing.

Find all Children, Mark Data Loops

Find all children of AAA. Also, mark the first value in any infinitely looping data structure.

```
WITH PARENT (CKEY, LVL, PATH) AS              ANSWER
(SELECT DISTINCT PKEY, 0                      ========================
       ,VARCHAR(PKEY,20)                      CKEY LVL PATH
 FROM   TROUBLE                               ---- --- ---------------
 WHERE  PKEY = 'AAA'                          AAA    0 AAA
 UNION ALL                                    BBB    1 AAA>BBB
 SELECT CASE                                  CCC    1 AAA>CCC
        WHEN LOCATE(C.CKEY||'>',P.PATH)>0     DDD    1 AAA>DDD
             THEN CHAR('>>>',3)               EEE    2 AAA>CCC>EEE
        ELSE C.CKEY                           >>>    2 AAA>DDD>AAA
        END                                   EEE    2 AAA>DDD>EEE
       ,P.LVL +1                              FFF    2 AAA>DDD>FFF
       ,P.PATH||'>'||C.CKEY                   GGG    3 AAA>DDD>FFF>GGG
 FROM   TROUBLE   C
       ,PARENT    P
 WHERE  P.CKEY  = C.PKEY
)
SELECT CKEY, LVL, PATH
FROM   PARENT;
```

Figure 565, Recursive SQL, Mark data loops

Any row in the above answer set that has a '>>>' in the CKEY2 refers to an infinite loop. The CKEY column contains the first key in the loop and the PATH column contains a description of the loop. For all other rows, the CKEY2 field contains the current lowest-level child.

A CASE statement is used above to determine what value to put in the CKEY2 field. If the current child row has been processed before, the string '>>>' is used, else CKEY used. This key-value substitution is required in order to stop the statement from going forever.

Find all Data Loops - Only

A variation on the above SQL statement can be used to list just those rows that are part of a data loop (e.g. for database integrity checking). Simply add a single predicate to the last SELECT statement that only gets those rows where the CKEY is '>>>'.

```
WITH PARENT (CKEY, PATH) AS                 ANSWER         TROUBLE
(SELECT DISTINCT PKEY                       ============   +---------+
       ,VARCHAR(PKEY,20)                    PATH           |PKEY|CKEY|
 FROM   TROUBLE                             ------------   |----|----|
 WHERE  PKEY = 'AAA'                        AAA>DDD>AAA    |AAA |BBB |
 UNION ALL                                                 |AAA |CCC |
 SELECT CASE                                               |AAA |DDD |
        WHEN LOCATE(C.CKEY||'>',P.PATH)>0                  |CCC |EEE |
             THEN CHAR('>>>',3)                            |DDD |AAA |
        ELSE C.CKEY                                        |DDD |FFF |
        END                                                |DDD |EEE |
       ,P.PATH||'>'||C.CKEY                                |FFF |GGG |
 FROM   TROUBLE   C                                        +---------+
       ,PARENT    P
 WHERE  P.CKEY  = C.PKEY
)
SELECT PATH
FROM   PARENT
WHERE  CKEY = '>>>';
```

Figure 566, Recursive SQL, List data loops

Stop if Data Loops

Find in all the children of AAA. If a data-loop is encountered during processing, stop the SQL statement and return a SQL error. Use the RAISE_ERROR function to do this.

```
WITH PARENT                                       ANSWER   TROUBLE
(CKEY, LVL, PATH) AS                              ======   +---------+
(SELECT DISTINCT                                  CKEY...  |PKEY|CKEY|
        PKEY                                      ----     |----|----|
       ,0                                         <error>  |AAA |BBB |
       ,VARCHAR(PKEY,20)                                   |AAA |CCC |
 FROM   TROUBLE                                            |AAA |DDD |
 WHERE  PKEY = 'AAA'                                       |CCC |EEE |
 UNION ALL                                                 |DDD |AAA |
 SELECT CASE                                               |DDD |FFF |
        WHEN LOCATE(C.CKEY||'>',P.PATH) > 0                |DDD |EEE |
             THEN RAISE_ERROR('70001',                     |FFF |GGG |
             'ERROR: LOOP IN DATABASE FOUND')              +---------+
        ELSE C.CKEY
        END
       ,P.LVL +1
       ,P.PATH||'>'||C.CKEY
 FROM   TROUBLE   C
       ,PARENT    P
 WHERE  P.CKEY  = C.PKEY
)
SELECT CKEY, LVL, PATH
FROM   PARENT;
```

Figure 567, Recursive SQL, Set error if data loop found

The above SQL uses a CASE statement to determine what value to put in the PATH field. If the current child row has been processed before, the RAISE_ERROR function is used to generate a SQL error (which stops the statement), else CKEY used. Another way to do the same thing is to delay the RAISE_ERROR processing till the final SELECT. The advantage of doing this is that at least some rows will be fetched before the failure occurs.

```
WITH PARENT                                       ANSWER   TROUBLE
(CKEY, LVL, PATH) AS                              ======   +---------+
(SELECT DISTINCT                                  CKEY...  |PKEY|CKEY|
        PKEY                                      ----     |----|----|
       ,0                                         AAA      |AAA |BBB |
       ,VARCHAR(PKEY,20)                          BBB      |AAA |CCC |
 FROM   TROUBLE                                   CCC      |AAA |DDD |
 WHERE  PKEY = 'AAA'                              DDD      |CCC |EEE |
 UNION ALL                                        EEE      |DDD |AAA |
 SELECT CASE                                      <error>  |DDD |FFF |
        WHEN LOCATE(C.CKEY||'>',P.PATH) > 0                |DDD |EEE |
             THEN CHAR('>>>',3)                            |FFF |GGG |
        ELSE C.CKEY                                        +---------+
        END
       ,P.LVL +1
       ,P.PATH||'>'||C.CKEY
 FROM   TROUBLE   C
       ,PARENT    P
 WHERE  P.CKEY  = C.PKEY
)
SELECT CASE
       WHEN LOCATE('>',CKEY) > 0
            THEN RAISE_ERROR('70001',
             'ERROR: LOOP IN DATABASE FOUND')
       ELSE CKEY
       END   AS CKEY
      ,LVL, PATH
FROM   PARENT;
```

Figure 568, Recursive SQL, Set error if data loop found

Note that the above trick does not work if the field containing the RAISE_ERROR function is in an ORDER BY list. This is because the function will be processed during the row sort that occurs during the cursor open, and not at fetch time.

Working with Other Key Types

In much of the above SQL the ">" character has been used for two special purposes. On the one hand, it helps to improve the general readability of the output. More importantly, it also acts as a key-value separator when the LOCATE function is used to look for prior references to the current key in the PATH field. It is for this reason that the table DDL disallows the use of the ">" in either of the two key columns.

If one wishes to use SQL similar to that shown above (i.e. using the ">" technique) then one is going to have to set-aside some special character in the keys of the target table(s) accordingly. Most applications should have no trouble finding at least one ASCII character that is suitable for the task. In the unlikely event that the key fields are defined FOR BIT DATA and actually contain the full range of valid binary values, one is simply out of luck.

Numeric Keys

To use the techniques described above (i.e. with the ">" value) on numeric keys one simply uses the CHAR function to convert the number to a valid character. The rest is unchanged.

Stopping Simple Recursive Statements Using FETCH FIRST code

Very simple recursive SQL statements (i.e. not the ones shown above) can be stopped in mid-stream using the FETCH FIRST n ROWS ONLY syntax. For example, the following statement will only fetch five rows:

```
WITH TEMP (COL1) AS                                             ANSWER
(VALUES (SMALLINT(1))                                           ======
 UNION ALL                                                        COL1
 SELECT COL1 + 1                                                  ----
 FROM   TEMP)                                                        1
SELECT   COL1                                                        2
FROM     TEMP                                                        3
FETCH FIRST 5 ROWS ONLY;                                             4
                                                                     5
```

Figure 569, Stop recursion using FETCH FIRST n ROWS ONLY, works

The FETCH FIRST syntax stops the above recursive statement because the output from each recursive call is not sent to a temporary file, but rather is passed straight to the output buffer. After five rows are fetched, the whole cursor, including the recursion, stops.

The next example fails because, in this case, the output from the recursion has to be sent to a work file (for sorting) before it can be fetched. Consequently, the recursion runs for a while (in this case until COL1 is larger than the largest allowable smallint value), then fails with an overflow error:

```
WITH TEMP (COL1) AS                                             ANSWER
(VALUES (SMALLINT(1))                                           ======
 UNION ALL                                                       error
 SELECT COL1 + 1
 FROM   TEMP)
SELECT   COL1
FROM     TEMP
ORDER BY COL1
FETCH FIRST 5 ROWS ONLY;
```

Figure 570, Stop recursion using FETCH FIRST n ROWS ONLY, not work

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 571, 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 572, 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 573, 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 574, 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 575, Trigger to maintain exploded table after delete 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 576, 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 577, 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                              HIERARCHY#2
NO CASCADE BEFORE UPDATE OF PKEY ON HIERARCHY#2       +--------------+
REFERENCING NEW AS NNN                                |KEYY|PKEY|DATA|
            OLD AS OOO                                |----|----|----|
FOR EACH ROW MODE DB2SQL                              |AAA |AAA |S...|
WHEN (NNN.PKEY <> OOO.PKEY                            |BBB |AAA |M...|
 AND  NNN.PKEY <> NNN.KEYY)                           |CCC |BBB |M...|
   WITH TEMP (KEYY, PKEY) AS                          |DDD |CCC |M...|
   (VALUES (NNN.KEYY                                  |EEE |BBB |J...|
          ,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;
```

Figure 578, 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 579, 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 580, 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 581, 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 582, 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 583, 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 584, 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 585, 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 586, 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 587, 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 588, 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 589, 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 590, 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 591, 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 592, 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 593, 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 85) 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 594, 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 595, 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 596, 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 597, 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 598, 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 599, 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 600, 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 220).
- 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.

Statistical Analysis using SQL

DB2 contains two powerful statistical functions: STDDEV will get the standard deviation of a column of numeric data. VARIANCE will get the variance of the same. In addition, one uses other SQL functions to do many kinds of statistical analysis. Read on for details.

Sample Table DDL & DML

Below is the DDL for a data table into which we insert 1,000 rows. The table has five columns. The first contains the numbers zero through 999. The second, third, and fourth, have a series of random numbers in the range of zero to 32K. The last field only allows the values 'F' (for female) or 'M' (for male).

```
CREATE TABLE STATS
(KEY1       SMALLINT       NOT NULL  CHECK(KEY1 >= 0)
,DAT1       SMALLINT       NOT NULL  CHECK(DAT1 >= 0)
,DAT2       SMALLINT       NOT NULL  CHECK(DAT2 >= 0)
,DAT3       SMALLINT       NOT NULL  CHECK(DAT3 >= 0)
,SEX        CHAR(01)       NOT NULL  CHECK(SEX  IN ('F','M'))
,CONSTRAINT STX1 PRIMARY KEY (KEY1) );
COMMIT;

CREATE INDEX STX2 ON STATS (DAT1);
COMMIT;

INSERT INTO STATS
WITH TEMP1 (K1,R1,R2) AS
(VALUES    (0,RAND(1),RAND() )
 UNION ALL
 SELECT  K1+1, RAND(), RAND()
 FROM    TEMP1
 WHERE   K1+1 < 1000
)
SELECT K1 AS KEY1
      ,INT(R1 * 32767) AS DAT1
      ,INT(R2 * 32767) AS DAT2
      ,INT(POWER(R2,2)*32767) AS DAT3
      ,CASE
         WHEN R1+0.4 > 1  THEN 'M'
         ELSE 'F'
       END AS SEX
FROM   TEMP1;
```

Figure 601, Sample table DDL - Statistical Analysis

Observe that all of the "DAT" fields contain random data values. Also note that the first two fields are random with respect to each other, while the last two are somewhat related.

Mean

The mean and the average are the same. They are simply the sum of all the individual scores, divided by the number of values where the value is not null. Use the DB2 AVG function to calculate the mean.

```
Mean =      sum(values)
       ---------------------
         num-rows (not null)
```

Figure 602, Get the Mean - Formula

```
SELECT AVG(DAT1)                                                ANSWER
FROM   STATS;                                                   ======
                                                                 16271
```

Figure 603, Get the Mean - SQL

NOTE: Null values are ignored during statistical analysis.

Median

The median is the point where half of the items in the sample are above (in value) and half are below. To calculate the median, one must first get a count of all the items in the sample, then sequence each individual value (from low to high by value). Lastly, one must get that item which is halfway down the series.

All of the above can be done using a single SQL statement in DB2, but the performance stinks! The only efficient way to answer the question would be to add fields and indexes to the original table. Rather than do this, we simply admitted defeat.

Mode

The mode is the point (or points) in the sample with the highest frequency. This one is easy:

```
SELECT   DAT1                                                ANSWER
        ,COUNT(*) AS CNT                                     =========
FROM     STATS                                               DAT1  CNT
GROUP BY DAT1                                                ----- ---
HAVING   COUNT(*) >= ALL                                        28   2
        (SELECT   COUNT(*)                                    1626   2
         FROM     STATS                                       3093   2
         GROUP BY DAT1)                                       4313   2
ORDER BY 1;                                                   etc..
```

Figure 604, Get the Mode - SQL

Average Deviation

The average deviation is a little used measure of statistical variability. It gets the average of the difference between individual scores and the mean of the sample.

```
Avg Deviation = sum( absolute(value - mean))
                ----------------------------
                     num-rows (not null)
```

Figure 605, Get the Average Deviation - Formula

```
WITH MEAN (MEAN) AS                                             ANSWER
(SELECT  AVG(DAT1)                                              ======
 FROM    STATS                                                    8207
)
SELECT  SUM(ABS(DAT1 - MEAN)) / COUNT(*)
FROM    STATS
       ,MEAN;
```

Figure 606, Get the Average Deviation - SQL

A slightly simpler way to write the same code follows. The DB2 SUM and COUNT functions have been replaced by the functionally equivalent AVG function.

```
WITH MEAN (MEAN) AS                                             ANSWER
(SELECT  AVG(DAT1)                                              ======
 FROM    STATS                                                    8207
)
SELECT  AVG(ABS(DAT1 - MEAN))
FROM    STATS
       ,MEAN;
```

Figure 607, Get the Average Deviation - SQL

Variance

The mean of the squared deviation scores is called the variance.

```
Variance = sum( square( absolute(value - mean)))
           -------------------------------------
                     num-rows (not null)
```

Figure 608, Get the Variance - Formula

The VARIANCE column function returns the variance of a set of numeric data. The function output is always of type double - regardless of the input type.

```
SELECT  VARIANCE(DAT1)                 ANSWER
       ,BIGINT(VARIANCE(DAT1))         ===============================
FROM    STATS;                         1                      2
                                       ---------------------- --------
                                       +8.93711402817241e+007 89371140
```

Figure 609, Get the Variance - SQL

Standard Deviation

The standard deviation is one of the most widely used measures of data variability. To calculate it one first gets the square of each individual deviation from the mean (note: there is one result per data value). The square root of the mean of the sum of all the individual results is then calculated. This result is the standard deviation.

```
Std Deviation = sqroot( sum( square( absolute(value - mean))))
                ----------------------------------------------
                                num-rows
```

Figure 610, Get the Standard Deviation - Formula

The STDDEV column function gets the standard deviation of a set of numeric data. The function output is always of type DOUBLE - regardless of the input type.

```
SELECT  STDDEV(DAT1)                       ANSWER
       ,INTEGER(STDDEV(DAT1))              ===========================
FROM    STATS;                             1                      2
                                           ---------------------- ----
                                           +9.45363106333879e+003 9453
```

Figure 611, Get the Standard Deviation - SQL

NOTE: Both the STDDEV and the VARIANCE function come with a syntax option that allows one to apply the function against the DISTINCT list of input data. Doing this makes sense is some cases, but can be misleading. The feature should be used with care.

Data Distribution

In a sample-set with normal data distribution there ought to be a set percentage of values within each standard deviation of the mean. This is illustrated in the following diagram:

```
                                *****
                             ***  |  ***
                          ***     |     ***
                       ***        |        ***
                    *** |         |         | ***
                ****    |         |         |    ****
           *****        |    34%  |  34%    |        *****
     ******   |    14%  |         |         |   14%   |   ******
<<<--------------------------------------------------------------->>>
             -2S       -1S        M        +1S       +2S
```

Figure 612, Normal distribution data divided into standard deviations

Now we shall see if our sample data follows this trend. To do this we shall write a multi-part SQL statement that does the following (in sequence):

- Get the mean of the whole sample.

- Calculate the first and second standard deviations above and below the mean.
- Count the number of rows that fall within each standard deviation.

```
WITH STD_DV (NUM,M,M_N1,M_N2,M_P1,M_P2) AS     ANSWER
(SELECT  COUNT(*) * 1E0                        =======================
        ,AVG(DAT1)                             SUB2  SUB1  ADD1  ADD1
        ,AVG(DAT1) -(1*(STDDEV(DAT1)))         ----- ----- ----- -----
        ,AVG(DAT1) -(2*(STDDEV(DAT1)))          0.21  0.27  0.29  0.21
        ,AVG(DAT1) +(1*(STDDEV(DAT1)))
        ,AVG(DAT1) +(2*(STDDEV(DAT1)))
 FROM    STATS
)
SELECT  DECIMAL(SUM(CASE
          WHEN DAT1 BETWEEN M_N2 AND M_N1 THEN 1
          ELSE 0
        END) / AVG(NUM),3,2) AS SUB2
       ,DECIMAL(SUM(CASE
          WHEN DAT1 BETWEEN M_N1 AND M  THEN 1
          ELSE 0
        END) / AVG(NUM),3,2) AS SUB1
       ,DECIMAL(SUM(CASE
          WHEN DAT1 BETWEEN M  AND M_P1 THEN 1
          ELSE 0
        END) / AVG(NUM),3,2) AS ADD1
       ,DECIMAL(SUM(CASE
          WHEN DAT1 BETWEEN M_P1 AND M_P2 THEN 1
          ELSE 0
        END) / AVG(NUM),3,2) AS ADD1
FROM    STATS ,STD_DV;
```

Figure 613, See if data has normal data distribution

The values in the sample table do not have a normal data distribution. Given that the rows were created using a random number generator, this is not surprising.

Standard Error

Imagine that the data being processed is but a small fraction of that available in the real world. The standard error is an estimate what the variability of the means of a standard sample would be - based on the available data.

```
Std Error =   Std-deviation
             ----------------
             sqroot(num-rows)
```

Figure 614, Get the Standard Error - Formula

```
SELECT  INT(STDDEV(DAT1)                                        ANSWER
           /SQRT(COUNT(*)))                                     ======
FROM    STATS;                                                     298
```

Figure 615, Get the Standard Error - SQL

Pearson Product-Moment Coefficient

The Pearson product-moment coefficient is a measure of the relationship between two sets of values. The result varies from minus one to one . If the result is one (or close to one), then the data values in the two fields are strongly related. If the result is zero, there is no relationship. A result that is less than zero implies a negative relationship.

```
Pearsons PMC = sum((value1 - mean1) * (value2 - mean2))
                ----------------------------------------
                    num-rows * std-dev1 * std-dev2
```

Figure 616, Get the Pearson Product-Moment Coefficient - Formula

In the following statement we compare DAT1 against DAT2 and DAT2 against DAT3. The first pair are unrelated while the second pair are strongly related in that one is the square of the other (see DDL on page 222).

```
WITH DATA (M1,S1,M2,S2,M3,S3) AS                          ANSWER
(SELECT  AVG(DAT1), STDDEV(DAT1)                          ============
        ,AVG(DAT2), STDDEV(DAT2)                          PP1-2  PP2-3
        ,AVG(DAT3), STDDEV(DAT3)                          ------ -----
 FROM    STATS                                            -0.010 0.968
)
SELECT  DEC(SUM(1E0 * (DAT1 - M1) * (DAT2 - M2))
          /(COUNT(*) * AVG(S1) * AVG(S2)),4,3) AS "PP1-2"
       ,DEC(SUM(1E0 * (DAT2 - M2) * (DAT3 - M3))
          /(COUNT(*) * AVG(S2) * AVG(S3)),4,3) AS "PP2-3"
FROM    STATS,DATA;
```

Figure 617, Get the Pearson Product-Moment Coefficient - SQL

NOTE: The use of the 1E0 in the above SQL converts the intermediary results into floating point and so avoids overflow errors. Just in case you've forgot, 1E0 is the value one.

Time-Series Data Usage

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 618, 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.

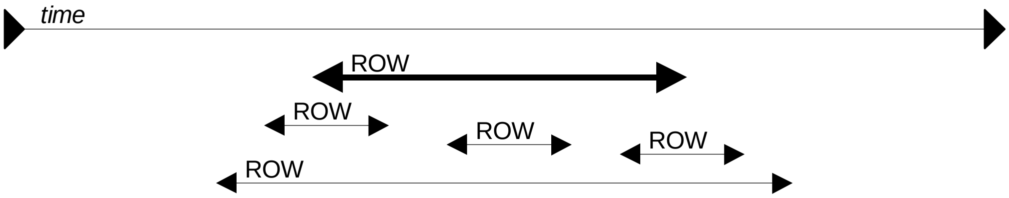


Figure 619, Overlapping Time-Series rows - Definition

> 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 flavours 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 620, 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 226. 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                               TIME_SERIES
      ,A.BGN_DT                            +------------------------+
      ,A.END_DT                            |KYY|BGN_DT    |END_DT    |
      ,B.BGN_DT                            |---|----------|----------|
      ,B.END_DT                            |AAA|1995-10-01|1995-10-04|
      ,DAYS(B.BGN_DT) -                    |AAA|1995-10-06|1995-10-06|
       DAYS(A.END_DT)                      |AAA|1995-10-07|1995-10-07|
         AS DIFF                           |AAA|1995-10-15|1995-10-19|
FROM   TIME_SERIES A                       |BBB|1995-10-01|1995-10-01|
      ,TIME_SERIES B                       |BBB|1995-10-03|1995-10-03|
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;
```

Figure 621, 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 622, 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                               TIME_SERIES
      ,A.END_DT + 1 DAY                    +------------------------+
         AS BGN_GAP                        |KYY|BGN_DT    |END_DT    |
      ,B.BGN_DT - 1 DAY                    |---|----------|----------|
         AS END_GAP                        |AAA|1995-10-01|1995-10-04|
      ,(DAYS(B.BGN_DT) -                   |AAA|1995-10-06|1995-10-06|
        DAYS(A.END_DT) - 1)                |AAA|1995-10-07|1995-10-07|
         AS GAP_SIZE                       |AAA|1995-10-15|1995-10-19|
FROM   TIME_SERIES A                       |BBB|1995-10-01|1995-10-01|
      ,TIME_SERIES B                       |BBB|1995-10-03|1995-10-03|
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;
```

Figure 623, 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 624, 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                                  TIME_SERIES
(KYY, GAP_DT, GSIZE) AS                    +-------------------------+
(SELECT A.KYY                              |KYY|BGN_DT    |END_DT    |
       ,A.END_DT + 1 DAY                   |---|----------|----------|
       ,(DAYS(B.BGN_DT) -                  |AAA|1995-10-01|1995-10-04|
         DAYS(A.END_DT) - 1)               |AAA|1995-10-06|1995-10-06|
 FROM   TIME_SERIES A                      |AAA|1995-10-07|1995-10-07|
       ,TIME_SERIES B                      |AAA|1995-10-15|1995-10-19|
 WHERE  A.KYY    = B.KYY                   |BBB|1995-10-01|1995-10-01|
   AND  A.END_DT < B.BGN_DT - 1 DAY        |BBB|1995-10-03|1995-10-03|
   AND  NOT EXISTS                         +-------------------------+
       (SELECT *
        FROM   TIME_SERIES Z
        WHERE  Z.KYY    = A.KYY
          AND  Z.KYY    = B.KYY                ANSWER
          AND  Z.BGN_DT > A.BGN_DT             =======================
          AND  Z.BGN_DT < B.BGN_DT)            KEYCOL GAP_DT     GSIZE
 UNION ALL                                     ------ ---------- -----
 SELECT KYY                                    AAA    10/05/1995     1
       ,GAP_DT + 1 DAY                         AAA    10/08/1995     7
       ,GSIZE  - 1                             AAA    10/09/1995     6
 FROM   TEMP                                   AAA    10/10/1995     5
 WHERE  GSIZE  > 1                             AAA    10/11/1995     4
)                                              AAA    10/12/1995     3
SELECT   *                                     AAA    10/13/1995     2
FROM     TEMP                                  AAA    10/14/1995     1
ORDER BY 1,2;                                  BBB    10/02/1995     1
```

Figure 625, Show each day in Time-Series gap

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 626, 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 627, Acceptable conversion values

Checking the Input

A CASE statement can be used to check that the input string is a valid representation of a number before doing the data conversion. In the next example we are converting to smallint, so the input has be a valid character representation of an integer value.

```
WITH TEMP1 (C1) AS (VALUES ' 123','456 ',' 1 2',' 33%',NULL)
SELECT C1
      ,CHAR(RTRIM(LTRIM(C1)),4)
      ,CHAR(TRANSLATE(RTRIM(LTRIM(C1)),'$',' 0123456789'),4)
      ,CASE
         WHEN TRANSLATE(RTRIM(LTRIM(C1)),'$',' 0123456789') = ''
         THEN SMALLINT(C1)
         ELSE SMALLINT(0)
       END
FROM   TEMP1;
```

Figure 628, Covert Character to Numeric (check input), SQL

The answer follows. The left-most field is the original input, the right-most is the desired output. The two in the middle illustrate what is being done.

```
C1     2      3      4
----   ----   ----   ------
 123   123              123
456    456              456
 1 2   1 2     $          0
 33%   33%      %         0
-      -      -           0
```

Figure 629, Covert Character to Numeric (check input), Answer

The objective of the above CASE statement is to see if there are any characters in the input field other than digits. To do this, first all leading and trailing blanks are removed. Then embedded blanks are converted to dollar signs and all digits are converted to blank. If the result is a blank value then we must have a simple integer value as input.

Converting character to decimal is illustrated below. It is much the same as above, except that now one must also accept as valid input a number with an embedded dot - but not a single dot by itself, nor multiple dots:

```
WITH TEMP1 (C1) AS (VALUES '.123','4.6.',' 1.2',' 33.',NULL)
SELECT C1
      ,CHAR(RTRIM(LTRIM(C1)),4)
      ,CHAR(TRANSLATE(RTRIM(LTRIM(C1)),'$',' 0123456789'),4)
      ,CASE
         WHEN LTRIM(TRANSLATE(RTRIM(LTRIM(C1)),'$',' 0123456789'))
              = ''  THEN DECIMAL(C1,7,3)
         WHEN LTRIM(TRANSLATE(RTRIM(LTRIM(C1)),'$',' 0123456789'))
              = '.' AND LTRIM(C1) <> '.' THEN DECIMAL(C1,7,3)
         ELSE DECIMAL(   0,7,3)
       END
FROM   TEMP1;
```

Figure 630, Covert Character to Decimal (check input), SQL

```
C1     2      3      4
----   ----   ----   -------
.123   .123   .        0.123
4.6.   4.6.    . .     0.000
 1.2   1.2     .       1.200
 33.   33.      .     33.000
-      -      -        0.000
```

Figure 631, Covert Character to Decimal (check input), Answer

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 632, 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 633, 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 634, 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 635, Make chart using SQL

```
COL1   PRETTY_CHART
----   --------------------------------------------------
  12   ************
  22   **********************
  33   *********************************
  16   ****************
   0
  44   ********************************************
  15   ***************
  15   ***************
```

Figure 636, 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 637, Make charts using SQL

```
COL1   PRETTY_CHART
----   --------------------------------------------------
  12   ************
  22   **********************
  33   *********************************
  16   ****************
   0
  66   ***********************************************>>>
  15   ***************
  15   ***************
```

Figure 638, 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 107 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 639, 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 640, 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 641, 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 642, 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 643, 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 644, 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                                           ANSWER
        ,COUNT(*) AS #STAFF                              =============
FROM     STAFF                                           YEARS  #STAFF
WHERE    UCASE(NAME)  LIKE '%E%'                         -----  ------
  AND    YEARS          <=  5                                1       1
GROUP BY YEARS;                                              4       2
                                                             5       3
```

Figure 645, 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                                ANSWER
(VALUES (0),(1),(2),(3),(4),(5)                           ============
)                                                         YEARS #STAFF
SELECT   YEAR#             AS YEARS                       ----- ------
        ,COALESCE(#STFF,0) AS #STAFF                          0      0
FROM     LIST_YEARS                                           1      1
LEFT OUTER JOIN                                               2      0
        (SELECT   YEARS                                       3      0
                  ,COUNT(*) AS #STFF                          4      2
         FROM     STAFF                                       5      3
         WHERE    UCASE(NAME) LIKE '%E%'
           AND    YEARS         <=  5
         GROUP BY YEARS
        )AS XXX
ON       YEAR# = YEARS
ORDER BY 1;
```

Figure 646, 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 647, 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 648, 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 649, 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 650, 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   *
FROM     TEMP1 D
WHERE    W# =
        (SELECT MAX(W#)
         FROM   TEMP1 E
         WHERE  D.DEPT = E.DEPT)
ORDER BY DEPT;
```

Figure 651, 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#      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 652, Denormalize Normalized Data - Answer

Reversing Field Contents

DB2 lacks a simple function for reversing the contents of a data field. Fortunately, there are several, somewhat tedious, ways to achieve the required result in SQL.

Input vs. Output

Before we do any data reversing, we have to define what the reversed output should look like for 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.

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.

Reversing Integers

The following SQL statement reverses the contents of a four-digit numeric field. Leading zeros are implied when the input is less than four digits long.

```
WITH TEMP1 (N1) AS                                         ANSWER
(VALUES (-0124),(+0000),(+0001)                            ===========
       ,(+0456),(+6789),(+9999))                           N1    2
SELECT N1                                                  ----- -----
      ,( ((N1/   1)-(N1/   10)*10) *1000) +                 -124 -4210
       ( ((N1/  10)-(N1/  100)*10) * 100) +                    0     0
       ( ((N1/ 100)-(N1/ 1000)*10) *  10) +                    1  1000
       ( ((N1/1000)-(N1/10000)*10) *   1)                    456  6540
FROM   TEMP1;                                               6789  9876
                                                            9999  9999
```

Figure 653, Reverse digits in four-byte integer field - assume leading zeros

The next statement does the job when leading zeros are not assumed. However, it does assume that the input has a maximum of four digits.

```
WITH TEMP1 (N1) AS                                         ANSWER
(VALUES (-0124),(+0000),(+0001)                            ===========
       ,(+0456),(+6789),(+9999))                           N1    2
SELECT N1                                                  ----- -----
      ,(( ((N1/   1)-(N1/   10)*10) *1000) +                -124  -421
        ( ((N1/  10)-(N1/  100)*10) * 100) +                   0     0
        ( ((N1/ 100)-(N1/ 1000)*10) *  10) +                   1     1
        ( ((N1/1000)-(N1/10000)*10) *   1))/                 456   654
        CASE                                                6789  9876
           WHEN ABS(N1) <   10 THEN 1000                    9999  9999
           WHEN ABS(N1) <  100 THEN  100
           WHEN ABS(N1) < 1000 THEN   10
           ELSE 1
        END
FROM   TEMP1;
```

Figure 654, Reverse digits in four-byte integer field - assume no leading zeros

Another way to reverse the contents of a numeric field is to first convert the data to character. Then use the character-reversing logic described below before converting the character data back to numbers.

Reversing Characters

The next statement reverses the contents of an eight-character field. Trailing blanks are converted into leading blanks in the output.

```
SELECT   JOB                                        ANSWER
        ,SUBSTR(JOB,8,1) || SUBSTR(JOB,7,1)         =================
      || SUBSTR(JOB,6,1) || SUBSTR(JOB,5,1)         JOB      BOJ
      || SUBSTR(JOB,4,1) || SUBSTR(JOB,3,1)         -------- --------
      || SUBSTR(JOB,2,1) || SUBSTR(JOB,1,1)         ANALYST   TSYLANA
         AS BOJ                                     CLERK       KRELC
FROM     EMPLOYEE                                   DESIGNER RENGISED
GROUP BY JOB;                                       FIELDREP PERDLEIF
                                                    MANAGER   REGANAM
                                                    OPERATOR ROTAREPO
                                                    PRES         SERP
                                                    SALESREP PERSELAS
```

Figure 655, Reverse characters - keep trailing blanks

In the following statement, trailing blanks (which become leading blanks) are discarded.

```
SELECT   JOB                                        ANSWER
        ,CHAR(LTRIM(                                =================
         SUBSTR(JOB,8,1) || SUBSTR(JOB,7,1)         JOB      BOJ
      || SUBSTR(JOB,6,1) || SUBSTR(JOB,5,1)         -------- --------
      || SUBSTR(JOB,4,1) || SUBSTR(JOB,3,1)         ANALYST  TSYLANA
      || SUBSTR(JOB,2,1) || SUBSTR(JOB,1,1)         CLERK    KRELC
         ),8) AS BOJ                                DESIGNER RENGISED
FROM     EMPLOYEE                                   FIELDREP PERDLEIF
GROUP BY JOB;                                       MANAGER  REGANAM
                                                    OPERATOR ROTAREPO
                                                    PRES     SERP
                                                    SALESREP PERSELAS
```

Figure 656, Reverse characters - discard trailing blanks

The above two statements have to be altered if the field being reversed is either longer or shorter that what is given. We can use recursion to define a more general-purpose query. In the following example we will reverse the NAME field in the staff table. The logic goes:

- Begin by putting the last character of the NAME into the NEW_NAME column and the rest of the NAME into the REST column.
- Recursively, keep concatenating the last character in the REST column to what is already in the NEW_NAME column while the leaving the rest of the name in the REST column. Stop processing when the REST column has no more characters.
- Finally, select those rows where the REST column is empty.

```
WITH TEMP (OLD_NAME, NEW_NAME, REST) AS
(SELECT   NAME
         ,SUBSTR(NAME,LENGTH(NAME))
         ,SUBSTR(NAME,1,LENGTH(NAME)-1)
 FROM     STAFF
 WHERE    ID < 100
 UNION    ALL
 SELECT   OLD_NAME
         ,NEW_NAME CONCAT
          SUBSTR(REST,LENGTH(REST))
         ,SUBSTR(REST,1,LENGTH(REST)-1)
 FROM     TEMP
 WHERE    LENGTH(REST) > 0
)
SELECT   OLD_NAME
        ,NEW_NAME
FROM     TEMP
WHERE    REST = ''
ORDER BY OLD_NAME;
```

```
ANSWER
=================
OLD_NAME NEW_NAME
-------- --------
Hanes    senaH
James    semaJ
Koonitz  ztinooK
Marenghi ihgneraM
O'Brien  neirB'O
Pernal   lanreP
Quigley  yelgiuQ
Rothman  namhtoR
Sanders  srednaS
```

Figure 657, Reverse character field using recursion

Stripping Characters

DB2 for OS/390 comes with a cute function for stripping characters (blank, or non-blank) from either or both ends of a character string. Baby DB2 lacks such a function, so below it has been emulated below using the native RTRIM and LTRIM functions. The trick is to first change all of the blanks to something else, then the character to be stripped to blank, then trim either end, then change everything back again. This query generates a friendly error message if the input string has the character value that will be used to hold the blanks:

```
WITH TEMP (TXT) AS
(VALUES ('HAS TRAILING ZEROS 0000')
       ,('HAS 0 IN MIDDLE    0000')
       ,('HAS TEMP ~ CHAR    0000'))
SELECT  TXT
       ,TRANSLATE(
          TRANSLATE(
            LTRIM(
              RTRIM(
                TRANSLATE(
                  TRANSLATE(TXT,'~',' ')
                ,' ','0')
              )
            )
          ,'0',' ')
        ,' ','~') AS TXT_STRIPPED
FROM    TEMP
WHERE   CASE LOCATE('~',TXT)
           WHEN 0 THEN ''
           ELSE RAISE_ERROR('80001','FOUND ~ YOU IDIOT!')
        END = '';
```

```
ANSWER
===============================================
TXT                      TXT_STRIPPED
-----------------------  -------------------
HAS TRAILING ZEROS 0000  HAS TRAILING ZEROS
HAS 0 IN MIDDLE    0000  HAS 0 IN MIDDLE
error 80001, "FOUND ~ YOU IDIOT!"
```

Figure 658, Stripping non-blank characters

Use the REPLACE function to remove characters from anywhere in a character string. In the following example, every "e" is removed from the staff name:

```
SELECT NAME                 AS OLD_NAME             ANSWER
      ,REPLACE(NAME,'e','') AS NEW_NAME             ==================
FROM   STAFF                                        OLD_NAME  NEW_NAME
WHERE  ID < 50;                                     --------  --------
                                                    Sanders   Sandrs
                                                    Pernal    Prnal
                                                    Marenghi  Marnghi
                                                    O'Brien   O'Brin
```

Figure 659, Strip characters from text

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 660, 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.

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 661, 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 662, Timestamp comparison - Correct

Be aware that, as with everything else in this section, what is shown above is not a bug. It has always worked this way, even in DB2 for OS/390, and probably always will. Code with care.

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 663, Query with no matching rows (1 of 8)

```
SELECT   MAX(CREATOR)                                           ANSWER
FROM     SYSIBM.SYSTABLES                                       ======
WHERE    CREATOR = 'ZZZ';                                       <null>
```

Figure 664, 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 665, 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 666, Query with no matching rows (4 of 8)

```
SELECT   MAX(CREATOR)                                          ANSWER
FROM     SYSIBM.SYSTABLES                                     ========
WHERE    CREATOR = 'ZZZ'                                      <no row>
GROUP BY CREATOR;
```

Figure 667, Query with no matching rows (5 of 8)

```
SELECT   CREATOR                                               ANSWER
FROM     SYSIBM.SYSTABLES                                     ========
WHERE    CREATOR = 'ZZZ'                                      <no row>
GROUP BY CREATOR;
```

Figure 668, Query with no matching rows (6 of 8)

```
SELECT   COUNT(*)                                              ANSWER
FROM     SYSIBM.SYSTABLES                                     ========
WHERE    CREATOR = 'ZZZ'                                      <no row>
GROUP BY CREATOR;
```

Figure 669, Query with no matching rows (7 of 8)

```
SELECT   COUNT(*)                                               ANSWER
FROM     SYSIBM.SYSTABLES                                       ======
WHERE    CREATOR = 'ZZZ';                                            0
```

Figure 670, 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.

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 671, 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 672, 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 673, 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 674, 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 675, 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 676, 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 677, 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 678, 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 679, 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 680, 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 681, 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 682, 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 683, 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 684, 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 685, 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 686, 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 687, Use LIKE on VARCHAR field

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 that specifies the output format of date-time data. This bind option has no impact on the sequence with which date-time data is presented. To illustrate, the plan that was used to run the following SQL is set 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 688, 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 689, 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 690, 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 691, 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 692, 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 693, 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 694, 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 695, 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 696, 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 697, 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 698, 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 699, 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 700, 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 701, Trademark Valid SQL

Fortunately, we still get the same (correct) answer.

DB2 Dislikes

This book documents lots of things that are not covered in the official manuals. It wouldn't complete unless it also mentioned some of those items that are really lousy in DB2.

Performance Analysis

I'd love to write a book on monitoring and tuning DB2 SQL statements and applications. In fact, I did write such a book in the past - way back in V2. IBM would also like me to write such a book, but I never do. So what gives? There are problems…

The DB2 Monitor

The monitor that comes with DB2 is absolutely abysmal. However, like any DBMS monitor that pretends to be even half-decent, it has all the usual components:

- A snapshot monitor to see the current state of the system.
- An event monitor to trace events as they occur.
- A pretty little GUI to view the trace data.
- A way to export the trace data to external files.

What is Wrong

- There is no way to get a basic measure of SQL costs. One can not run a statement with the monitor on and then get useful data about the performance of that statement. For example, one can not get an accurate measure of CPU time, nor the I/O time, nor the number of I/O done, and not even a count of the number of rows fetched.
- The performance data that is presented (in a SQL statement trace report) is generally either inaccurate or misleading. Some values are zero when they should not be, and some values are accumulated in some cases but not others.
- The documentation is vague at best and more often missing. I could not figure out what the output of a SQL statement trace meant without actually asking the people in IBM who wrote the code.

Because one can not accurately measure the performance of a statement, one can not do basic SQL tuning. One can not, for example, add or remove indexes and detect minor differences in query performance. Nor can one easily simply turn a trace on, and trace only those statements that are running unduly long.

One bit of good news is that the DB2 monitor has been this bad of years, so at least IBM is consistent. Also, having an abysmal monitor makes it much easier for IBM to add new features to DB2 (e.g. triggers) that add five or ten percent to general SQL statement costs. By contrast, IBM can not do this in DB2 for OS/390, because too many big customers would detect the extra costs and complain.

What is Needed

A monitor should **not** be a collection of canned reports. Instead it should be a report-writer that much like DB2 itself lets users compose their own traces, selecting their own fields and rows based on their needs.

A simplified version of SQL would be an ideal language for such a monitor. To illustrate, the following hypothetical query would trace all SQL statements, excluding inserts, that took more than 100 seconds to complete:

```
SELECT  EVENT_TIMESTAMP
       ,PROGRAM_NAME
       ,STATEMENT#
       ,SQL_TYPE
       ,AUTHID
       ,ELP_SEC
       ,CPU_SEC
       ,#ROWS
FROM    TRACE.SQL_STATEMENT
WHERE   ELP_SEC   >  100
  AND   SQL_TYPE <> 'INSERT';
```
Figure 702, Hypothetical trace query - SQL Statement

The following query would get the current state of the sort heap:

```
SELECT  *
FROM    SNAPSHOT.SORT_HEAP
```
Figure 703, Hypothetical trace query - Sort Heap

The next query would collect global memory information every 30 seconds:

```
SELECT  *
FROM    SNAPSHOT.GLOBAL_MEMORY
EVERY   30 SECONDS;
```
Figure 704, Hypothetical trace query - Global Memory

Obviously, all of the above could be wrapped in a suitable GUI for those who don't like to write their own code.

Win/NT Problems

Below is some output from a DB2BATCH run. What is shown is the little performance summary that is given at the bottom of the output - with one line per statement that was run:

```
Summary of Results
==================
                Elapsed             Agent CPU         Rows      Rows
Statement #     Time (s)            Time (s)          Fetched
Printed
1                     0.016       Not Collected         1         1
2             Not Collected       Not Collected        10        10
3                     0.016       Not Collected        10        10
4             Not Collected       Not Collected        10        10
5                     0.016       Not Collected       100       100
6                     0.063       Not Collected      1000       300
7                     0.109       Not Collected         1         1
8             Not Collected       Not Collected         1         1
9                     0.016       Not Collected         1         1

Arith. mean     0.039               0.0156
Geom.  mean     0.028               0.0787
```
Figure 705, DB2BATCH SQL-summary listing

What is Wrong

- Observe that quite a few SQL statements took 16 milliseconds to run. In truth, no statement took 16 milliseconds to run. This is simply the precision of the OS clock that DB2 is using to get the times. All of the above elapsed times are approximate multiples of 16 milliseconds.
- Observe that for some statements, the elapsed time was "not collected". It was collected, but it was (according to the silly OS clock) zero, and so ignored.
- Observe that no individual CPU times values were collected. Yet somehow DB2 has managed to work out that the average CPU time was 0.0156 seconds, which is suspiciously close to 16 milliseconds.

The DB2 monitor seems to give more accurate elapsed times, which are however wrong for other reasons. The SQL statement CPU time that is recorded by the DB2 monitor is either zero or some vague multiple of 16 milliseconds.

> NOTE: Different machines and different operating systems may have a different OS clock tick values. Not every machine will use 16 milliseconds.

Visual Explain

Imagine that I had two logically equivalent SQL statements, one of which ran much faster than the other. How can I describe to you the two access paths? I can not use Visual Explain because it is a GUI tool, and much of the useful information is in various drill-down boxes. Nor can I use the text-based Explain that also comes with DB2, because it is not well known, and not at all well documented.

What is Wrong

- As with many GUI products, the data displayed by Visual Explain can not be shared. You can not simply cut-and-paste a Visual Explain tree-structure and email it to a co-worker. And if you can not share, you can not act as a team, and you will learn (how to tune SQL) much slower than you should because you will always be working by yourself.
- The Visual Explain output is **graphical** and not **verbal**. Consequently, it is hard to describe in a conversation (e.g. over the phone) the access path that you see in front of your eyes. This is especially true when even a moderately complicated join is involved.
- Visual Explain can not be used to go hunting for inefficient SQL statements. One can not, for example, use it to find all those statements that do a non-indexed scan on a particular table. To answer such a question, one has to query the underlying EXPLAIN_OBJECT table directly.

What is Needed

Visual Explain is a fine product **-** as far as it goes. But IBM should put more emphasis on alternative technologies that enable one to share access path information in a group.

> NOTE: It seems that IBM's long-term objective is to make SQL statement tuning largely unnecessary by including in DB2 intelligent software that will suggest and perhaps even build suitable indexes as needed.

Other Whines

Statement Terminator

It used to be that you could always terminate a SQL statement with a semi-colon. Then IBM added triggers - with atomic SQL. Now you can sometimes use a semi-colon, and sometimes not. In short, DB2 no longer has a standard all-purpose SQL statement terminator. To illustrate, here is some trigger DDL that ends with a semi-colon:

```
CREATE TRIGGER XXX.EMP_DELETE
AFTER DELETE ON XXX.EMPLOYEE
REFERENCING OLD AS OOO
FOR EACH ROW
MODE DB2SQL
   DELETE
   FROM  XXX.EMP_ACT
   WHERE EMPNO = OOO.EMPNO;
```

Figure 706, Trigger DDL that can end with a semi-colon

The following trigger is logically identical the above, but because it uses atomic SQL, a semi-colon can not be used. I have chosen instead to use an exclamation mark:

```
CREATE TRIGGER XXX.EMP_DELETE
AFTER DELETE ON XXX.EMPLOYEE
REFERENCING OLD AS OOO
FOR EACH ROW
MODE DB2SQL
   BEGIN ATOMIC
      DELETE
      FROM  XXX.EMP_ACT
      WHERE EMPNO = OOO.EMPNO;
   END!
```

Figure 707, Trigger DDL that can not end with a semi-colon

When atomic SQL is used, each sub-statement ends with a semi-colon. If the main statement also ended with a semi-colon, then DB2BATCH and/or the DB2 Command Processor would have to, in some sense, "understand" the SQL being invoked in order to know what was a valid statement end and what was not.

Rather than solve this problem, IBM decided to make it **our** problem. If we use atomic SQL then we have to go to the effort of choosing some alternate character as the statement terminator. Fortunately, they have made things somewhat easy by letting us alter the terminator value in-stream using the "#SET" notation:

```
--#SET DELIMITER ?

SQL statements that end with a "?"

--#SET DELIMITER ;

SQL statements that end with a ";"
```

Figure 708, Changing the statement delimiter in DB2BATCH

Stopping Recursion

I have a whole section (see page 202) on how to stop recursive SQL statements from running forever. This section shouldn't be needed. There ought to be a simple way check if one has gotten into a loop.

One solution that I proposed to IBM was a LOCATE_BLOCK function that'd scan a character string, a block at a time. If one concatenated each key returned to a string of keys so far retrieved (as I do in my examples), this function could then determine if one was in a loop.

Unfortunately, IBM initially agreed to put the function in V6, but then I pulled it out because I was told that the problem could be solved using XML. Silly me.

RAND Function

The RAND function returns random values in the range of zero to one inclusive. Two features of this function features are especially annoying:

- It only returns 32,768 (i.e. 32K) distinct values. This forces one to play various silly games if more unique values are required.
- It occasionally returns the value zero and/or one. Both of these values are valid, but their all too frequent occurrence can complicate code when one is trying to generate an even distribution of random values within a given range.

The following SQL statement returns 640,000 rows. It was run on both DB2 for NT and DB2 for OS/390 in order to illustrate the RAND function implementation differences:

```
SELECT   COUNT(*)                    AS #ROWS
        ,COUNT(DISTINCT RAN)         AS #VALUES
        ,DECIMAL(AVG(RAN),8,7)       AS AVG_RAN
        ,DECIMAL(MIN(RAN),8,7)       AS MIN_RAN
        ,DECIMAL(MAX(RAN),8,7)       AS MAX_RAN
        ,DECIMAL(STDDEV(RAN),8,7)    AS STD_DEV
FROM    (SELECT  RAND(2) AS RAN
         FROM    SYSIBM.SYSCOLUMNS A
                ,SYSIBM.SYSCOLUMNS B
                ,SYSIBM.SYSCOLUMNS C
                ,SYSIBM.SYSCOLUMNS D
         WHERE   A.TBCREATOR       = 'SYSIBM'
           AND   A.TBNAME          = 'SYSTABLES'
           AND   A.COLNO     BETWEEN  1 AND 40
           AND   B.TBCREATOR       = 'SYSIBM'
           AND   B.TBNAME          = 'SYSTABLES'
           AND   B.COLNO     BETWEEN  1 AND 40
           AND   C.TBCREATOR       = 'SYSIBM'
           AND   C.TBNAME          = 'SYSTABLES'
           AND   C.COLNO     BETWEEN  1 AND 40
           AND   D.TBCREATOR       = 'SYSIBM'
           AND   D.TBNAME          = 'SYSTABLES'
           AND   D.COLNO     BETWEEN  1 AND 10
        )AS XXX
FOR FETCH ONLY;
```

```
   ANSWER
   ===================================================================
   OP-SYS   #ROWS  #VALUES   AVG_RAN    MIN_RAN    MAX_RAN    STD_DEV
   ------  ------  -------  ---------  ---------  ---------  ---------
   WIN/NT  640000    32768  0.4997653  0.0000000  1.0000000  0.2883718
   OS/390  640000   640000  0.5000490  0.0000021  0.9999974  0.2888712
```

Figure 709, RAND function - by DB2 flavour

What is Needed

Ideally, DB2 functions should work the same way on all flavors of DB2. With regard to the RAND function, the DB2 for Win/NT version should be changed so that it returns the same results as the DB2 for OS/390 version.

SYSFUN Character Functions

Some of the SYSFUN character functions are anything but fun to use - because their output is of type VARCHAR and length 4000, regardless of the input length. The offenders are:

- INSERT function.
- LEFT function.
- REPEAT function.
- REPLACE function.
- RIGHT function.
- SPACE function.

There used to many more problematic functions, but IBM seems to be slowly addressing the issue by converting them all to type SYSIBM.

STRIP Function

The DB2 STRIP function can remove blank, or non-blank, values from one, or both, ends of a character string. Unfortunately, it only exists in DB2 for OS/390.

DB2BATCH Reliability

For various reasons, I am now using Windows/2000 and DB2 V7.2 to write this book. What a disaster! I typically test thousands of SQL statements in DB2BATCH. Unfortunately, this program tends to freeze DB2 after running a couple of hundred SQL statements when it is used in Windows/2000. I have to reboot my machine to get going again

Appendix

DB2 Sample Tables

Class Schedule

```
CREATE TABLE CL_SCHED
(CLASS_CODE         CHARACTER  (00007)
,DAY                SMALLINT
,STARTING           TIME
,ENDING             TIME);
```

Figure 710, 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 711, 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 712, 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 713, 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 714, 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 715, 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 716, 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 717, 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 718, 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 719, 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 720, 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 721, EMP_RESUME sample table - Data

In Tray

```
CREATE TABLE IN_TRAY
(RECEIVED           TIMESTAMP
,SOURCE             CHARACTER  (00008)
,SUBJECT            CHARACTER  (00064)
,NOTE_TEXT          VARCHAR    (03000));
```

Figure 722, 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 723, 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 724, 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 725, 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 726, PROJECT sample table - Data

Sales

```
CREATE TABLE SALES
(SALES_DATE         DATE
,SALES_PERSON       VARCHAR    (00015)
,REGION             VARCHAR    (00015)
,SALES              INTEGER);
```

Figure 727, SALES sample table - DDL

```
SALES_DATE SALES_PERSON    REGION          SALES
---------- --------------- --------------- -----------
12/31/1995 LUCCHESSI       Ontario-South             1
12/31/1995 LEE             Ontario-South             3
12/31/1995 LEE             Quebec                    1
12/31/1995 LEE             Manitoba                  2
12/31/1995 GOUNOT          Quebec                    1
03/29/1996 LUCCHESSI       Ontario-South             3
03/29/1996 LUCCHESSI       Quebec                    1
03/29/1996 LEE             Ontario-South             2
03/29/1996 LEE             Ontario-North             2
03/29/1996 LEE             Quebec                    3
03/29/1996 LEE             Manitoba                  5
03/29/1996 GOUNOT          Ontario-South             3
03/29/1996 GOUNOT          Quebec                    1
03/29/1996 GOUNOT          Manitoba                  7
03/30/1996 LUCCHESSI       Ontario-South             1
03/30/1996 LUCCHESSI       Quebec                    2
03/30/1996 LUCCHESSI       Manitoba                  1
03/30/1996 LEE             Ontario-South             7
03/30/1996 LEE             Ontario-North             3
03/30/1996 LEE             Quebec                    7
03/30/1996 LEE             Manitoba                  4
03/30/1996 GOUNOT          Ontario-South             2
03/30/1996 GOUNOT          Quebec                   18
03/30/1996 GOUNOT          Manitoba                  1
03/31/1996 LUCCHESSI       Manitoba                  1
03/31/1996 LEE             Ontario-South            14
03/31/1996 LEE             Ontario-North             3
03/31/1996 LEE             Quebec                    7
03/31/1996 LEE             Manitoba                  3
03/31/1996 GOUNOT          Ontario-South             2
03/31/1996 GOUNOT          Quebec                    1
04/01/1996 LUCCHESSI       Ontario-South             3
04/01/1996 LUCCHESSI       Manitoba                  1
04/01/1996 LEE             Ontario-South             8
04/01/1996 LEE             Ontario-North             -
04/01/1996 LEE             Quebec                    8
04/01/1996 LEE             Manitoba                  9
04/01/1996 GOUNOT          Ontario-South             3
04/01/1996 GOUNOT          Ontario-North             1
04/01/1996 GOUNOT          Quebec                    3
04/01/1996 GOUNOT          Manitoba                  7
```

Figure 728, 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 729, 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 730, 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, (½" 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 flavour called *Duro Contact Cement* that is quite a bit better. This is toxic stuff, so be careful.
- CLOTH TAPE, (1" 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

A

ABS function, 71
ACOS function, 72
AGGREGATION function, 62
ALIAS, 15
ALL, sub-query, 149, 159
AND vs. OR, precedence rules, 24
ANY, sub-query, 148, 157
Arithmetic, precedence rules, 24
AS statement
- Correlation name, 18
- Renaming fields, 19

ASCII function, 72
ASIN function, 72
ATAN function, 72
Average Deviation, 223
AVG function, 39, 222

B

Balanced hierarchy, 201
BETWEEN
- AGGREGATION function, 66
- Predicate, 22

BIGINT function, 72, 252
BLOB function, 73

C

Cache, Identity column, 185
Cartesian Product, 135
CASE expression
- Character to Number, 230
- Definition, 35
- Recursive processing, 213
- Sample data creation, usage, 221
- Selective column output, 231
- UPDATE usage, 37
- Wrong sequence, 250
- Zero divide (avoid), 38

CAST expression
- CASE usage, 38
- Definition, 32

CEIL function, 73
CHAR function, 74
Character to Number, 229
Chart making using SQL, 231
CHR function, 75
Circular Reference. *See* You are lost
Clean hierarchies, 209
CLOB function, 76
COALESCE function, 76, 139
Common table expression
- Definition, 26
- Full-select clause, 29

CONCAT function, 77, 204
Convergent hierarchy, 200
Correlated sub-query
- Definition, 154
- NOT EXISTS, 156

CORRELATION function, 41
Correlation name, 18
COS function, 78
COT function, 78
COUNT DISTINCT function
- Definition, 41
- Null values, 52

COUNT function
- Definition, 41
- No rows, 42, 132
- Null values, 41

COUNT_BIG function, 42
COVARIANCE function, 42
Create Table
- Example, 14
- Identity Column, 183
- Summary Table, 166

CUBE, 127

D

Data in view definition, 14
DATE
- Function, 78
- Manipulation, 247
- Output order, 250

DAY function, 79
DAYNAME function, 79
DAYOFWEEK function, 79, 80
DAYOFYEAR function, 80
DAYS function, 80
DB2BATCH reliability, 262
DECIMAL
- Function, 81, 231, 252
- Multiplication, 25, 95

DECRYPT_BIN function, 81
DECRYPT_CHAR function, 81
Deferred Refresh summary tables, 167
Definition Only summary tables, 167
DEGRESS function, 81
Delimiter, 260
Denormalize data, 237
DENSE_RANK function, 50
DIFFERENCE function, 82
DIGITS function, 82
DISTINCT, 39, 70
Divergent hierarchy, 199
DOUBLE function, 83

E

Efficient triggers, summary tables, 176
ENCRYPT function, 83
ESCAPE phrase, 23
EXCEPT, 162

EXISTS, sub-query, 22, 150, 155, 156
EXP function, 84

F

FETCH FIRST clause
- Definition, 17
- Efficient usage, 60
- Stop recursion, 208

FLOAT function, 85, 252
Floating-point numbers, 252
FLOOR function, 85
Fractional date manipulation, 247
Full Outer Join
- COALESCE function, 139
- Definition, 138
- ON processing, 139
- WHERE processing, 140

Full-select
- Definition, 29
- TABLE function, 30
- UPDATE usage, 32

G

GENERATE_UNIQUE function, 85, 218
GETHINT function, 86
GROUP BY
- CUBE, 127
- Definition, 117
- GROUPING SETS, 119
- Join usage, 132
- ORDER BY usage, 131
- PARTITION comparison, 69
- ROLLUP, 123

GROUPING function, 43, 121
GROUPING SETS, 119

H

HEX function, 86, 116, 231, 253
Hierarchy
- Balanced, 201
- Convergent, 200
- Denormalizing, 209
- Divergent, 199
- Find loops, 206
- Recursive, 200
- Summary tables, 209
- Triggers, 209

HOUR function, 87

I

Identity column
- Cache usage, 185
- IDENTITY_VAL_LOCAL function, 187
- Restart value, 184
- Usage notes, 183

IDENTITY_VAL_LOCAL function, 87, 187
Immediate Refresh summary tables, 168
IN
- Predicate, 22
- Sub-query, 153, 155

Index on summary table, 172
Inefficient triggers, summary tables, 173
Inner Join, 135
INSERT
- Common table expression, 28
- Full-select, 31
- Function, 88

INTEGER
- Arithmetic, 24
- Function, 88
- Truncation, 249

INTERSECT, 162

J

Join
- Cartesian Product, 135
- COALESCE function, 139
- DISTINCT usage warning, 39
- Full Outer Join, 138
- GROUP BY usage, 132
- Inner Join, 135
- Left Outer Join, 137
- ON processing, 139
- Right Outer Join, 137
- Syntax, 133
- WHERE processing, 140

JULIAN_DAY function
- Definition, 88
- History, 89

L

LCASE function, 90
LEFT function, 91, 262
Left Outer Join, 137
LENGTH function, 91
LIKE predicate
- Definition, 23
- ESCAPE usage, 23
- Varchar usage, 248

LN function, 92
LOCATE function, 92
LOG function, 92
LOG10 function, 92
Lousy Index. *See* Circular Reference
LTRIM function, 93

M

MAX
- Function, 43
- Rows, getting, 57
- Values, getting, 55, 58

Mean, 222
Median, 223
MICROSECOND function, 93
MIDNIGHT_SECONDS function, 93
MIN function, 44
MINUTE function, 94
Missing rows, 234
MOD function, 94
Mode, 223
Monitor, DB2, 257
MONTH function, 95
MONTHNAME function, 95
MULITPLY_ALT function, 95
Multiplication, overflow, 95

N

NODENUMBER function, 96

Normalize data, 236
NOT EXISTS, sub-query, 154, 156
NOT IN, sub-query, 153, 156
NOT predicate, 21
NULLIF function, 96
Nulls
  CAST expression, 32
  COUNT DISTINCT function, 41, 52
  COUNT function, 156
  Definition, 19
  GROUP BY usage, 118
  Order sequence, 116
  Predicate usage, 24
  Ranking, 52

O

OLAP functions
  AGGREGATION function, 62
  DENSE_RANK function, 50
  RANK function, 50
  ROW_NUMBER function, 56
ON processing, 139
OPTIMIZE FOR clause, 60
OR vs. AND, precedence rules, 24
ORDER BY
  AGGREGATION function, 64
  CONCAT function, 77
  Date usage, 250
  Definition, 115
  FETCH FIRST, 18
  GROUP BY usage, 131
  Nulls processing, 52, 116
  RANK function, 51
  ROW_NUMBER function, 56
Outer Join
  COALESCE function, 139
  Definition, 138
  ON processing, 139
  WHERE processing, 140
Overflow errors, 95

P

Partition
  AGGREGATION function, 69
  GROUP BY comparison, 69
  RANK function, 53
  ROW_NUMBER function, 57
PARTITION function, 96
Pearson P-M Coefficient, 225
Performance analysis, 257
POSSTR function, 97
POWER function, 97
Precedence rules, 24

R

RAISE_ERROR function, 98
RAND function
  Description, 98
  Predicate usage, 245
  Problems, 261
  Random row selection, 101
  Reproducable usage, 99
  Reproducible usage, 217
RANGE (AGGREGATION function), 68
RANK function, 50
REAL function, 101
Recursion
  Fetch first n rows, 61
  Halting processing, 202, 260
  How it works, 191
  Level (in hierarchy), 195
  List children, 194
  Multiple invocations, 197
  Normalize data, 236
  Stopping, 202, 260
  Warning message, 198
  When to use, 191
Recursive hierarchy
  Definition, 200
  Denormalizing, 210, 212
  Triggers, 210, 212
Refresh Deferred summary tables, 167
Refresh Immediate summary tables, 168
REGRESSION functions, 44
REPEAT function, 102, 262
REPLACE function, 102, 241, 262
Restart, Identity column, 184
Reversing values, 238
RIGHT function, 103, 262
Right Outer Join, 137
ROLLUP, 123
ROUND function, 103
ROW_NUMBER function, 56
ROWS (AGGREGATION function), 65
RTRIM function, 103

S

SELECT statement
  Correlation name, 18
  Definition, 15
  Full-select, 31
  Random row selection, 101
  Syntax diagram, 16
Sequence numbers. *See* Identity column
SIGN function, 104
SIN function, 104
SMALLINT function, 104
SOME, sub-query, 148, 157
SOUNDEX function, 104
SPACE function, 105, 262
SQLCACHE_SNAPSHOT function, 106
SQRT function, 106
Standard Deviation. *See* STDDEV function
Standard Error, 225
STDDEV function, 45, 224
Strip
  Functions. *See* LTRIM or RTRIM
  Roll your own, 240
Sub-query
  Correlated, 154
  Error prone, 148
  EXISTS usage, 150, 155
  IN usage, 153, 155
  Multi-field, 155
  Nested, 155
SUBSTR function
  Chart making, 231

Definition, 107
SUM function, 45, 64
Summary tables
Definition, 166
DDL restrictions, 166
Definition Only, 167
Index usage, 172
Null data, 173
Recursive hierarchies, 209
Refresh Deferred, 167
Refresh Immediate, 168
Syntax diagram, 166
Triggers - efficient, 176
Triggers - inefficent, 173

T

Table. *See* Create Table
TABLE function, 30
TABLE_NAME function, 108
TABLE_SCHEMA function, 108
Test Data. *See* Sample Data
Time Series data, 226
TIMESTAMP
Function, 109
Manipulation, 247
TIMESTAMP_ISO function, 109
TIMESTAMPDIFF function, 110
TRANSLATE function, 111
Triggers
Delimiter, 260
Identity column, 186, 189
Recursive hierarchies, 210, 212
Summary tables, 173
TRIM. *See* LTRIM or RTRIM
TRUNCATE function, 112
Truncation, numeric, 249

U

UCASE function, 112
Unbalanced hierarchy, 201
Uncorrelated sub-query, 154
Nested, 155
UNION
Outer join usage, 164
Precedence Rules, 163
Recursion, 192
UNION ALL, 162
View usage, 164
UPDATE
CASE usage, 37
Full-select, 32

V

VALUE function, 113
VALUES expression
Definition, 33
View usage, 35
VARCHAR function, 113
VARIANCE function, 46, 223
VIEW
Data in definition, 14
DDL example, 14, 15, 35
UNION usage, 164
Visual Explain, 259

W

WEEK function, 113, 249
WHERE processing, 24, 140
WITH statement
Defintion, 26
Insert usage, 28
MAX values, getting, 58
Multiple tables, 28
Recursion, 192
VALUES expression, 34

Y

YEAR function, 114
You are lost. *See* Lousy Index

Z

Zero divide (avoid), 38

Reader Comments

If you have any comments about the contents of this book, or any suggestions for future editions, please contact the author at the address shown below. Whenever possible, all comments will receive a written response.

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 V7.2 SQL Cookbook ©
Print:   11 March, 2002
Fname:   BOOK08.DOC
#pages:  277
```

Reader Comments

Name and Address