

DB2 9

SQL Cookbook

Graeme Birchall

20-Feb-2007

Preface

Important!

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

This Cookbook is written for DB2 for Linux, UNIX and Windows. It is **not** suitable for DB2 for z/OS unless you are running DB2 8 in new-function-mode, or (even better) DB2 9.

Acknowledgements

I did not come up with all of the ideas presented in this book. Many of the best examples were provided by readers, friends, and/or coworkers too numerous to list. Thanks also to the many people at IBM for their (strictly unofficial) assistance.

Disclaimer & Copyright

DISCLAIMER: This document is a best effort on my part. However, I screw up all the time, so it would be extremely unwise to trust the contents in its entirety. I certainly don't. And if you do something silly based on what I say, life is tough.

COPYRIGHT: You can make as many copies of this book as you wish. And I encourage you to give it to others. But you cannot charge for it (other than to recover reproduction costs), nor claim the material as your own, nor replace my name with another. You are also encouraged to use the related class notes for teaching. In this case, you can charge for your time and materials - and your expertise. But you cannot charge any licensing fee, nor claim an exclusive right of use. In other words, you can pretty well do anything you want. And if you find the above too restrictive, just let me know.

TRADEMARKS: Lots of words in this document, like "DB2", are registered trademarks of the IBM Corporation. Lots of other words, like "Windows", are registered trademarks of the Microsoft Corporation. Acrobat is a registered trademark of the Adobe Corporation.

Tools Used

This book was written on a Dell PC that came with oodles of RAM. All testing was done in DB2 9. Word for Windows was used to write the document. Adobe Acrobat was used to make the PDF file.

Book Binding

This book looks best when printed on a doubled sided laser printer and then suitably bound. To this end, I did some experiments a few years ago to figure out how to bind books cheaply using commonly available materials. I came up with what I consider to be a very satisfactory solution that is fully documented on page 435.

Author / Book

Author: Graeme Birchall ©
Email: Graeme_Birchall@verizon.net
Web: http://mysite.verizon.net/Graeme_Birchall/
Title: DB2 9 SQL Cookbook ©
Date: 20-Feb-2007

Author Notes

Book History

This book originally began a series of notes for my own use. After a while, friends began to ask for copies, and enemies started to steal it, so I decided to tidy everything up and give it away. Over the years, new chapters have been added as DB2 has evolved, and as I have found new ways to solve problems. Hopefully, this process will continue for the foreseeable future.

Why Free

This book is free because I want people to use it. The more people that use it, and the more that it helps them, the more inclined I am to keep it up to date. For these reasons, if you find this book to be useful, please share it with others.

This book is free, rather than formally published, because I want to deliver the best product that I can. If I had a publisher, I would have the services of an editor and a graphic designer, but I would not be able to get to market so quickly, and when a product changes as quickly as DB2 does, timeliness is important. Also, giving it away means that I am under no pressure to make the book marketable. I simply include whatever I think might be useful.

Other Free Documents

The following documents are also available for free from my web site:

- **SAMPLE SQL:** The complete text of the SQL statements in this Cookbook is available in an HTML file. Only the first and last few lines of the file have HTML tags, the rest is raw text, so it can easily be cut and paste into other files.
- **CLASS OVERHEADS:** Selected SQL examples from this book have been rewritten as class overheads. This enables one to use this material to teach DB2 SQL to others. Use this cookbook as the student notes.
- **OLDER EDITIONS:** This book is rewritten, and usually much improved, with each new version of DB2. Some of the older editions are available from my website. The others can be emailed upon request. However, the latest edition is the best, so you should probably use it, regardless of the version of DB2 that you have.

Answering Questions

As a rule, I do not answer technical questions because I need to have a life. But I'm interested in hearing about interesting SQL problems, and also about any bugs in this book. However you may not get a prompt response, or any response. And if you are obviously an idiot, don't be surprised if I point out (for free, remember) that you are an idiot.

Software Whines

This book is written using Microsoft Word for Windows. I've been using this software for many years, and it has generally been a bunch of bug-ridden junk. I do confess that it has been mildly more reliable in recent years. However, I could have written more than twice as much that was twice as good in half the time - if it weren't for all of the bugs in Word.

Graeme

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 in Postscript Print File format.
- 1998-02-26: The DB2 V2.1.1 SQL Cookbook was converted to an Adobe Acrobat file and posted to my web site. Some minor cosmetic changes were made.
- 1998-08-19: First edition of DB2 UDB V5 SQL Cookbook posted. Every SQL statement was checked for V5, and there were new chapters on OUTER JOIN and GROUP BY.
- 1998-08-26: About 20 minor cosmetic defects were corrected in the V5 Cookbook.
- 1998-09-03: Another 30 or so minor defects were corrected in the V5 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: This book was based on the second edition of the V5.2 upgrade.
- 1999-01-25: A chapter on Summary Tables (new in the Dec/98 fixpack) was added and all the SQL was checked for changes.
- 1999-01-28: Some more SQL was added to the new chapter on Summary Tables.
- 1999-02-15: The section of stopping recursive SQL statements was completely rewritten, and a new section was added on denormalizing hierarchical data structures.
- 1999-02-16: Minor editorial changes were made.
- 1999-03-16: Some bright spark at IBM pointed out that my new and improved section on stopping recursive SQL was all wrong. Damn. I undid everything.
- 1999-05-12: Minor editorial changes were made, and one new example (on getting multiple counts from one value) was added.
- 1999-09-16: DB2 V6.1 edition. All SQL was rechecked, and there were some minor additions - especially to summary tables, plus a chapter on "DB2 Dislikes".
- 1999-09-23: Some minor layout changes were made.
- 1999-10-06: Some bugs fixed, plus new section on index usage in summary tables.
- 2000-04-12: Some typos fixed, and a couple of new SQL tricks were added.
- 2000-09-19: DB2 V7.1 edition. All SQL was rechecked. The new areas covered are: OLAP functions (whole chapter), ISO functions, and identity columns.
- 2000-09-25: Some minor layout changes were made.
- 2000-10-26: More minor layout changes.
- 2001-01-03: Minor layout changes (to match class notes).
- 2001-02-06: Minor changes, mostly involving the RAND function.
- 2001-04-11: Document new features in latest fixpack. Also add a new chapter on Identity Columns and completely rewrite sub-query chapter.
- 2001-10-24: DB2 V7.2 fixpack 4 edition. Tested all SQL and added more examples, plus a new section on the aggregation function.
- 2002-03-11: Minor changes, mostly to section on precedence rules.
- 2002-08-20: DB2 V8.1 (beta) edition. A few new functions are added. New section on temporary tables. Identity Column and Join chapters rewritten. Whine chapter removed.

- 2003-01-02: DB2 V8.1 (post-Beta) edition. SQL rechecked. More examples added.
- 2003-07-11: New sections added on DML, temporary tables, compound SQL, and user defined functions. Halting recursion section changed to use user-defined function.
- 2003-09-04: New sections on complex joins and history tables.
- 2003-10-02: Minor changes. Some more user-defined functions.
- 2003-11-20: Added "quick find" chapter.
- 2003-12-31: Tidied up the SQL in the Recursion chapter, and added a section on the merge statement. Completely rewrote the chapter on materialized query tables.
- 2004-02-04: Added select-from-DML section, and tidied up some code. Also managed to waste three whole days due to bugs in Microsoft Word.
- 2004-07-23: Rewrote chapter of identity column and sequences. Made DML separate chapter. Added chapters on protecting data and XML functions. Other minor changes.
- 2004-11-03: Upgraded to V8.2. Retested all SQL. Documented new SQL features. Some major hacking done on the GROUP BY chapter.
- 2005-04-15: Added short section on cursors, and a chapter on using SQL to make SQL.
- 2005-06-01: Added a chapter on triggers.
- 2005-11-11: Updated MQT table chapter and added bibliography. Other minor changes.
- 2005-12-01: Applied fixpack 10. Changed my website name.
- 2005-12-16: Added notes on isolation levels, data-type functions, transforming data.
- 2006-01-26: Fixed dumb bugs generated by WORD. What stupid software. Also wrote an awesome new section on joining meta-data to real data.
- 2006-02-17: Touched up the section on joining meta-data to real data. Other minor fixes.
- 2006-02-27: Added precedence rules for SQL statement processing, and a description of a simplified nested table expression.
- 2006-03-23: Added better solution to avoid fetching the same row twice.
- 2006-04-26: Added trigger that can convert HEX value to number.
- 2006-09-08: Upgraded to V9.1. Retested SQL. Removed the XML chapter as it is now obsolete. I'm still cogitating about XQuery. Looks hard. Added some awesome java code.
- 2006-09-13: Fixed some minor problems in the initial V9.1 book.
- 2006-10-17: Fixed a few cosmetic problems that were bugging me.
- 2006-11-06: Found out that IBM had removed the "UDB" from the DB2 product name, so I did the same. It is now just plain "DB2 V9".
- 2006-11-29: I goofed. Turns out DB2 is now called "DB2 9". I relabeled accordingly.
- 2006-12-15: Improved code to update or delete first "n" rows.
- 2007-02-22: Get unique timestamp values during multi-row insert. Other minor changes.

Table of Contents

PREFACE	3
AUTHOR NOTES	4
BOOK EDITIONS	5
TABLE OF CONTENTS	7
QUICK FIND	15
Index of Concepts.....	15
INTRODUCTION TO SQL	19
Syntax Diagram Conventions.....	19
SQL Components	20
DB2 Objects.....	20
DB2 Data Types.....	22
Date/Time Arithmetic.....	23
DB2 Special Registers.....	25
Distinct Types.....	27
Fullselect, Subselect, & Common Table Expression.....	28
SELECT Statement.....	29
FETCH FIRST Clause.....	31
Correlation Name.....	32
Renaming Fields.....	32
Working with Nulls.....	33
Quotes and Double-quotes.....	34
SQL Predicates	34
Basic Predicate.....	35
Quantified Predicate.....	35
BETWEEN Predicate.....	36
EXISTS Predicate.....	36
IN Predicate.....	37
LIKE Predicate.....	37
NULL Predicate.....	39
Special Character Usage.....	39
Precedence Rules.....	39
Processing Sequence.....	40
CAST Expression	41
VALUES Clause	42
CASE Expression	44
CASE Syntax Styles.....	44
Sample SQL.....	45
Miscellaneous SQL Statements	48
Cursor.....	48
Select Into.....	50
Prepare.....	50
Describe.....	51
Execute.....	52
Execute Immediate.....	52
Set Variable.....	52
Set DB2 Control Structures.....	52
Unit-of-Work Processing	53
Commit.....	53
Savepoint.....	53
Release Savepoint.....	54

Rollback	54
DATA MANIPULATION LANGUAGE.....	57
Insert.....	57
Update.....	61
Delete	64
Select DML Changes	66
Merge	69
COMPOUND SQL.....	75
Introduction.....	75
Statement Delimiter	75
SQL Statement Usage	76
DECLARE Variables.....	76
FOR Statement	77
GET DIAGNOSTICS Statement	77
IF Statement.....	78
ITERATE Statement	78
LEAVE Statement	79
SIGNAL Statement.....	79
WHILE Statement.....	80
Other Usage	80
Trigger.....	81
Scalar Function	82
Table Function.....	83
COLUMN FUNCTIONS	85
Introduction	85
Column Functions, Definitions.....	85
AVG.....	85
CORRELATION	87
COUNT	87
COUNT_BIG	88
COVARIANCE.....	88
GROUPING.....	89
MAX	89
MIN	90
REGRESSION	90
STDDEV.....	91
SUM	91
VAR or VARIANCE.....	92
OLAP FUNCTIONS	93
Introduction.....	93
OLAP Functions, Definitions	96
Ranking Functions.....	96
Row Numbering Function	102
Aggregation Function	108
SCALAR FUNCTIONS.....	117
Introduction	117
Sample Data.....	117
Scalar Functions, Definitions	117
ABS or ABSVAL.....	117
ACOS	118
ASCII.....	118
ASIN.....	118
ATAN	118
ATANH.....	118
ATAN2.....	118
BIGINT	118
BLOB.....	119
CEIL or CEILING	119
CHAR	120
CHR	122

CLOB	123
COALESCE	123
CONCAT	124
COS	124
COSH	125
COT	125
DATE	125
DAY	126
DAYNAME	126
DAYOFWEEK	126
DAYOFWEEK_ISO	127
DAYOFYEAR	127
DAYS	127
DBCLOB	128
DBPARTITIONNUM	128
DEC or DECIMAL	128
DEGREES	129
DEREF	129
DECRYPT_BIN and DECRYPT_CHAR	129
DIFFERENCE	129
DIGITS	130
DLCOMMENT	130
DLINKTYPE	130
DLNEWCOPY	130
DLPREVIOUSCOPY	130
DLREPLACECONTENT	130
DLURLCOMPLETE	131
DLURLCOMPLETEONLY	131
DLURLCOMPLETEWRITE	131
DLURLPATH	131
DLURLPATHONLY	131
DLURLPATHWRITE	131
DLURLSCHEME	131
DLURLSERVER	131
DLVALUE	131
DOUBLE or DOUBLE_PRECISION	131
ENCRYPT	132
EVENT_MON_STATE	132
EXP	132
FLOAT	133
FLOOR	133
GENERATE_UNIQUE	133
GETHINT	135
GRAPHIC	135
HASHEDVALUE	135
HEX	135
HOUR	136
IDENTITY_VAL_LOCAL	136
INSERT	137
INT or INTEGER	137
JULIAN_DAY	137
LCASE or LOWER	139
LEFT	140
LENGTH	140
LN or LOG	140
LOCATE	140
LOG or LN	141
LOG10	141
LONG_VARCHAR	141
LONG_VARGRAPHIC	141
LOWER	141
LTRIM	141
MICROSECOND	142
MIDNIGHT_SECONDS	142
MINUTE	142
MOD	143
MONTH	143
MONTHNAME	143
MULTIPLY_ALT	143
NULLIF	144
PARTITION	144
POSSTR	144
POWER	145
QUARTER	145
RADIANS	145
RAISE_ERROR	145

RAND	146
REAL	149
REPEAT	149
REPLACE	150
RIGHT	150
ROUND	150
RTRIM	151
SECOND	151
SIGN	151
SIN	151
SINH	152
SMALLINT	152
SNAPSHOT Functions	152
SOUNDEX	152
SPACE	153
SQRT	153
STRIP	153
SUBSTR	154
TABLE	155
TABLE_NAME	155
TABLE_SCHEMA	156
TAN	156
TANH	156
TIME	156
TIMESTAMP	156
TIMESTAMP_FORMAT	157
TIMESTAMP_ISO	157
TIMESTAMPDIFF	158
TO_CHAR	158
TO_DATE	159
TRANSLATE	159
TRIM	160
TRUNC or TRUNCATE	160
TYPE_ID	160
TYPE_NAME	160
TYPE_SCHEMA	160
UCASE or UPPER	160
VALUE	160
VARCHAR	161
VARCHAR_FORMAT	161
VARGRAPHIC	161
VEBLOB_CP_LARGE	161
VEBLOB_CP_LARGE	161
WEEK	161
WEEK_ISO	162
YEAR	162
"+" PLUS	162
"-" MINUS	163
** MULTIPLY	163
/" DIVIDE	163
" " CONCAT	164
USER DEFINED FUNCTIONS	165
Sourced Functions	165
Scalar Functions	167
Description	167
Examples	168
Table Functions	172
Description	172
Examples	173
Useful User-Defined Functions	174
Julian Date Functions	174
Get Prior Date	174
Generating Numbers	176
Check Data Value Type	177
ORDER BY, GROUP BY, AND HAVING	181
Order By	181
Notes	181
Sample Data	181
Order by Examples	182

Group By and Having	184
Rules and Restrictions	184
GROUP BY Flavors	185
GROUP BY Sample Data	186
Simple GROUP BY Statements	186
GROUPING SETS Statement	187
ROLLUP Statement	191
CUBE Statement.....	195
Complex Grouping Sets - Done Easy	198
Group By and Order By	200
Group By in Join	200
COUNT and No Rows.....	201
JOINS	203
Why Joins Matter	203
Sample Views	203
Join Syntax.....	203
Query Processing Sequence	205
ON vs. WHERE	205
Join Types	206
Inner Join	206
Left Outer Join	207
Right Outer Join	209
Full Outer Joins	210
Cartesian Product	214
Join Notes	216
Using the COALESCE Function	216
Listing non-matching rows only	216
Join in SELECT Phrase	218
Predicates and Joins, a Lesson	220
Joins - Things to Remember	221
Complex Joins	222
SUB-QUERY	225
Sample Tables.....	225
Sub-query Flavours	225
Sub-query Syntax	225
Correlated vs. Uncorrelated Sub-Queries	232
Multi-Field Sub-Queries	233
Nested Sub-Queries	233
Usage Examples	234
True if NONE Match.....	234
True if ANY Match.....	235
True if TEN Match.....	236
True if ALL match.....	237
UNION, INTERSECT, AND EXCEPT	239
Syntax Diagram	239
Sample Views	239
Usage Notes	240
Union & Union All.....	240
Intersect & Intersect All	240
Except & Except All.....	240
Precedence Rules.....	241
Unions and Views	242
MATERIALIZED QUERY TABLES	243
Introduction	243
Usage Notes	243
Syntax Options	244
Select Statement.....	245
Optimizer Options	246
Refresh Deferred Tables.....	248
Refresh Immediate Tables	249
Usage Notes and Restrictions.....	251
Multi-table Materialized Query Tables	252
Indexes on Materialized Query Tables	254
Organizing by Dimensions	255

Using Staging Tables	255
IDENTITY COLUMNS AND SEQUENCES.....	257
Identity Columns.....	257
Rules and Restrictions.....	258
Altering Identity Column Options	261
Gaps in Identity Column Values	262
IDENTITY_VAL_LOCAL Function	263
Sequences	265
Getting the Sequence Value.....	265
Multi-table Usage	267
Counting Deletes	269
Identity Columns vs. Sequences - a Comparison	269
Roll Your Own	270
Support Multi-row Inserts.....	271
TEMPORARY TABLES.....	275
Introduction.....	275
Temporary Tables - in Statement	277
Common Table Expression	278
Full-Select	280
Declared Global Temporary Tables	284
RECURSIVE SQL	287
Use Recursion To.....	287
When (Not) to Use Recursion	287
How Recursion Works.....	287
List Dependents of AAA	288
Notes & Restrictions.....	289
Sample Table DDL & DML	289
Introductory Recursion	290
List all Children #1	290
List all Children #2.....	290
List Distinct Children.....	291
Show Item Level.....	291
Select Certain Levels.....	292
Select Explicit Level.....	293
Trace a Path - Use Multiple Recursions	293
Extraneous Warning Message.....	294
Logical Hierarchy Flavours.....	295
Divergent Hierarchy.....	295
Convergent Hierarchy.....	296
Recursive Hierarchy	296
Balanced & Unbalanced Hierarchies	297
Data & Pointer Hierarchies	297
Halting Recursive Processing	298
Sample Table DDL & DML	298
Stop After "n" Levels.....	299
Stop When Loop Found.....	300
Keeping the Hierarchy Clean.....	303
Clean Hierarchies and Efficient Joins	305
Introduction	305
Limited Update Solution	305
Full Update Solution	307
TRIGGERS	311
Trigger Syntax.....	311
Usage Notes.....	311
Trigger Usage.....	312
Trigger Examples.....	313
Sample Tables	313
Before Row Triggers - Set Values	313
Before Row Trigger - Signal Error.....	314
After Row Triggers - Record Data States.....	314
After Statement Triggers - Record Changes.....	315

Examples of Usage.....	316
PROTECTING YOUR DATA.....	319
Sample Application	319
Enforcement Tools.....	320
Distinct Data Types.....	321
Customer-Balance Table	321
US-Sales Table.....	322
Triggers	322
Conclusion.....	326
RETAINING A RECORD	327
Schema Design	327
Recording Changes	327
Multiple Versions of the World	330
USING SQL TO MAKE SQL	337
Export Command	337
SQL to Make SQL.....	338
RUNNING SQL WITHIN SQL.....	341
Introduction.....	341
Generate SQL within SQL.....	341
Make Query Column-Independent	342
Business Uses.....	343
Meta Data Dictionaries.....	344
DB2 SQL Functions	344
Function and Stored Procedure Used	344
Different Data Types.....	345
Usage Examples.....	346
Java Functions.....	348
Scalar Functions	348
Tabular Functions	349
Transpose Function	352
Update Real Data using Meta-Data	359
Usage Examples.....	360
FUN WITH SQL	365
Creating Sample Data.....	365
Data Generation	365
Make Reproducible Random Data	365
Make Random Data - Different Ranges	366
Make Random Data - Varying Distribution	366
Make Random Data - Different Flavours.....	367
Make Test Table & Data	367
Time-Series Processing	369
Find Overlapping Rows.....	370
Find Gaps in Time-Series	371
Show Each Day in Gap.....	372
Other Fun Things.....	372
Randomly Sample Data	372
Convert Character to Numeric	374
Convert Number to Character.....	376
Convert Timestamp to Numeric.....	379
Selective Column Output	380
Making Charts Using SQL.....	380
Multiple Counts in One Pass.....	381
Find Missing Rows in Series / Count all Values	382
Multiple Counts from the Same Row.....	383
Normalize Denormalized Data	384
Denormalize Normalized Data	385
Transpose Numeric Data.....	387
Reversing Field Contents.....	390
Fibonacci Series	391
Business Day Calculation	393
Query Runs for "n" Seconds	393

Sort Character Field Contents	394
Calculating the Median	396
Converting HEX Data to Number	399
QUIRKS IN SQL	403
Trouble with Timestamps	403
No Rows Match	404
Dumb Date Usage	405
RAND in Predicate	406
Date/Time Manipulation	408
Use of LIKE on VARCHAR	409
Comparing Weeks	410
DB2 Truncates, not Rounds	410
CASE Checks in Wrong Sequence	411
Division and Average	411
Date Output Order	411
Ambiguous Cursors	412
Multiple User Interactions	413
Floating Point Numbers	416
Legally Incorrect SQL	419
APPENDIX	421
DB2 Sample Tables	421
ACT	421
CATALOG	421
CL_SCHED	422
CUSTOMER	422
DEPARTMENT	422
EMP_PHOTO	423
EMP_RESUME	423
EMPLOYEE	424
EMPMDC	425
EMPPROJACT	426
IN_TRAY	427
INVENTORY	427
ORG	428
PRODUCT	428
PRODUCTSUPPLIER	428
PROJACT	429
PROJECT	430
PURCHASEORDER	431
SALES	431
STAFF	432
SUPPLIERS	433
BOOK BINDING	435
BIBLIOGRAPHY	437
IBM Sources	437
DB2 Manuals	437
Red Books	437
Online Tutorials	438
Other Sources	438
Books Published	438
Roger Sanders Books	438
DB2 Magazine	438
INDEX	439

Quick Find

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

Index of Concepts

Join Rows

To combine matching rows in multiple tables, use a join (see page 203).

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

Figure 1, Join example

Outer Join

To get all of the rows from one table, plus the matching rows from another table (if there are any), use an outer join (see page 206).

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

Figure 2, Left-outer-join example

To get rows from either side of the join, regardless of whether they match (the join) or not, use a full outer join (see page 210).

Null Values - Replace

Use the COALESCE function (see page 123) to replace a null value (e.g. generated in an outer join) with a non-null value.

Select Where No Match

To get the set of the matching rows from one table where something is true or false in another table (e.g. no corresponding row), use a sub-query (see page 225).

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

Figure 3, Sub-query example

Append Rows

To add (append) one set of rows to another set of rows, use a union (see page 239).

```

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

```

Figure 4, Union example

Assign Output Numbers

To assign line numbers to SQL output, use the ROW_NUMBER function (see page 102).

```

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

```

Figure 5, Assign row-numbers example

Assign Unique Key Numbers

The make each row inserted into a table automatically get a unique key value, use an identity column, or a sequence, when creating the table (see page 257).

If-Then-Else Logic

To include if-then-else logical constructs in SQL stmts, use the CASE phrase (see page 44).

```

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

```

Figure 6, Case stmt example

Get Dependents

To get all of the dependents of some object, regardless of the degree of separation from the parent to the child, use recursion (see page 287).

```

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

```

Figure 7, Recursion example

Convert String to Rows

To convert a (potentially large) set of values in a string (character field) into separate rows (e.g. one row per word), use recursion (see page 384).


```

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

Figure 8, Convert string to rows

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

Convert Rows to String

To convert a (potentially large) set of values that are in multiple rows into a single combined field, use recursion (see page 385).

```

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

Figure 9, Convert rows to string

Fetch First "n" Rows

To fetch the first "n" matching rows, use the FETCH FIRST notation (see page 31).

```

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

Figure 10, Fetch first "n" rows example

Another way to do the same thing is to assign row numbers to the output, and then fetch those rows where the row-number is less than "n" (see page 103).

Fetch Subsequent "n" Rows

To the fetch the "n" through "n + m" rows, first use the ROW_NUMBER function to assign output numbers, then put the result in a nested-table-expression, and then fetch the rows with desired numbers (see page 103).

Fetch Uncommitted Data

To retrieve data that may have been changed by another user, but which they have yet to commit, use the WITH UR (Uncommitted Read) notation.

```

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

Figure 11, Fetch WITH UR example

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

Summarize Column Contents

Use a column function (see page 85) to summarize the contents of a column.

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

Figure 12, Column Functions example

Subtotals and Grand Totals

To obtain subtotals and grand-totals, use the ROLLUP or CUBE statements (see page 191).

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

Figure 13, Subtotal and Grand-total example

Enforcing Data Integrity

When a table is created, various DB2 features can be used to ensure that the data entered in the table is always correct:

- Uniqueness (of values) can be enforced by creating unique indexes.
- Check constraints can be defined to limit the values that a column can have.
- Default values (for a column) can be defined - to be used when no value is provided.
- Identity columns (see page 257), can be defined to automatically generate unique numeric values (e.g. invoice numbers) for all of the rows in a table. Sequences can do the same thing over multiple tables.
- Referential integrity rules can be created to enforce key relationships between tables.
- Triggers can be defined to enforce more complex integrity rules, and also to do things (e.g. populate an audit trail) whenever data is changed.

See the DB2 manuals for documentation or page 319 for more information about the above.

Hide Complex SQL

One can create a view (see page 20) to hide complex SQL that is run repetitively. Be warned however that doing so can make it significantly harder to tune the SQL - because some of the logic will be in the user code, and some in the view definition.

Summary Table

Some queries that use a GROUP BY can be made to run much faster by defining a summary table (see page 243) that DB2 automatically maintains. Subsequently, when the user writes the original GROUP BY against the source-data table, the optimizer substitutes with a much simpler (and faster) query against the summary table.

Introduction to SQL

This chapter contains a basic introduction to DB2 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 SQL statements. The following diagram shows the conventions used.

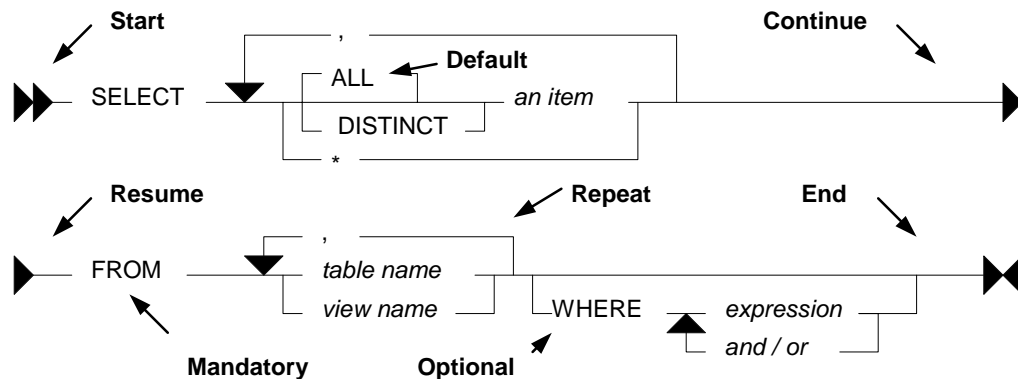


Figure 14, Syntax Diagram Conventions

Rules

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

SQL Comments

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

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

Figure 15, SQL Comment example

Some DB2 command processors (e.g. DB2BATCH on the PC, or SPUFI on the mainframe) can process intelligent comments. These begin the line with a "--#SET" phrase, and then identify the value to be set. In the following example, the statement delimiter is changed using an intelligent comment:

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

Figure 16, Set Delimiter example

When using the DB2 Command Processor (batch) script, the default statement terminator can be set using the "-tdx" option, where "x" is the value have chosen.

NOTE: See the section titled Special Character Usage on page 39 for notes on how to refer to the statement delimiter in the SQL text.

Statement Delimiter

DB2 SQL does not come with a designated statement delimiter (terminator), though a semi-colon is often used. A semi-colon cannot be used when writing a compound SQL statement (see page 75) because that character is used to terminate the various sub-components of the statement.

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
,firstme 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 17, 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 18, 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 19, Define a view using a VALUES clause

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

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

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

Figure 20, 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
(VVALUES (1)
 UNION ALL
 SELECT num1 + 1
 FROM   temp1
 WHERE  num1 < 10000)
SELECT *
FROM   temp1;
```

Figure 21, 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 22, 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.

Nickname

A nickname is the name that one provides to DB2 for either a remote table, or a non-relational object that one wants to query as if it were a table.

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

Figure 23, Define a nickname

Tablesample

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

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

Figure 24, TABLESAMPLE example

See page 372 for information on using the TABLESAMPLE feature.

DB2 Data Types

DB2 comes with the following standard data types:

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

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

```
CREATE TABLE sales_record
(sales#          INTEGER          NOT NULL
GENERATED ALWAYS AS IDENTITY
(START WITH 1
,INCREMENT BY 1
,NO MAXVALUE
,NO CYCLE)
,sale_ts        TIMESTAMP          NOT NULL
,num_items      SMALLINT          NOT NULL
,payment_type   CHAR(2)           NOT NULL
,sale_value     DECIMAL(12,2)     NOT NULL
,sales_tax      DECIMAL(12,2)
,employee#     INTEGER          NOT NULL
,CONSTRAINT sales1 CHECK(payment_type IN ('CS','CR'))
,CONSTRAINT sales2 CHECK(sale_value > 0)
,CONSTRAINT sales3 CHECK(num_items > 0)
,CONSTRAINT sales4 FOREIGN KEY(employee#)
REFERENCES staff(id)
ON DELETE RESTRICT
,PRIMARY KEY(sales#));
```

Figure 25, Sample table definition

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

- The sales# is automatically generated (see page 257 for details). It is also the primary key of the table, and so must always be unique.
- The payment-type must be one of two possible values.
- Both the sales-value and the num-items must be greater than zero.
- The employee# must already exist in the staff table. Furthermore, once a row has been inserted into this table, any attempt to delete the related row from the staff table will fail.

Default Lengths

The following table has two columns:

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

Figure 26, Table with default column lengths

The length has not been provided for either of the above columns. In this case, DB2 defaults to CHAR(1) for the first column and DECIMAL(5,0) for the second column.

Data Type Usage

In general, use the standard DB2 data types as follows:

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

A DB2 data type is not just a place to hold data. It also defines what rules are applied when the data is manipulated. For example, storing monetary data in a DB2 floating-point field is a no-no, in part because the data-type is not precise, but also because a floating-point number is not manipulated (e.g. during division) according to internationally accepted accounting rules.

Date/Time Arithmetic

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

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

The valid labeled durations are listed below:

Labeled Durations		Item	Works with Date/Time		
Singular	Plural	Fixed Size	Date	Time	Timestamp
YEAR	YEARS	N	Y	-	Y
MONTH	MONTHS	N	Y	-	Y
DAY	DAYS	Y	Y	-	Y
HOUR	HOURS	Y	-	Y	Y
MINUTE	MINUTES	Y	-	Y	Y
SECOND	SECONDS	Y	-	Y	Y
MICROSECOND	MICROSECONDS	Y	-	Y	Y

Figure 27, Labeled Durations and Date/Time Types

Usage Notes

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

Now for some examples:

```

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

```

ANSWER
=====

```

<= 1995-12-31
<= 1995-12-21
<= 1995-11-30
<= 2094-12-31
<= 1994-04-24
<= 1996-01-10

```

Figure 28, Example, Labeled Duration usage

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


```

                                ANSWER
                                =====
SELECT  sales_date              <= 1995-12-31
        ,sales_date + 2 MONTH AS d1 <= 1996-02-29
        ,sales_date + 3 MONTHS AS d2 <= 1996-03-31
        ,sales_date + 2 MONTH
        + 1 MONTH AS d3           <= 1996-03-29
        ,sales_date + (2+1) MONTHS AS d4 <= 1996-03-31
FROM    sales
WHERE   sales_person = 'GOUNOT'
AND    sales_date = '1995-12-31';

```

Figure 29, Adding Months - Varying Results

Date/Time Duration Usage

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

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

Figure 30, Date/Time Durations

Below is an example of date duration generation:

```

SELECT  empno                    ANSWER
        ,hiredate                =====
        ,birthdate                EMPNO  HIREDATE  BIRTHDATE
        ,hiredate - birthdate    -----
FROM    employee                000150 1972-02-12 1947-05-17 240826.
WHERE   workdept = 'D11'        000200 1966-03-03 1941-05-29 240905.
AND    lastname < 'L'          000210 1979-04-11 1953-02-23 260116.
ORDER BY empno;

```

Figure 31, Date Duration Generation

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

```

                                ANSWER
                                =====
SELECT  hiredate                <= 1972-02-12
        ,hiredate - 12345678.   <= 0733-03-26
        ,hiredate - 1234 years
        - 56 months
        - 78 days                <= 0733-03-26
FROM    employee
WHERE   empno = '000150';

```

Figure 32, Subtracting a Date Duration

Date/Time Subtraction

One date/time can be subtracted (only) from another valid date/time value. The result is a date/time duration value. Figure 31 above has an example.

DB2 Special Registers

A special register is a DB2 variable that contains information about the state of the system. The complete list follows:

SPECIAL REGISTER	UPDATE	DATA-TYPE
=====	=====	=====
CURRENT CLIENT_ACCTNG	no	VARCHAR(255)
CURRENT CLIENT_APPLNAME	no	VARCHAR(255)
CURRENT CLIENT_USERID	no	VARCHAR(255)
CURRENT CLIENT_WRKSTNNAME	no	VARCHAR(255)
CURRENT DATE	no	DATE
CURRENT DBPARTITIONNUM	no	INTEGER
CURRENT DEFAULT TRANSFORM GROUP	yes	VARCHAR(18)
CURRENT DEGREE	yes	CHAR(5)
CURRENT EXPLAIN MODE	yes	VARCHAR(254)
CURRENT EXPLAIN SNAPSHOT	yes	CHAR(8)
CURRENT FEDERATED ASYNCHRONY	yes	INTEGER
CURRENT IMPLICIT XMLPARSE OPTION	yes	VARCHAR(128)
CURRENT ISOLATION	yes	CHAR(2)
CURRENT LOCK TIMEOUT	yes	INTEGER
CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION	yes	VARCHAR(254)
CURRENT PACKAGE PATH	yes	VARCHAR(4096)
CURRENT PATH	yes	VARCHAR(254)
CURRENT QUERY OPTIMIZATION	yes	INTEGER
CURRENT REFRESH AGE	yes	DECIMAL(20,6)
CURRENT SCHEMA	yes	VARCHAR(128)
CURRENT SERVER	no	VARCHAR(18)
CURRENT TIME	no	TIME
CURRENT TIMESTAMP	no	TIMESTAMP
CURRENT TIMEZONE	no	DECIMAL(6,0)
CURRENT USER	no	VARCHAR(128)
SESSION_USER	yes	VARCHAR(128)
SYSTEM_USER	no	VARCHAR(128)
USER	yes	VARCHAR(128)

Figure 33, DB2 Special Registers

Usage Notes

- Some special registers can be referenced using an underscore instead of a blank in the name - as in: CURRENT_DATE.
- Some special registers can be updated using the SET command (see list above).
- All special registers can be queried using the SET command. They can also be referenced in ordinary SQL statements.
- Those special registers that automatically change over time (e.g. current timestamp) are always the same for the duration of a given SQL statement. So if one inserts a thousand rows in a single insert, all will get the same current timestamp.
- One can reference the current timestamp in an insert or update, to record in the target table when the row was changed. To see the value assigned, query the DML statement. See page 66 for details.

Refer to the DB2 SQL Reference Volume 1 for a detailed description of each register.

Sample SQL

```

SET CURRENT ISOLATION = RR;
SET CURRENT SCHEMA    = 'ABC';

SELECT  CURRENT TIME           AS cur_TIME
        ,CURRENT ISOLATION    AS cur_ISO
        ,CURRENT SCHEMA       AS cur_ID
FROM    sysibm.sysdummy1;

```

ANSWER

```

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

```

Figure 34, Using Special Registers

Distinct Types

A distinct data type is a field type that is derived from one of the base DB2 field types. It is used when one wants to prevent users from combining two separate columns that should never be manipulated together (e.g. adding US dollars to Japanese Yen).

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

```
► CREATE DISTINCT TYPE type-name AS source-type WITH COMPARISONS ►
```

Figure 35, Create Distinct Type Syntax

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

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

Below is a typical create and drop statement:

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

Figure 36, Create and drop distinct type

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

Usage Example

Imagine that we had the following customer table:

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

Figure 37, Sample table, without distinct types

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

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

Figure 38, Silly query, but works

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

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

Figure 39, Create Distinct Type examples

Now we can define the customer table thus:

```

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

```

Figure 40, Sample table, with distinct types

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

```

SELECT   id
        ,usa_sales + eur_sales AS tot_sales
FROM     customer;

```

Figure 41, Silly query, now fails

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

```

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

```

Figure 42, Silly query, works again

Fullselect, Subselect, & Common Table Expression

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

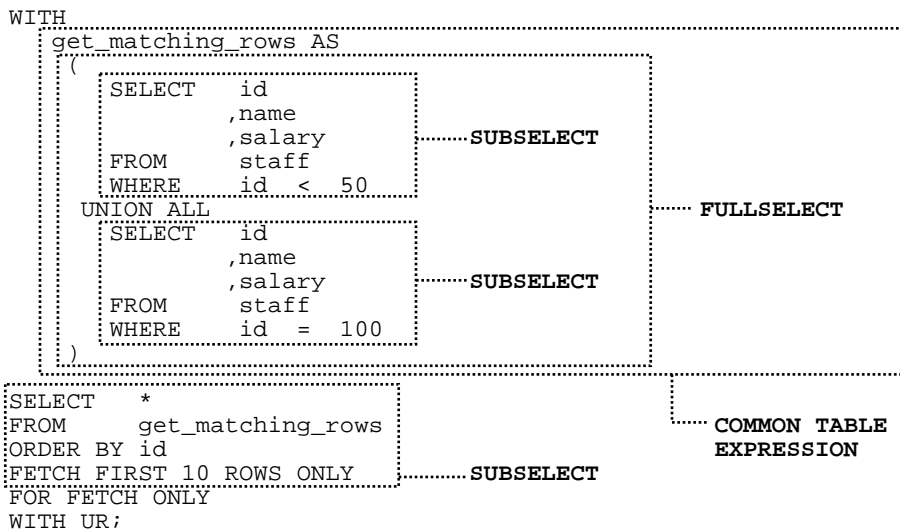


Figure 43, Query components

Query Components

- **SUBSELECT**: A query that selects zero or more rows from one or more tables.

- **FULLSELECT:** One or more subselects or VALUES clauses, connected using a UNION, INTERSECT, or EXCEPT, all enclosed in parenthesis.
- **COMMON TABLE EXPRESSION:** A named fullselect that can be referenced one more times in another subselect. See page 278 for a more complete definition.

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 278).
- **ORDER BY clause.** Optionally, order the final output (see page 181).
- **FETCH FIRST clause.** Optionally, stop the query after "n" rows (see page 31). 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.

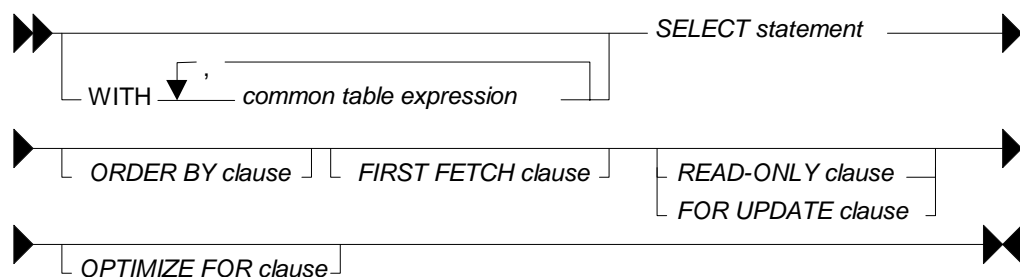
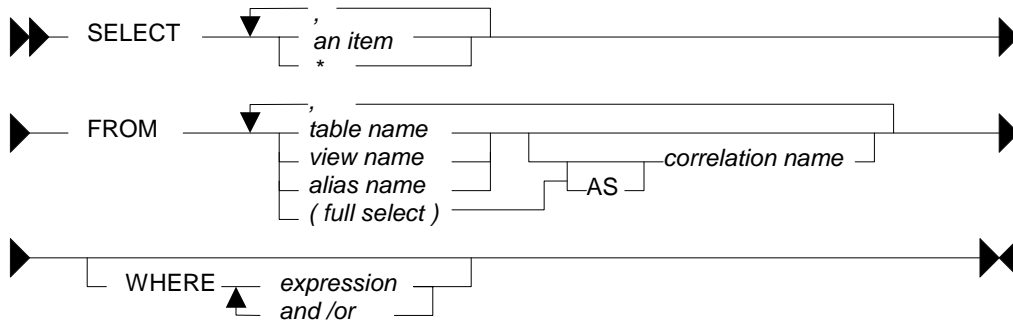


Figure 44, *SELECT Statement Syntax (general)*

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 45, *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 46, *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 47, *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
      ,department.*
FROM department
WHERE deptname LIKE '%ING%'
ORDER BY 1;
    
```

ANSWER (part of)
=====
DEPTNO DEPTNO etc...
----->>>
B01 B01 PLANNING
D11 D11 MANUFACTU

Figure 48, Select an individual column, and all columns

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

```

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

ANSWER (part of)
=====
DEPTNO etc...
----->>>
B01 PLANNING

Figure 49, 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.

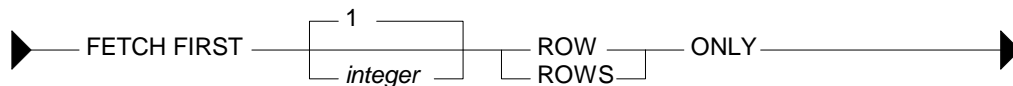


Figure 50, Fetch First clause Syntax

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

ANSWER
=====
YEARS NAME ID
----->>>
7 Sanders 10
8 Pernal 20
5 Marenghi 30

Figure 51, 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
      ,name
      ,id
FROM staff
WHERE years IS NOT NULL
ORDER BY years DESC
FETCH FIRST 3 ROWS ONLY;
    
```

ANSWER
=====
YEARS NAME ID
----->>>
13 Graham 310
12 Jones 260
10 Hanes 50

Figure 52, 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
           ,name
           ,id
FROM        staff
WHERE       years IS NOT NULL
ORDER BY   years DESC
           ,id   DESC
FETCH FIRST 3 ROWS ONLY;

```

ANSWER		
=====		
YEARS	NAME	ID

13	Graham	310
12	Jones	260
10	Quill	290

Figure 53, *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 104 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
           ,a.lastname
FROM        employee a
           ,(SELECT MAX(empno)AS empno
            FROM   employee) AS b
WHERE       a.empno = b.empno;

```

ANSWER	
=====	
EMPNO	LASTNAME

000340	GOUNOT

Figure 54, *Correlation Name usage example*

```

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

```

ANSWER		
=====		
EMPNO	LASTNAME	DEPT

000090	HENDERSON	E11
000280	SCHNEIDER	E11
000290	PARKER	E11
000300	SMITH	E11
000310	SETRIGHT	E11

Figure 55, *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
           ,midinit AS "m int"
           ,phoneno AS "..."
FROM        employee
WHERE       empno < '000030'
ORDER BY 1;

```

ANSWER		
=====		
E_NUM	M INT	...

000010	I	3978
000020	L	3476

Figure 56, *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 fullselect 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;

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

ANSWER		
=====		
E_NUM	M INT	...

000010	I	3978

Figure 57, 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
        ,SUM(comm) / COUNT(*) AS a2
FROM     staff
WHERE    id < 100;
```

ANSWER	
=====	
A1	A2

796.025	530.68

Figure 58, 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
        ,MAX(lastname) AS max
FROM     employee
WHERE    firstnme = 'FRED';
```

ANSWER	
=====	
NUM	MAX

0	-

Figure 59, 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
WHERE       id < 100
            AND comm IS NOT NULL;
            796.025   796.02

```

Figure 60, AVG of those rows that are not null

Quotes and Double-quotes

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

```

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

```

Figure 61, Quote usage

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

```

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

```

Figure 62, Double-quote usage

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

SQL Predicates

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

Predicate Precedence

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

- Predicates separated by an OR may need parenthesis - see page 40.
- Checks specified in a CASE statement are done in the order written - see page 46.

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.

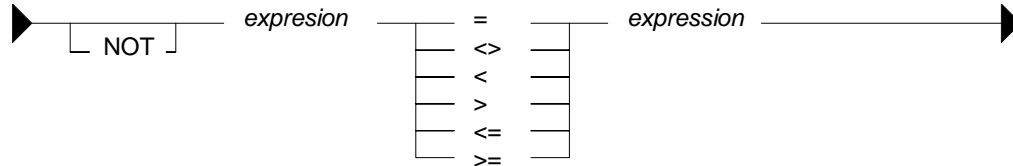


Figure 63, Basic Predicate syntax, 1 of 2

```

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

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

Figure 64, Basic Predicate examples

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

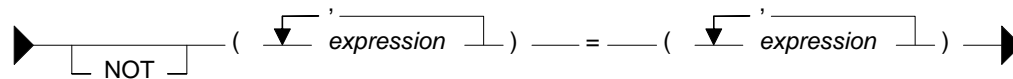


Figure 65, Basic Predicate syntax, 2 of 2

```

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

ANSWER		
ID	DEPT	JOB
30	38	Mgr

Figure 66, Basic Predicate example, multi-value check

Below is the same query written the old fashioned way:

```

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

ANSWER		
ID	DEPT	JOB
30	38	Mgr

Figure 67, Same query as prior, using individual predicates

Quantified Predicate

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

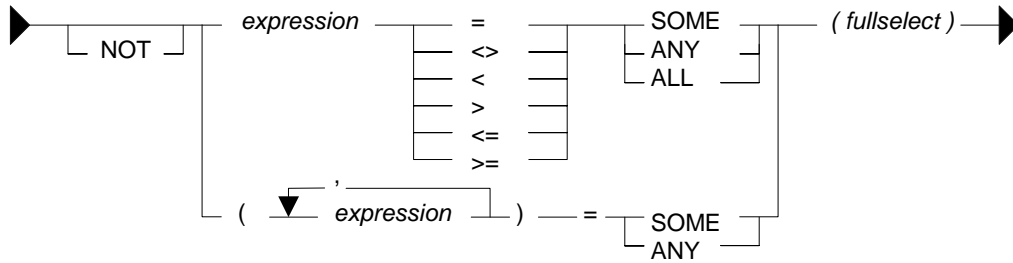


Figure 68, Quantified Predicate syntax

```

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

ANSWER
=====
ID JOB
--- ----
10 Mgr

Figure 69, Quantified Predicate example, two single-value sub-queries

```

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

ANSWER
=====
ID DEPT JOB
--- ----
20 20 Sales

Figure 70, Quantified Predicate example, multi-value sub-query

See the sub-query chapter on page 225 for more data on this predicate type.

BETWEEN Predicate

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



Figure 71, 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
FROM staff
WHERE id BETWEEN 10 AND 30
      AND id NOT BETWEEN 30 AND 10
      AND NOT id NOT BETWEEN 10 AND 30
ORDER BY id;
    
```

ANSWER
=====
ID JOB
--- ----
10 Mgr
20 Sales
30 Mgr

Figure 72, BETWEEN Predicate examples

EXISTS Predicate

An EXISTS predicate tests for the existence of matching rows.



Figure 73, EXISTS Predicate syntax

```

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

```

ANSWER	
=====	
ID	JOB

10	Mgr
20	Sales
30	Mgr
40	Sales

Figure 74, EXISTS Predicate example

NOTE: See the sub-query chapter on page 225 for more data on this predicate type.

IN Predicate

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

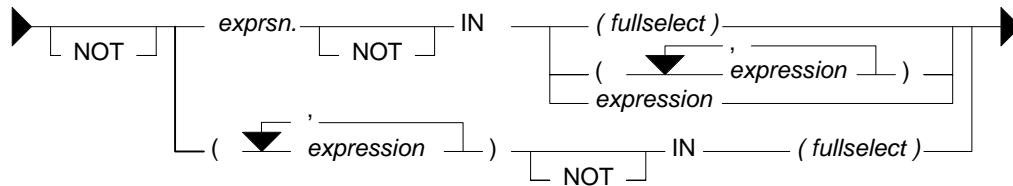


Figure 75, IN Predicate syntax

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

```

ANSWER	
=====	
ID	JOB

10	Mgr
20	Sales
30	Mgr

Figure 76, 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 225 for more details.

```

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

```

ANSWER	
=====	
EMPNO	LASTNAME

000260	JOHNSON
000270	PEREZ

Figure 77, IN Predicate example, multi-value

NOTE: See the sub-query chapter on page 225 for more data on this statement type.

LIKE Predicate

The LIKE predicate does partial checks on character strings.



Figure 78, 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 79, 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 80, 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 81, 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.

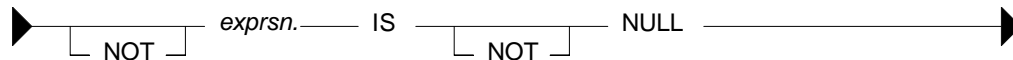


Figure 82, NULL Predicate syntax

```

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

Figure 83, NULL predicate examples

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

Special Character Usage

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

```

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

Figure 84, Refer to semi-colon in SQL text

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
              ^         ^         ^         ^         ^
              5th      2nd      3rd      1st         4th
ANSWER
=====
          423
    
```

Figure 85, 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 86, 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 87, Precedence rules, decimal example

AND/OR Precedence

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

```

SELECT *                               ANSWER>>  COL1 COL2  TABLE1
FROM   table1                          -----
WHERE  coll = 'C'                       A   AA
      AND coll >= 'A'                   B   BB
      OR  coll >= 'AA'                  C   CC
ORDER BY coll;

```

```

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

```

```

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

```

Figure 88, 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.

Processing Sequence

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

```

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

```

Figure 89, Query Processing Sequence

Observe that ON predicates (e.g. in an outer join) are always processed before any WHERE predicates (in the same join) are applied. Ignoring this processing sequence can cause what looks like an outer join to run as an inner join (see figure 578). Likewise, a function that is referenced in the SELECT section of a query (e.g. row-number) is applied after the set of matching rows has been identified, but before the data has been ordered.

CAST Expression

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

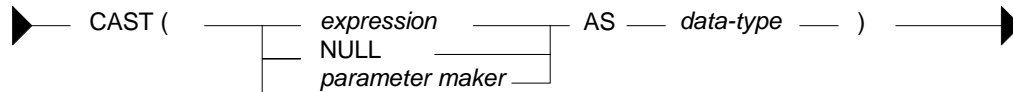


Figure 90, CAST expression syntax

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
ORDER BY   id;
            10 18357.50 18357
            20 18171.25 18171

```

Figure 91, 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
ORDER BY   id;
            10 Mgr    Mgr
            20 Sales Sal

```

Figure 92, 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
WHERE       id < 30
ORDER BY   id;
            10 -
            20 -

```

Figure 93, Use CAST expression to define SMALLINT field with null values

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

```

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

```

```

ANSWER
=====
ID EMPNO
-- -----
10 -
20 000020
30 000030
40 -
50 000050

```

Figure 94, CAST expression in join

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

```

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

```

```

ANSWER
=====
ID EMPNO
-- -----
10 -
20 000020
30 000030
40 -
50 000050

```

Figure 95, Function usage in join

VALUES Clause

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

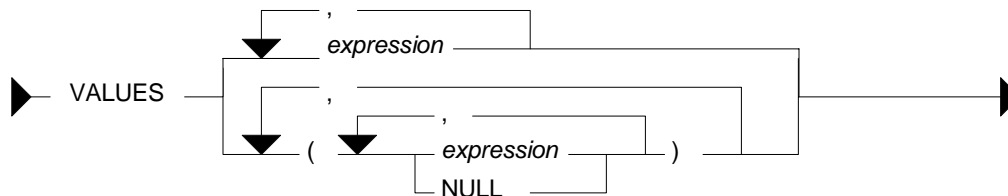


Figure 96, VALUES expression syntax

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 97, 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
(VALUES      ( 0, 'AA')
           ,( 1, 'BB')
           ,( 2, NULL)
)
SELECT *
FROM temp1;

```

ANSWER	
=====	
COL1	COL2

0	AA
1	BB
2	-

Figure 98, 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
(VALUES      (DECIMAL(0 ,3,1), 'AA')
           ,(DECIMAL(1 ,3,1), 'BB')
           ,(DECIMAL(2 ,3,1), NULL)
)
SELECT *
FROM temp1;

```

ANSWER	
=====	
COL1	COL2

0.0	AA
1.0	BB
2.0	-

Figure 99, 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
(VALUES      ( 0, CAST('AA' AS CHAR(1)))
           ,( 1, CAST('BB' AS CHAR(1)))
           ,( 2, CAST(NULL AS CHAR(1)))
)
SELECT *
FROM temp1;

```

ANSWER	
=====	
COL1	COL2

0	A
1	B
2	-

Figure 100, 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
(VALUES      ( 0, CHAR('AA',1))
           ,( 1, CHAR('BB',1))
           ,( 2, NULL)
)
SELECT *
FROM temp1;

```

ANSWER	
=====	
COL1	COL2

0	A
1	B
2	-

Figure 101, 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
(VALUES      ( 0, 'AA', 0.00)
           ,( 1, 'BB', 1.11)
           ,( 2, 'CC', 2.22)
)
,temp2 (col1b, colx) AS
(SELECT      col1
           ,col1 + col3
FROM        temp1
)
SELECT *
FROM temp2;

```

ANSWER		
=====		
COL1B	COLX	

0	0.00	
1	2.11	
2	4.22	

Figure 102, 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 103, Define a view using a VALUES clause

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

```
ANSWER
=====
COL1
-----
0
1
2
3
etc
```

Figure 104, Use VALUES defined data to seed a recursive SQL statement

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

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

```
ANSWER
=====
--- --
234 DEF
123 ABC
```

Figure 105, Generate table with unnamed columns

CASE Expression

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

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

CASE Syntax Styles

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

CASE Expression, 1st Type

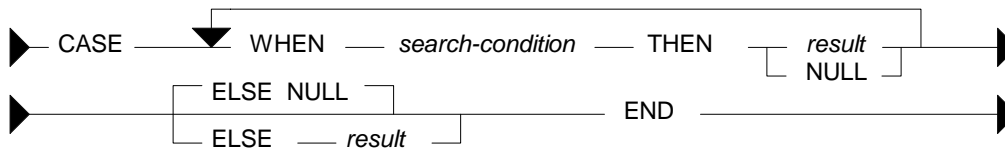


Figure 106, CASE expression syntax - 1st type

```

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

```

ANSWER		
LASTNAME	SX	SEXX
JEFFERSON	M	MALE
JOHNSON	F	FEMALE
JONES	M	MALE

Figure 107, Use CASE (1st type) to expand a value

CASE Expression, Type 2

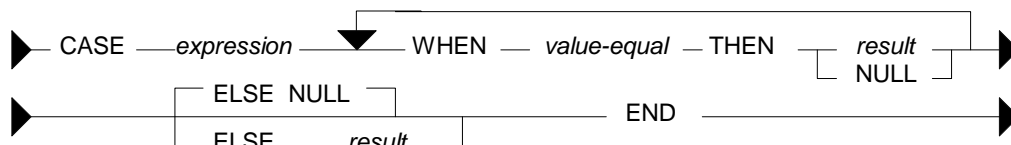


Figure 108, CASE expression syntax - 2nd type

```

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

```

ANSWER		
LASTNAME	SX	SEXX
JEFFERSON	M	MALE
JOHNSON	F	FEMALE
JONES	M	MALE

Figure 109, Use CASE (1st type) to expand a value

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.

Sample SQL

```

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

```

ANSWER			
LASTNAME	MI	SX	MX
JEFFERSON	J	M	M
JOHNSON	P	F	P
JONES	T	M	T

Figure 110, 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 111, Use CASE to get multiple counts in one pass

```

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

```

Figure 112, 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 113, UPDATE statement with nested CASE expressions

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

```

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

```

Figure 114, Use CASE to avoid divide by zero

At least one of the results in a CASE expression must be a value (i.e. not null). This is so that DB2 will know what output type to make the result.

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

```

ANSWER		
=====		
LASTNAME	SX	SXX

JEFFERSON	M	MAL
JOHNSON	F	FEM
JONES	M	MAL

Figure 115, Use CASE to derive a value (correct)

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

```

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

```

ANSWER		
=====		
LASTNAME	SX	SXX

JEFFERSON	M	FEM
JOHNSON	F	FEM
JONES	M	FEM

Figure 116, 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.

CASE in Predicate

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

```

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

```

ANSWER			
=====			
ID	DEPT	SALARY	COMM

130	42	10505.90	75.60
270	66	18555.50	-
330	66	10988.00	55.50

Figure 117, Use CASE in a predicate

The above query is arguably more complex than it seems at first glance, because unlike in an ordinary query, the CASE checks are applied in the sequence they are defined. So a row will only match "B" if it has not already matched "A".

In order to rewrite the above query using standard AND/OR predicates, we have to reproduce the CASE processing sequence. To this end, the three predicates in the next example that look for matching rows also apply any predicates that preceded them in the CASE statement:

```

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

ANSWER

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

Figure 118, Same stmt as prior, without CASE predicate

Miscellaneous SQL Statements

This section will briefly discuss several miscellaneous SQL statements. See the DB2 manuals for more details.

Cursor

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

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

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

- OPEN the cursor. This is when the contents of on any host variables referenced by the cursor (in the predicate part of the query) are transferred to DB2.
- FETCH rows from the cursor. One does as many fetches as is needed. If no row is found, the SQLCODE from the fetch will be 100.
- CLOSE the cursor.

Declare Cursor Syntax

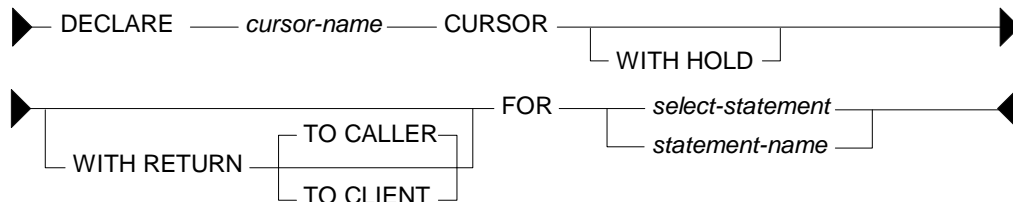


Figure 119, DECLARE CURSOR statement syntax

Syntax Notes

- The cursor-name must be unique with the application program.
- The WITH HOLD phrase indicates that the cursor will remain open if the unit of work ends with a commit. The cursor will be closed if a rollback occurs.

- The WITH RETURN phrase is used when the cursor will generate the result set returned by a stored procedure. If the cursor is open when the stored procedure ends the result set will be return either to the calling procedure, or directly to the client application.
- The FOR phrase can either refer to a select statement, the text for which will follow, or to the name of a statement has been previously prepared.

Usage Notes

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

Examples

```

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

```

Figure 120, Sample cursor

```

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

OPEN fred

DO UNTIL SQLCODE = 100

    FETCH  fred
    INTO   :name-var
          ,:salary-var

    IF salary < 1000 THEN DO

        UPDATE staff
        SET    salary = :new-salary-var
        WHERE CURRENT OF fred

    END-IF

END-DO

CLOSE fred

```

Figure 121, Use cursor in program

Select Into

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

Example

```

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

```

Figure 122, Singleton select

Prepare

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

```

▶ PREPARE — statement-name [ OUTPUT INTO — result-descriptor-name ]
[ INPUT INTO — input-descriptor-name ] FROM — host-variable ◀

```

Figure 123, PREPARE statement syntax

Syntax Notes

- The statement name names the statement. If the name is already in use, it is overridden.
- The OUTPUT descriptor will contain information about the output parameter markers. The DESCRIBE statement may be used instead of this clause.

- The INPUT descriptor will contain information about the input parameter markers.
- The FROM phrase points to the host-variable which contains the SQL statement text.

Prepared statement can be used by the following:

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

Figure 124, What statements can use prepared statement

Describe

The DESCRIBE statement is typically used in an application program to get information about a prepared statement. It can also be used in the DB2 command processor (but not in DB2BATCH) to get a description of a table, or the output columns in a SQL statement:

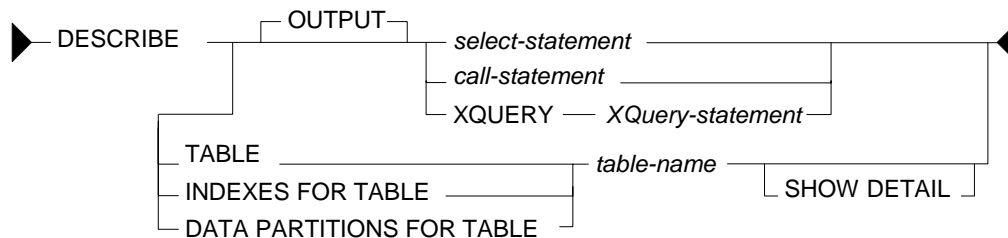


Figure 125, DESCRIBE statement syntax

Below are some examples of using the statement:

```
DESCRIBE OUTPUT SELECT * FROM staff

SQLDA Information
sqldaid : SQLDA      sqldabc: 896  sqln: 20  sqld: 7
Column Information

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

Figure 126, DESCRIBE the output columns in a select statement

```
DESCRIBE TABLE staff

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

Figure 127, DESCRIBE the columns in a table

Execute

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

Execute Immediate

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

Set Variable

The SET statement is used in an application program to set one or more program variables to values that are returned by DB2.

Examples

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

Figure 128, SET single host-variable

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

Figure 129, SET multiple host-variables

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

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

Figure 130, SET using row-fullselect

Set DB2 Control Structures

In addition to setting a host-variable, one can also set various DB2 control structures:

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

Figure 131, Other SET statements

Unit-of-Work Processing

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

Commit

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

Savepoint

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

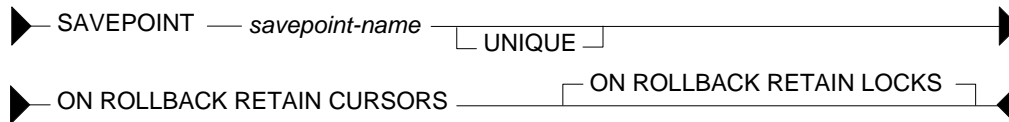


Figure 132, SAVEPOINT statement syntax

Notes

- If the savepoint name is the same as a savepoint that already exists within the same level, it overrides the prior savepoint - unless the latter was defined as being unique, in which case an error is returned.
- The RETAIN CURSORS phrase tells DB2 to, if possible, keep open any active cursors.
- The RETAIN LOCKS phrase tells DB2 to retain any locks that were obtained subsequent to the savepoint. In other words, the changes are rolled back, but the locks that came with those changes remain.

Savepoint Levels

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

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

A savepoint level ends when the process that caused its creation finishes. When a savepoint level ends, all of the savepoints created within it are released.

The following rules apply to savepoint usage:

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

Example

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

```

INSERT INTO transaction_audit_table;

SAVEPOINT before_updates ON ROLLBACK RETAIN CURSORS;

UPDATE savings_account
SET    balance = balance - 100
WHERE  cust#   = 1234;
IF SQLCODE <> 0 THEN
    ROLLBACK TO SAVEPOINT before_updates;
ELSE
    UPDATE checking_account
    SET    balance = balance + 100
    WHERE  cust#   = 1234;
    IF SQLCODE <> 0 THEN
        ROLLBACK TO SAVEPOINT before_updates;
    END
END

COMMIT;

```

Figure 133, Example of savepoint usage

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

Savepoints vs. Commits

Savepoints differ from commits in the following respects:

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

Release Savepoint

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

► RELEASE TO SAVEPOINT *savepoint-name* ◀

Figure 134, RELEASE SAVEPOINT statement syntax

Rollback

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



Figure 135, ROLLBACK statement syntax

Data Manipulation Language

The chapter has a very basic introduction to the DML (Data Manipulation Language) statements. See the DB2 manuals for more details.

Select DML Changes

A special kind of SELECT statement (see page 66) can encompass an INSERT, UPDATE, or DELETE statement to get the before or after image of whatever rows were changed (e.g. select the list of rows deleted). This kind of SELECT can be very useful when the DML statement is internally generating a value that one needs to know (e.g. an INSERT automatically creates a new invoice number using a sequence column).

Insert

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

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

Figure 136, EMP_ACT sample table - DDL

Insert Syntax

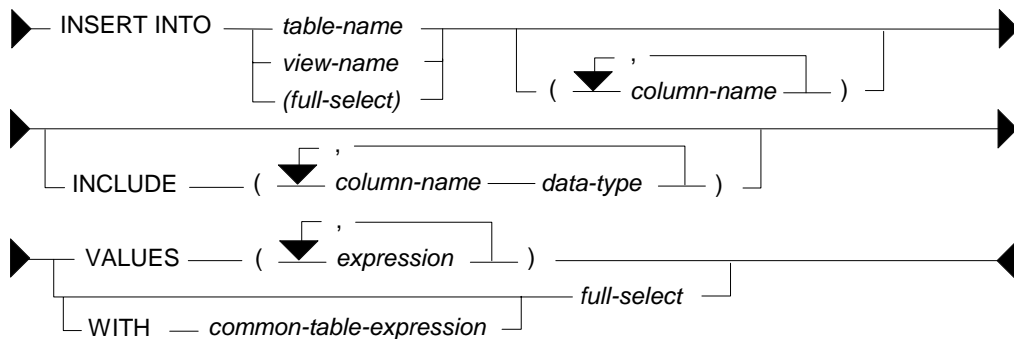


Figure 137, INSERT statement syntax

Target Objects

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

- The list of columns selected cannot include a column function (e.g. MIN).
- There must be no GROUP BY or HAVING acting on the select list.
- The list of columns selected must include all those needed to insert a new row.
- The list of columns selected cannot include one defined from a constant, expression, or a scalar function.

- Sub-queries, and other predicates, are fine, but are ignored (see figure 142).
- The query cannot be a join, nor (plain) union.
- A "union all" is permitted - as long as the underlying tables on either side of the union have check constraints such that a row being inserted is valid for one, and only one, of the tables in the union.

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

Usage Notes

- One has to provide a list of the columns (to be inserted) if the set of values provided does not equal the complete set of columns in the target table, or are not in the same order as the columns are defined in the target table.
- The columns in the INCLUDE list are not inserted. They are intended to be referenced in a SELECT statement that encompasses the INSERT (see page 66).
- The input data can either be explicitly defined using the VALUES statement, or retrieved from some other table using a fullselect.

Direct Insert

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

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

Figure 138, Single row insert

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

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

Figure 139, Multi row insert

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

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

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

Figure 140, Using null and default values

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

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

Figure 141, Explicitly listing columns being populated during insert

Insert into Full-Select

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

```

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

```

Figure 142, Insert into a fullselect

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

Insert from Select

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

```

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

```

Figure 143, Insert result of select statement

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

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

```

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

```

Figure 144, Insert result of select - specified columns only

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

```

INSERT INTO emp_act
SELECT *
FROM   emp_act;

```

Figure 145, Stupid - insert - doubles rows

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

```

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

```

Figure 146, Inserting result of union

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

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

Figure 147, Insert from common table expression

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

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

Figure 148, Insert with irrelevant sub-query

Insert into Multiple Tables

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

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

Figure 149, Customer tables - for insert usage

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

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

Figure 150, Insert into multiple tables

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

Update

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

```
UPDATE emp_act
SET   emp_time = NULL
      , emendate = DEFAULT
      , emstartdate = CURRENT DATE + 2 DAYS
      , actno = ACTNO / 2
      , projno = 'ABC'
WHERE empno = '100000';
```

Figure 151, Single row update

Update Syntax

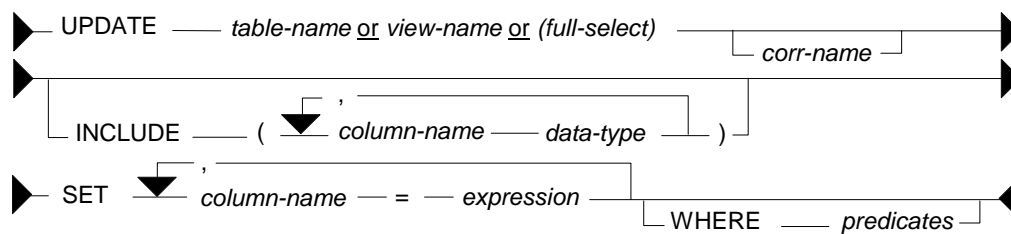


Figure 152, UPDATE statement syntax

Usage Notes

- One can update rows in a table, view, or fullselect. If the object is not a table, then it must be updateable (i.e. refer to a single table, not have any column functions, etc).
- The correlation name is optional, and is only needed if there is an expression or predicate that references another table.
- The columns in the INCLUDE list are not updated. They are intended to be referenced in a SELECT statement that encompasses the UPDATE (see page 66).
- The SET statement lists the columns to be updated, and the new values they will get.
- Predicates are optional. If none are provided, all rows in the table are updated.
- Usually, all matching rows are updated. To update some fraction of the matching rows, use a fullselect (see page: 62).

Update Examples

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

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

Figure 153, Mass update

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

```
UPDATE emp_act act1
SET   actno = actno * 2
      , emp_time = actno * 2
WHERE empno LIKE '910%';
```

Figure 154, Two columns get same value

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

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

Figure 155, Update using select

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

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

Figure 156, Multi-row update using select

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

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

Figure 157, Multi-row update using correlated select

Use Full-Select

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

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

Figure 158, Direct update of table

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

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

Figure 159, Update of fullselect

Update First "n" Rows

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

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

In the next example, the five employees with the highest salary get a big fat commission:

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

Figure 160, Update first "n" rows

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

Use OLAP Function

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

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

Figure 161, Set employee-time in row to MAX - for given employee

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

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

Figure 162, Use OLAP function to get max-time, then apply (correct)

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

The next update is similar to the prior - but it does the wrong update! In this case, the scope of the OLAP function is constrained by the predicate on PROJNO, so it no longer gets the MAX time for the employee:

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

Figure 163, Use OLAP function to get max-time, then apply (wrong)

Correlated and Uncorrelated Update

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

```

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

```

Figure 164, Update with correlated query

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

```

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

```

Figure 165, Update with uncorrelated query

Delete

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

```

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

```

Figure 166, Single-row delete

Delete Syntax

```

DELETE FROM table-name or view-name or (full-select) corr-name
          INCLUDE ( column-name data-type )
          WHERE predicates

```

Figure 167, DELETE statement syntax

Usage Notes

- One can delete rows from a table, view, or fullselect. If the object is not a table, then it must be deletable (i.e. refer to a single table, not have any column functions, etc).
- The correlation name is optional, and is only needed if there is a predicate that references another table.

- The columns in the INCLUDE list are not updated. They are intended to be referenced in a SELECT statement that encompasses the DELETE (see page 66).
- Predicates are optional. If none are provided, all rows are deleted.
- Usually, all matching rows are deleted. To delete some fraction of the matching rows, use a fullselect (see page: 65).

Basic Delete

This statement would delete all rows in the EMP_ACT table:

```
DELETE
FROM   emp_act;
```

Figure 168, Mass delete

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

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

Figure 169, Selective delete

Correlated Delete

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

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

Figure 170, Correlated delete (1 of 2)

Here is another way to write the same:

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

Figure 171, Correlated delete (2 of 2)

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

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

Figure 172, Delete using fullselect and OLAP function

Delete First "n" Rows

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

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

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

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

Figure 173, Delete first "n" rows

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

Select DML Changes

One often needs to know what data a particular insert, update, or delete statement changed. For example, one may need to get the key (e.g. invoice number) that was generated on the fly (using an identity column - see page 257) during an insert, or get the set of rows that were removed by a delete. All of this can be done by coding a special kind of select.

Select DML Syntax

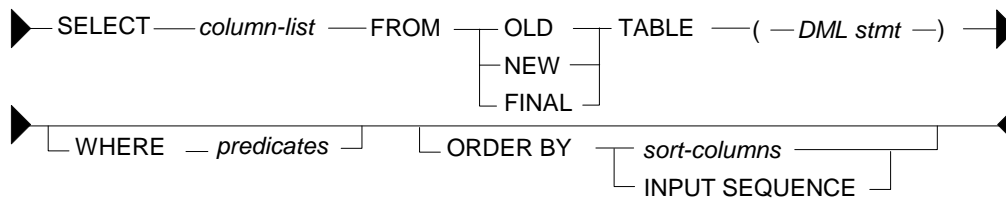


Figure 174, Select DML statement syntax

Table Types

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

Usage Notes

- Only one of the above tables can be listed in the FROM statement.
- The table listed in the FROM statement cannot be given a correlation name.
- No other table can be listed (i.e. joined to) in the FROM statement. One can reference another table in the SELECT list (see example page 69), or by using a sub-query in the predicate section of the statement.

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

Insert Examples

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

```

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

```

ANSWER		
EMPNO	PRJ	ACT
200000	ABC	10
200000	DEF	10

Figure 175, Select rows inserted

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

```

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

```

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

Figure 176, Include column to get insert sequence

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

```

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

```

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

Figure 177, Select rows in insert order

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

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

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

Now for the insert:

```

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

```

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

Figure 178, Select from an insert that has unknown values

Update Examples

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

```

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

```

ANSWER		
EMPNO	PRJ	ETIME
200000	ABC	1.00
200000	DEF	1.00

Figure 179, Select values - from before update

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

```

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

```

ANSWER		
PRJ	OLD_T	NEW_T
ABC	2.00	0.02
DEF	2.00	11.27

Figure 180, Select values - before and after update

Delete Examples

The following example lists the rows that were deleted:

```

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

```

Figure 181, List deleted rows

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

```

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

```

Figure 182, Assign row numbers to deleted rows

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

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

```

SELECT   empno                                ANSWER
        ,(SELECT  lastname                       =====
         FROM    (SELECT  empno AS e#
                  ,lastname
                  FROM    employee
                  )AS xxx
         WHERE   empno = e#)
        ,projno AS projno                       EMPNO  LASTNAME  PROJNO  ACT
        ,actno AS act                             --- ---
FROM     OLD TABLE                                000010 HAAS      AD3100  10
        (DELETE                                     000010 HAAS      MA2100  10
         FROM   emp_act                               000010 HAAS      MA2110  10
         WHERE  empno < '0001')                   000020 THOMPSON  PL2100  30
        FETCH FIRST 5 ROWS ONLY;                   000030 KWAN      IF1000  10

```

Figure 183, Join result to another table

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

Merge

The MERGE statement is a combination insert and update, or delete, statement on steroids. It can be used to take the data from a source table, and combine it with the data in a target table. The qualifying rows in the source and target tables are first matched by unique key value, and then evaluated:

- If the source row is already in the target, the latter can be either updated or deleted.
- If the source row is not in the target, it can be inserted.
- If desired, a SQL error can also be generated.

Below is the basic syntax diagram:

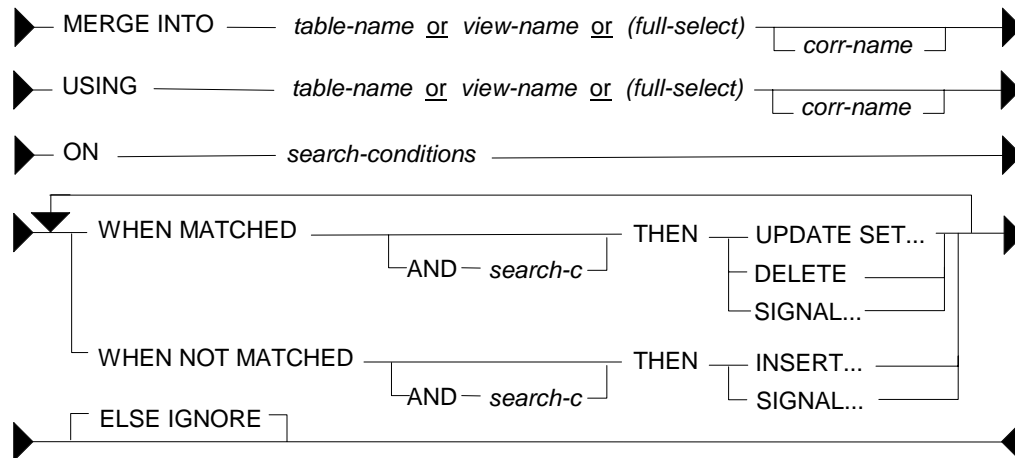


Figure 184, MERGE statement syntax

Usage Rules

The following rules apply to the merge statement:

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

- Statement-level triggers are activated, even if no rows are processed. So if a merge does either an insert, or an update, both types of statement triggers are invoked, even if all of the input is inserted.

Sample Tables

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

```

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

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

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

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

```

OLD_STAFF			NEW_STAFF	
ID	JOB	SALARY	ID	SALARY
20	Sales	18171.25	30	1750.67
30	Mgr	17506.75	40	1800.60
40	Sales	18006.00	50	2065.98

Figure 185, Sample tables for merge

Update or Insert Merge

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

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

Now for the code:

```

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

```

OLD_STAFF			NEW_STAFF	
ID	JOB	SALARY	ID	SALARY
20	Sales	18171.25	30	1750.67
30	Mgr	17506.75	40	1800.60
40	Sales	18006.00	50	2065.98

AFTER-MERGE		
ID	JOB	SALARY
20	Sales	18171.25
30	Mgr	1750.67
40	Sales	1800.60
50	?	2065.98

Figure 186, Merge - do update or insert

Delete-only Merge

The next statement deletes all matching rows:

```

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

```

AFTER-MERGE		
ID	JOB	SALARY
20	Sales	18171.25

Figure 187, Merge - delete if match

Complex Merge

The next statement has the following options:

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

Now for the code:

```

MERGE INTO old_staff oo
USING new_staff nn
ON oo.id = nn.id
WHEN MATCHED
AND oo.salary < 18000 THEN
    UPDATE
    SET oo.salary = nn.salary
WHEN MATCHED
AND oo.salary > 18000 THEN
    DELETE
WHEN NOT MATCHED
AND nn.id > 10 THEN
    INSERT
    VALUES (nn.id, '?', nn.salary)
WHEN NOT MATCHED THEN
    SIGNAL SQLSTATE '70001'
    SET MESSAGE_TEXT = 'New ID <= 10';

```

OLD_STAFF			NEW_STAFF	
ID	JOB	SALARY	ID	SALARY
20	Sales	18171.25	30	1750.67
30	Mgr	17506.75	40	1800.60
40	Sales	18006.00	50	2065.98

AFTER-MERGE		
ID	JOB	SALARY
20	Sales	18171.25
30	Mgr	1750.67
50	?	2065.98

Figure 188, Merge with multiple options

The merge statement is like the case statement (see page 44) in that the sequence in which one writes the WHEN checks determines the processing logic. In the above example, if the last check was written before the prior, any non-match would generate an error.

Using a Fullselect

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

```

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

```

AFTER-MERGE		
ID	JOB	SALARY
20	Sales	18171.25
30	Mgr	17506.75
40	Sales	18006.00
41	Sales	18171.25

Figure 189, Merge MAX row into table

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

```

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

```

Figure 190, Merge logic - done using insert

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

```

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

```

OLD_STAFF			NEW_STAFF	
ID	JOB	SALARY	ID	SALARY
20	Sales	18171.25	30	1750.67
30	Mgr	17506.75	40	1800.60
40	Sales	18006.00	50	2065.98

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

Figure 191, Merge using two fullselects

Observe that the above merge did the following:

- The target row with an ID of 30 was deleted - because it matched.
- The target row with an ID of 40 was not deleted, because it was excluded in the fullselect that was done before the merge.
- The source row with an ID of 40 was inserted, because it was not found in the target fullselect. This is why the base table now has two rows with an ID of 40.
- The source row with an ID of 50 was not inserted, because it was excluded in the fullselect that was done before the merge.

Listing Columns

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

```

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

```

AFTER-MERGE		
ID	JOB	SALARY
20	Sales	18171.25
30	?	1234.00
40	?	1234.00
50	?	5678.90

Figure 192, Listing columns and values in insert

Compound SQL

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

Introduction

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

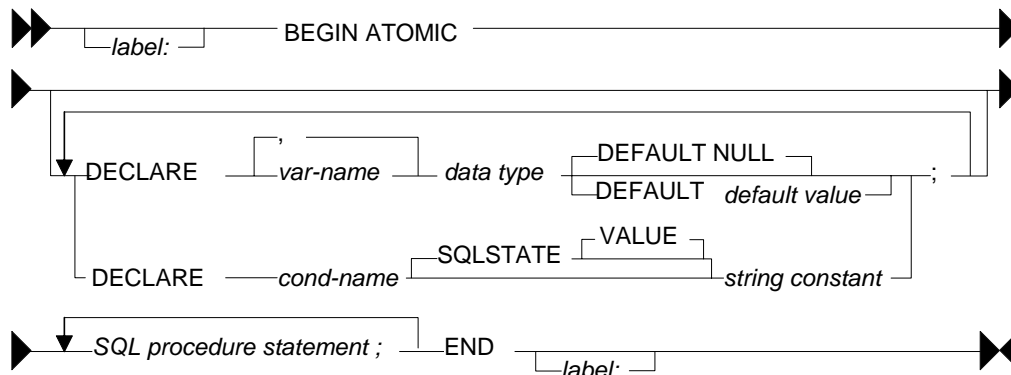


Figure 193, Compound SQL Statement syntax

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

```

BEGIN ATOMIC
  DECLARE cntr SMALLINT DEFAULT 1;
  FOR v1 AS
    SELECT id as idval
    FROM staff
    WHERE id < 80
    ORDER BY id
  DO
    UPDATE staff
    SET comm = cntr
    WHERE id = idval;
    SET cntr = cntr + 1;
  END FOR;
END

```

Figure 194, Sample Compound SQL statement

Statement Delimiter

DB2 SQL does not come with an designated statement delimiter (terminator), though a semi-colon is usually used. However, a semi-colon cannot be used in a compound SQL statement because that character is used to differentiate the sub-components of the statement.

In DB2BATCH, one can run the SET DELIMITER command (intelligent comment) to use something other than a semi-colon. The following script illustrates this usage:

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

Figure 195, Set Delimiter example

SQL Statement Usage

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

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

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

The following SQL statements can be issued:

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

DECLARE Variables

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

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

Figure 196, DECLARE examples

FOR Statement

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

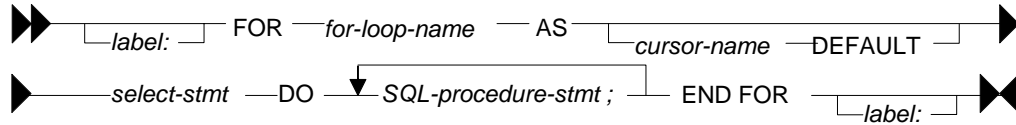


Figure 197, FOR statement syntax

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

```

BEGIN ATOMIC
  FOR v1 AS
    SELECT dept AS dname
           ,max(id) AS max_id
    FROM staff
    GROUP BY dept
    HAVING COUNT(*) > 1
    ORDER BY dept
  DO
    UPDATE staff
    SET id = id * -1
    WHERE id = max_id;
    UPDATE staff
    set dept = dept / 10
    WHERE dept = dname
    AND dept < 30;
  END FOR;
END
  
```

Figure 198, FOR statement example

GET DIAGNOSTICS Statement

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

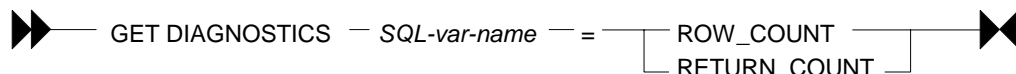


Figure 199, GET DIAGNOSTICS statement syntax

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

```

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

Figure 200, GET DIAGNOSTICS statement example

IF Statement

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

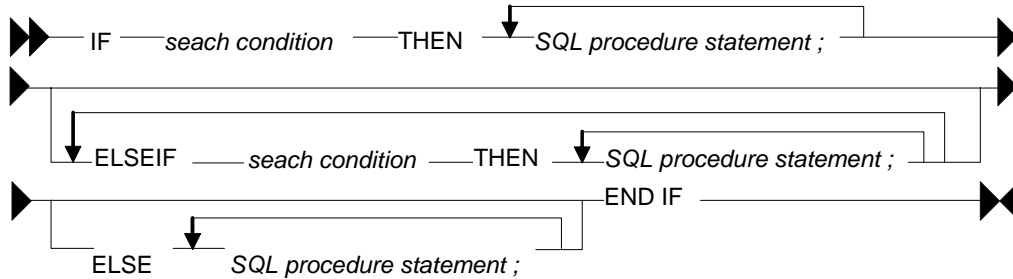


Figure 201, IF statement syntax

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

```

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

Figure 202, IF statement example

ITERATE Statement

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



Figure 203, ITERATE statement syntax

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

```

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

```

Figure 204, ITERATE statement example

LEAVE Statement

The LEAVE statement exits the labeled loop.

```

▶▶ LEAVE label ◀◀

```

Figure 205, LEAVE statement syntax

In the next example, the WHILE loop would continue forever, if left to its own devices. But after some random number of iterations, the LEAVE statement will exit the loop:

```

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

```

Figure 206, LEAVE statement example

SIGNAL Statement

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

```

▶▶ SIGNAL [SQLSTATE VALUE sqlstate string]
  [condition-name]
◀◀

▶ SET MESSAGE_TEXT = [variable-name diagnostic-string] ◀◀

```

Figure 207, SIGNAL statement syntax

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

```

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

```

Figure 208, SIGNAL statement example

WHILE Statement

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

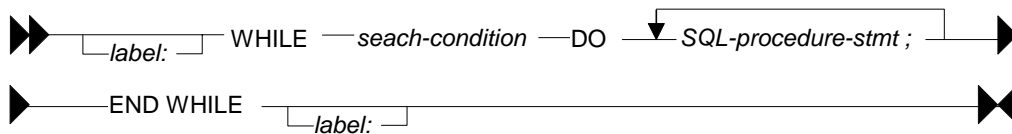


Figure 209, WHILE statement syntax

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

```

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

```

Figure 210, WHILE statement example

Other Usage

The following DB2 objects also support the language elements described above:

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

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

Test Query

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

SELECT	dept	ANSWER
	,count(*) as #rows	=====
FROM	staff	DEPT #ROWS
GROUP BY	dept	-----
ORDER BY	dept;	10 4
		15 4
		20 4
		38 5
		42 4
		51 5
		66 5
		84 4

Figure 211, List departments in STAFF table

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

Trigger

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

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

Now for the code:

```

--#SET DELIMITER !

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

CREATE TRIGGER dpt1 AFTER INSERT ON dpt
REFERENCING NEW AS NNN
FOR EACH ROW
MODE DB2SQL
BEGIN ATOMIC
  DECLARE namecnt SMALLINT DEFAULT 0;
  FOR getnames AS
    SELECT  COUNT(*) AS #n
    FROM    staff
    WHERE   dept = nnn.dept
  DO
    SET namecnt = #n;
  END FOR;
  UPDATE dpt
  SET   #names = namecnt
  WHERE dept   = nnn.dept;
END!
COMMIT!

INSERT INTO dpt (dept)
SELECT DISTINCT dept
FROM   staff!
COMMIT!

SELECT  *
FROM    dpt
ORDER BY dept!

```

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

ANSWER
=====
DEPT #NAMES

10	4
15	4
20	4
38	5
42	4
51	5
66	5
84	4

Figure 212, Trigger with compound SQL

NOTE: The above code was designed to be run in DB2BATCH. The "set delimiter" notation will probably not work in other environments.

Scalar Function

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

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

Here is the code:

```

--#SET DELIMITER !

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

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

```

IMPORTANT
=====

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

ANSWER
=====

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

Figure 213, Scalar Function with compound SQL

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

```

--#SET DELIMITER !

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

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

```

IMPORTANT
=====

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

Figure 214, Scalar Function with compound SQL

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

Table Function

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

```

--#SET DELIMITER !

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

--#SET DELIMITER ;

SELECT *
FROM   TABLE(dpt2()) T1
ORDER BY dept;

```

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

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

Figure 215, Table Function with compound SQL

Column Functions

Introduction

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

WARNING: Be very careful when using either a column function, or the DISTINCT clause, in a join. If the join is incorrectly coded, and does some form of Cartesian Product, the column function may get rid of 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 column(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.

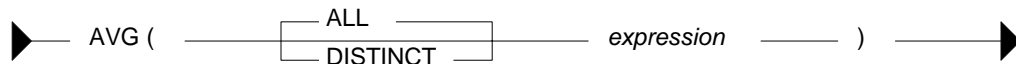


Figure 216, AVG function syntax

```

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 217, AVG function examples

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

Averaging Null and Not-Null Values

Some database designers have an intense and irrational dislike of using nullable fields. What they do instead is define all columns as not-null and then set the individual fields to zero (for numbers) or blank (for characters) when the value is unknown. This solution is reasonable in some situations, but it can cause the AVG function to give what is arguably the wrong answer.

One solution to this problem is some form of counseling or group therapy to overcome the phobia. Alternatively, one can use the CASE expression to put null values back into the answer-set being processed by the AVG function. The following SQL statement uses a modified version of the IBM sample STAFF table (all null COMM values were changed to zero) to illustrate the technique:

```

UPDATE staff
SET   comm = 0
WHERE comm IS NULL;

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

UPDATE staff
SET   comm = NULL
WHERE comm = 0;

```

ANSWER			
=====			
SALARY	COMM1	COMM2	

16675.6	351.9	513.3	

Figure 218, 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
      ,AVG(salary) AS a1
      ,COALESCE(AVG(salary),0) AS a2
      ,CASE
        WHEN AVG(salary) IS NULL THEN 0
        ELSE AVG(salary)
        END AS a3
FROM   staff
WHERE  id < 10;

```

ANSWER				
=====				
C1	A1	A2	A3	
-- -- -- --				
0	-	0	0	

Figure 219, Convert null output (from AVG) to zero

AVG Date/Time Values

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

Having said that, the following SQL gets the average birth-date of all employees:

```

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

```

ANSWER	
=====	
1	2

709113	1942-06-27

Figure 220, 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
          FROM    staff
          GROUP BY dept
          )AS xxx;
=====
<Overflow error>

```

Figure 221, 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.

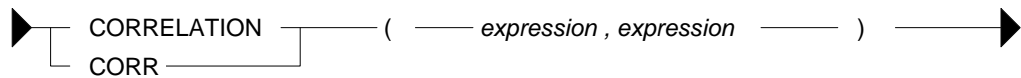


Figure 222, CORRELATION function syntax

```

WITH templ(col1, col2, col3, col4) AS
(VALUES (0 , 0 , 0 , RAND(1))
 UNION ALL
 SELECT col1 + 1
        ,col2 - 1
        ,RAND()
        ,RAND()
 FROM    templ
 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    templ;
=====
COR11  COR12  COR23  COR34
-----
1.000  -1.000  -0.017  -0.005

```

Figure 223, 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.

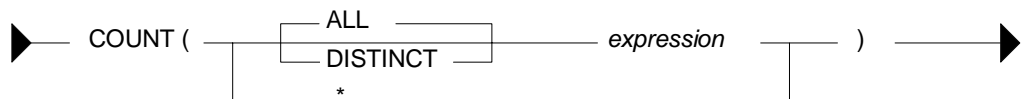


Figure 224, COUNT function syntax

```

SELECT COUNT(*) AS c1
      ,COUNT(INT(comm/10)) AS c2
      ,COUNT(ALL INT(comm/10)) AS c3
      ,COUNT(DISTINCT INT(comm/10)) AS c4
      ,COUNT(DISTINCT INT(comm)) AS c5
      ,COUNT(DISTINCT INT(comm))/10 AS c6
FROM    staff;
=====
C1 C2 C3 C4 C5 C6
-- -- -- -- --
35 24 24 19 24 2

```

Figure 225, COUNT function examples

There are 35 rows in the STAFF table (see C1 above), but only 24 of them have non-null commission values (see C2 above).

If no rows match, the COUNT returns zero - except when the SQL statement also contains a GROUP BY. In this latter case, the result is no row.

```

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

```

ANSWER	
=====	
C1	C2
----- --	
NO GP-BY	0

Figure 226, 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.

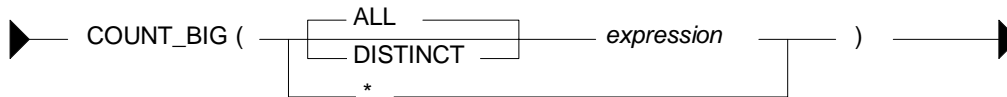


Figure 227, COUNT_BIG function syntax

```

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

```

ANSWER				
=====				
C1	C2	C3	C4	C5
--- -- -- -- --				
35.	35.	8.	7.	0.

Figure 228, COUNT_BIG function examples

COVARIANCE

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

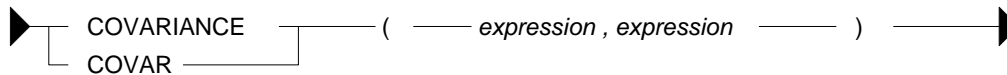


Figure 229, COVARIANCE function syntax

```

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

```

ANSWER			
=====			
COV11	COV12	COV23	COV34

83666.	-83666.	-1.4689	-0.0004

Figure 230, 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.

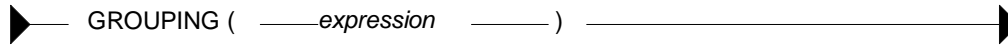


Figure 231, GROUPING function syntax

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 232, 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.

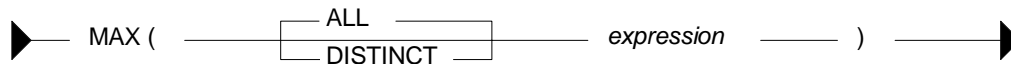


Figure 233, MAX function syntax

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 234, 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;		-----
			1980-09-30 09/30/1980 12/15/1976

Figure 235, 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
       ,MAX(CHAR(id))   AS chr
       ,MAX(DIGITS(id)) AS dig
FROM   staff;

```

ANSWER		
ID	CHR	DIG
350	90	00350

Figure 236, 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
       ,MAX(CHAR(id - 250)) AS chr
       ,MAX(DIGITS(id - 250)) AS dig
FROM   staff;

```

ANSWER		
ID	CHR	DIG
100	90	0000000240

Figure 237, 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.

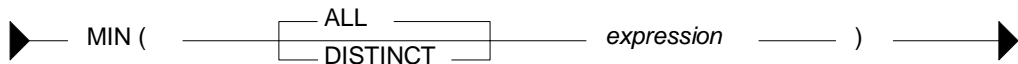


Figure 238, MIN function syntax

```

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

```

ANSWER			
1	2	3	4
10	10	10	1

Figure 239, 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 240, REGRESSION functions syntax

Functions

- REGR_AVGX returns a quantity that 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 241, 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.

```

▶ STDDEV ( [ ALL | DISTINCT ] expression ) ▶

```

Figure 242, 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 243, 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.

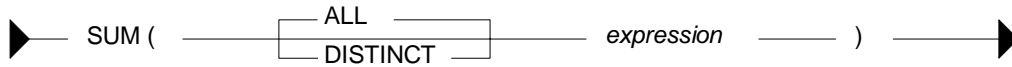


Figure 244, SUM function syntax

```

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 245, 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.

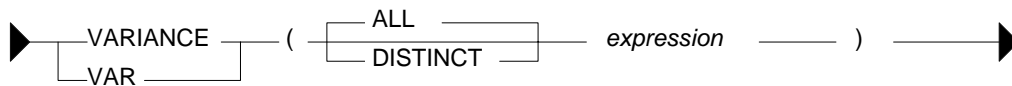


Figure 246, VARIANCE function syntax

```

SELECT  AVG(dept) AS a1           ANSWER
        ,VARIANCE(dept) AS s1     =====
        ,DEC(VARIANCE(dept),4,1) AS s2   A1 V1           V2  V3  V4
        ,DEC(VARIANCE(ALL dept),4,1) AS s3  ----
        ,DEC(VARIANCE(DISTINCT dept),4,1) AS s4  41 +5.533012244E+2 553 553 582
FROM    staff;
    
```

Figure 247, VARIANCE function examples

OLAP Functions

Introduction

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

The Bad Old Days

To really appreciate the value of the OLAP functions, one should try to do some seemingly trivial task without them. To illustrate this point, below is a simple little query:

```

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

```

ANSWER			
=====			
JOB	ID	SALARY	

Clerk	80	13504.60	
Mgr	10	18357.50	
Mgr	50	20659.80	

Figure 248, 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
WHERE   s1.name LIKE '%S%'
        AND s1.id < 90
ORDER BY s1.job
        ,s1.id;

```

ANSWER					
=====					
JOB	ID	SALARY	SUMSAL	R	

Clerk	80	13504.60	13504.60	1	
Mgr	10	18357.50	31862.10	2	
Mgr	50	20659.80	52521.90	3	

Figure 249, Using OLAP functions to get additional fields

If one does not have OLAP functions, or one is too stupid to figure out how to use them, or one gets paid by the hour, one can still get the required answer, but the code is quite tricky. The problem is that this seemingly simple query contains two nasty tricks:

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

Below are several examples that use plain SQL to get the above answer. All of the examples have the same generic design (i.e. join each matching row to itself and all previous matching rows) and share similar problems (i.e. difficult to read, and poor performance).

Nested Table Expression

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

- The TABLE phrase is required because the nested table expression has a correlated reference to the prior table. See page 281 for more details on the use of this phrase.

- There are no join predicates between the nested table expression output and the original STAFF table. They are unnecessary because these predicates are provided in the body of the nested table expression. With them there, and the above TABLE function, the nested table expression is resolved once per row obtained from the staff s1 table.
- The original literal predicates have to be repeated in the nested table expression.
- The correlated predicates in the nested table expression have to match the ORDER BY sequence (i.e. first JOB, then ID) in the final output.

Now for the query:

```

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

```

ANSWER				
JOB	ID	SALARY	SUMSAL	R
Clerk	80	13504.60	13504.60	1
Mgr	10	18357.50	31862.10	2
Mgr	50	20659.80	52521.90	3

Figure 250, 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
        ,SUM(s2.salary) AS sumsal
        ,COUNT(*) AS r
FROM    staff s1
        ,staff s2
WHERE   s1.name LIKE '%s%'
        AND s1.id < 90
        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;

```

ANSWER				
JOB	ID	SALARY	SUMSAL	R
Clerk	80	13504.60	13504.60	1
Mgr	10	18357.50	31862.10	2
Mgr	50	20659.80	52521.90	3

Figure 251, Using Self-Join and Group By to get additional fields

Nested Table Expressions in Select

In our final example, two nested table expressions 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%'
        AND s1.id < 90
ORDER BY s1.job
        ,s1.id;

```

ANSWER				
=====				
JOB	ID	SALARY	SUMSAL	R

Clerk	80	13504.60	13504.60	1
Mgr	10	18357.50	31862.10	2
Mgr	50	20659.80	52521.90	3

Figure 252, 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 253, 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.

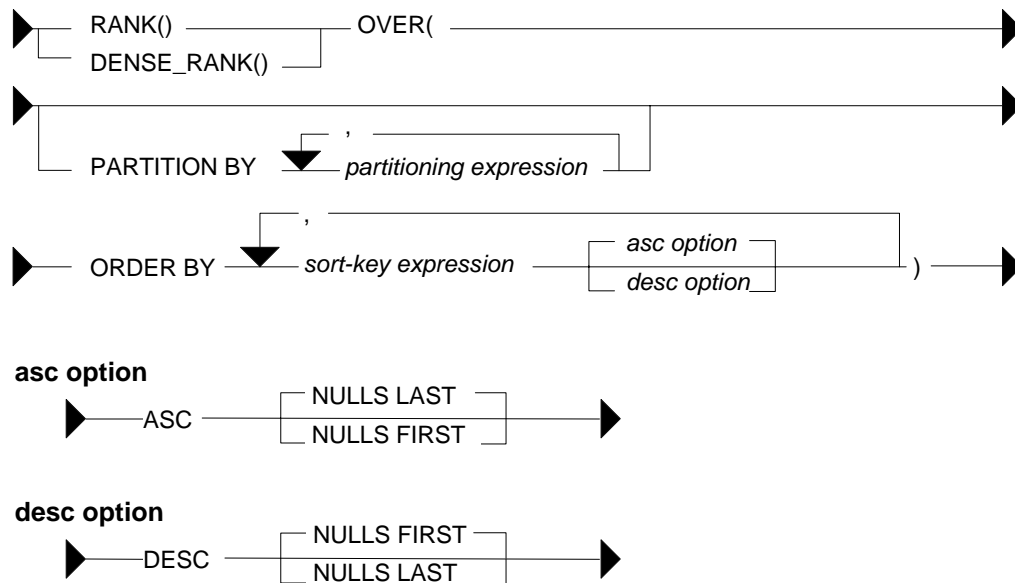


Figure 254, Ranking Functions syntax

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 97 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 preceding 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 preceding 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
ORDER BY years;

```

ANSWER

```

=====
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 255, 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 256, 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
        ,salary
        ,DENSE_RANK() OVER(ORDER BY years ASC)
        ,DENSE_RANK() OVER(ORDER BY years ASC NULLS FIRST)
        ,DENSE_RANK() OVER(ORDER BY years ASC NULLS LAST )
        ,DENSE_RANK() OVER(ORDER BY years DESC)
        ,DENSE_RANK() OVER(ORDER BY years DESC NULLS FIRST)
        ,DENSE_RANK() OVER(ORDER BY years DESC NULLS LAST )
FROM     staff
WHERE    id < 100
ORDER BY years
        ,salary;

```

ANSWER

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 257, 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
ORDER BY 1;

```

ANSWER

Y#1	Y#2
5	6

Figure 258, 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
        ,RANK() OVER(PARTITION BY years
                     ORDER BY salary) AS r1
FROM    staff
WHERE   id < 80
AND     years IS NOT NULL
ORDER BY years
        ,salary;

```

ID	YR	SALARY	R1
30	5	17506.75	1
40	6	18006.00	1
70	7	16502.83	1
10	7	18357.50	2
20	8	18171.25	1
50	0	20659.80	1

Figure 259, 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 260, 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 1)) AS dumb4
FROM    staff
WHERE   id < 40
AND     years IS NOT NULL
ORDER BY 1;

```

Figure 261, Dumb rankings, SQL

ID	YEARS	NAME	SALARY	DUMB1	DUMB2	DUMB3	DUMB4
10	7	Sanders	18357.50	1	3	1	1
20	8	Pernal	18171.25	3	2	3	1
30	5	Marenghi	17506.75	2	1	2	1

Figure 262, 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.*
          ,RANK()OVER(ORDER BY id) AS r2
FROM      (SELECT    id
          ,name
          ,RANK() OVER(ORDER BY id) AS r1
          FROM      staff
          WHERE     id    < 100
          AND      years IS NOT NULL
          )AS xxx
WHERE     id > 30
ORDER BY id;

```

ANSWER			
ID	NAME	R1	R2
40	O'Brien	4	1
50	Hanes	5	2
70	Rothman	6	3
90	Koonitz	7	4

Figure 263, 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
          ,RANK() OVER(PARTITION BY dept
          ORDER BY salary DESC) AS r1
          ,salary
          ,dept AS dp
FROM      staff
WHERE     id < 80
          AND years IS NOT NULL
ORDER BY r1 ASC
          ,salary DESC;

```

ANSWER			
ID	R1	SALARY	DP
50	1	20659.80	15
10	1	18357.50	20
40	1	18006.00	38
20	2	18171.25	20
30	2	17506.75	38
70	2	16502.83	15

Figure 264, Ordering rows by rank, using RANK function

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

```

SELECT    id
          ,(SELECT COUNT(*)
          FROM      staff s2
          WHERE     s2.id    < 80
          AND      S2.YEARS IS NOT NULL
          AND      s2.dept  = s1.dept
          AND      s2.salary >= s1.salary) AS R1
          ,SALARY
          ,dept AS dp
FROM      staff s1
WHERE     id < 80
          AND years IS NOT NULL
ORDER BY r1 ASC
          ,salary DESC;

```

ANSWER			
ID	R1	SALARY	DP
50	1	20659.80	15
10	1	18357.50	20
40	1	18006.00	38
20	2	18171.25	20
30	2	17506.75	38
70	2	16502.83	15

Figure 265, 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
            ORDER BY salary DESC) AS r1
            FROM          staff s1
            WHERE         id < 80
            AND           years IS NOT NULL
            )AS xxx
WHERE      r1 = 1
ORDER BY  dp;

```

ID	SALARY	DP
50	20659.80	15
10	18357.50	20
40	18006.00	38

Figure 266, 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
AND       years IS NOT NULL
AND       NOT EXISTS
          (SELECT *
          FROM   staff s2
          WHERE  s2.id < 80
                AND s2.years IS NOT NULL
                AND s2.dept = s1.dept
                AND s2.salary > s1.salary)
ORDER BY  DP;

```

ID	SALARY	DP
50	20659.80	15
10	18357.50	20
40	18006.00	38

Figure 267, 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
AND       years IS NOT NULL
AND       (dept, salary) IN
          (SELECT  dept, MAX(salary)
          FROM    staff
          WHERE   id < 80
                AND years IS NOT NULL
                GROUP BY dept)
ORDER BY  dp;

```

ID	SALARY	DP
50	20659.80	15
10	18357.50	20
40	18006.00	38

Figure 268, 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:

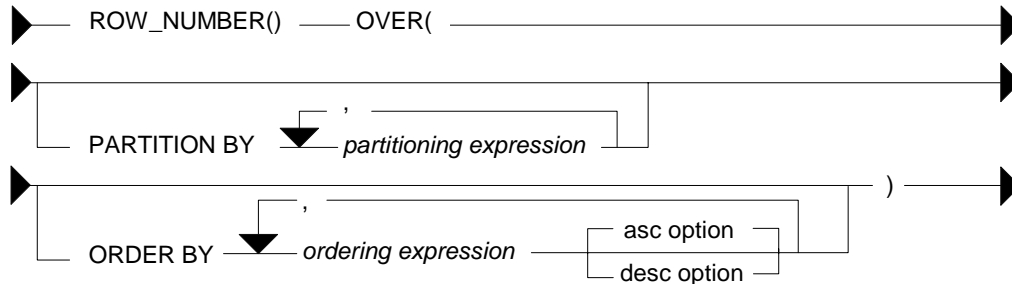


Figure 269, Numbering Function syntax

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 270, 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 1 1
ORDER BY	id;		40 O'Brien 2 2

Figure 271, 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 272, 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)
        AND job > 'L'
ORDER BY job
        ,years;

```

ANSWER						
JOB	YEARS	ID	NAME	ROW#	RN1#	RN2#
Mgr	6	140	Fraye	1	1	1
Mgr	7	10	Sanders	2	2	2
Mgr	7	100	Plotz	3	2	2
Sales	6	40	O'Brien	1	1	1
Sales	6	90	Koonitz	2	1	1
Sales	7	70	Rothman	3	3	2

Figure 273, 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  *
FROM    (SELECT  id
          ,name
          ,ROW_NUMBER() OVER(ORDER BY id) AS r
        FROM    staff
        WHERE   id < 100
          AND   years IS NOT NULL
        )AS xxx
WHERE   r <= 3
ORDER BY id;

```

ANSWER		
ID	NAME	R
10	Sanders	1
20	Pernal	2
30	Marenghi	3

Figure 274, 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 275, Select first 3 rows, using *FETCH FIRST* notation

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

```

SELECT  *                                ANSWER
FROM    (SELECT  id                               =====
        ,name                               ID NAME    R
        ,ROW_NUMBER() OVER(ORDER BY id) AS r  ---
        FROM    staff
        WHERE   id < 200
        AND     years IS NOT NULL
        )AS xxx
WHERE   r BETWEEN 3 AND 6
ORDER BY id;

```

Figure 276, 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
        WHERE   id < 200
        AND     years IS NOT NULL
        )AS xxx
WHERE   (r - 1) = ((r - 1) / 5) * 5
ORDER BY id;

```

Figure 277, Select every 5th matching row

In the next query we get the last two matching rows:

```

SELECT  *                                ANSWER
FROM    (SELECT  id                               =====
        ,name                               ID NAME    R
        ,ROW_NUMBER() OVER(ORDER BY id DESC) AS r  ---
        FROM    staff
        WHERE   id < 200
        AND     years IS NOT NULL
        )AS xxx
WHERE   r <= 2
ORDER BY id;

```

Figure 278, 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. Do all of this work in a nested table expression.
- Select from the nested table expression where the rank is three or less.

The query relies on the fact that the RANK function (see page 96) assigns the lowest common row number to each row with the same ranking:

```

SELECT      *
FROM        (SELECT  years
              ,id
              ,name
              ,RANK() OVER(ORDER BY years)      AS rnk
              ,ROW_NUMBER() OVER(ORDER BY years, id) AS row
            FROM    staff
            WHERE   id < 200
              AND  years IS NOT NULL
            )AS xxx
WHERE       rnk <= 3
ORDER BY   years
          ,id;

```

ANSWER				
=====				
YEARS	ID	NAME	RNK	ROW

3	180	Abrahams	1	1
4	170	Kermisch	2	2
5	30	Marengi	3	3
5	110	Ngan	3	4

Figure 279, 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 discuss 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 280, Performance test table - definition

The next SQL statement will insert 500,000 rows into the above table. After the rows are inserted a REORG and RUNSTATS are run, so the optimizer can 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 < 500000
 )
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 281, Performance test table - insert 500,000 rows

Imagine we want to retrieve the first five rows (only) from the above table. Below are several queries that get this result. For each query, the elapsed time, as measured by 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 500,000 matching rows, and then fetched the first five. It was not cheap:

```

SELECT s.*
FROM invoice s
ORDER BY inv#
FETCH FIRST 5 ROWS ONLY;

```

Figure 282, Fetch first 5 rows - 0.313 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. Nothing changed:

```

SELECT s.*
FROM invoice s
ORDER BY inv#
FETCH FIRST 5 ROWS ONLY
OPTIMIZE FOR 5 ROWS;

```

Figure 283, Fetch first 5 rows - 0.281 elapsed seconds

The next query is the same as the first, except that it uses the ROW_NUMBER function to sequence the output. This query is even more expensive than the first because of the cost of assigning the row numbers:

```

SELECT s.*
      ,ROW_NUMBER() OVER() AS row#
FROM invoice s
ORDER BY inv#
FETCH FIRST 5 ROWS ONLY;

```

Figure 284, Fetch first 5 rows+ number rows - 0.672 elapsed seconds

All of the above queries have processed all 500,000 matching rows, sorted them, and then fetched the first five. We can do much better if we somehow only process the five rows that we want to fetch, which is what the next query does:

```

SELECT *
FROM (SELECT s.*
      ,ROW_NUMBER() OVER() AS row#
      FROM invoice s
      )xxx
WHERE row# <= 5
ORDER BY inv#;

```

Figure 285, Process and number 5 rows only - 0.000 elapsed seconds

In the above query the "OVER()" phrase told DB2 to assign row numbers in the output order. In the next query we explicitly provide the row-number sequence, which happens to be the same as the ORDER BY sequence, but DB2 can't figure that out, so this query costs:

```
SELECT *
FROM (SELECT s.*
      ,ROW_NUMBER() OVER(ORDER BY inv#) AS row#
      FROM invoice s
      )xxx
WHERE row# <= 5
ORDER BY inv#;
```

Figure 286, Process and number 5 rows only - 0.281 elapsed seconds

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

In the following example, we start by getting the row with the MIN invoice-number. This row is then joined to the row with the next to lowest invoice-number, which is then joined to the next, and so on. After five such joins, the cycle is stopped and the result is selected:

```
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 287, Fetch first 5 rows - 0.000 elapsed seconds

The above technique is nice to know, but 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 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 500,000 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.

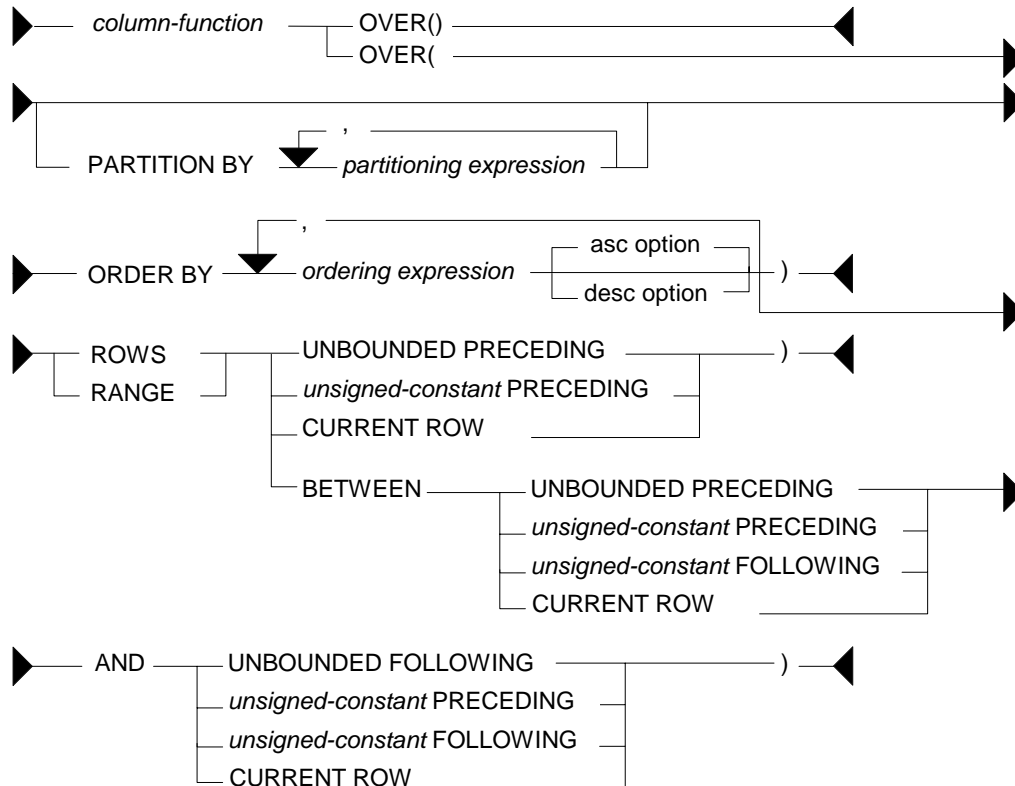


Figure 288, Aggregation Function syntax

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 109).
- 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 110).
- 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 111).
- 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 114).
- 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 289, 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 290, 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 291, 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 292, 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 293, 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 294, 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 295, 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 296, 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 preceding 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 297, 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 298, 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 299, 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
ORDER BY id;

```

ANSWER					
ID	NAME	APC	ACF	DPC	DCF
10	Sanders	10	30	30	10
20	Pernal	30	50	50	30
30	Marenghi	50	70	70	50
40	O'Brien	70	40	40	70

Figure 300, 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
=====
1 PRECEDING to CURRENT ROW      1ST-ROW    2ND-ROW    3RD-ROW    4TH-ROW
CURRENT ROW to 1 FOLLOWING      10=10     10+20=30  20+30=40  30+40=70
                                10+20=30  20+30=50  30+40=70  40 =40

DESC id (40,30,20,10)
READ ROWS, RIGHT to LEFT
=====
1 PRECEDING to CURRENT ROW      1ST-ROW    2ND-ROW    3RD-ROW    4TH-ROW
CURRENT ROW to 1 FOLLOWING      20+10=30  30+20=50  40+30=70  40 =40
                                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 301, 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 302, 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	Mareng	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 303, 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 304, 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 305, 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
        ,SUM(years) AS sum
        ,AVG(years) AS avg
        ,COUNT(*) AS row
FROM      staff
WHERE     id BETWEEN 40 AND 120
        AND years IS NOT NULL
GROUP BY dept;
    
```

ANSWER			
=====			
DEPT	SUM	AVG	ROW

15	22	7	3
38	6	6	1
42	13	6	2

Figure 306, 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 307, 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 308, 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.

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 templ (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 templ;
```

Figure 309, 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
1996-04-22	23:58:58	1996-04-22-23.58.58.123456
1996-08-15	15:15:15	1996-08-15-15.15.15.151515
0001-01-01	00:00:00	0001-01-01-00.00.00.000000

Figure 310, 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 311, 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 312, ASCII function examples

The CHR function is the inverse of the ASCII function.

ASIN

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

ATAN

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

ATANH

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

ATAN2

Returns the arctangent of x and y coordinates, specified by the first and second arguments, as an angle, expressed in radians. The output format is double.

BIGINT

Converts the input value to bigint (big integer) format. The input can be either numeric or character. If character, it must be a valid representation of a number.

```

WITH temp (big) AS
(VALUES BIGINT(1)
 UNION ALL
 SELECT big * 256
 FROM temp
 WHERE big < 1E16
 )
SELECT big
FROM temp;

```

ANSWER
=====
BIG

1
256
65536
16777216
4294967296
1099511627776
281474976710656
72057594037927936

Figure 313, 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 314, Convert FLOAT to DECIMAL and BIGINT, SQL

FLOAT1	DECIMAL1	BIGINT1
+1.234567890000000E+000		1
+1.234567890000000E+002	123.	123
+1.234567890000000E+004	12345.	12345
+1.234567890000000E+006	1234567.	1234567
+1.234567890000000E+008	123456789.	123456788
+1.234567890000000E+010	12345678900.	12345678899
+1.234567890000000E+012	1234567890000.	1234567889999
+1.234567890000000E+014	123456789000000.	123456788999999
+1.234567890000000E+016	12345678900000000.	12345678899999996
+1.234567890000000E+018	1234567890000000000.	1234567889999999488

Figure 315, Convert FLOAT to DECIMAL and BIGINT, answer

See page 416 for a discussion on floating-point number manipulation.

BLOB

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

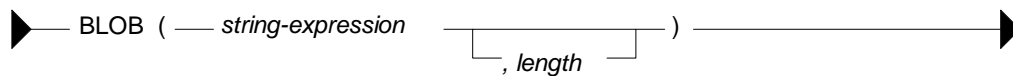


Figure 316, BLOB function syntax

CEIL or CEILING

Returns the next smallest integer value that is greater than or equal to the input (e.g. 5.045 returns 6.000). The output field type will equal the input field type.

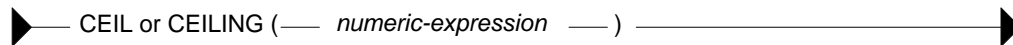


Figure 317, CEILING function syntax

SELECT d1	ANSWER (float output shortened)
,CEIL(d1) AS d2	=====
,f1	D1 D2 F1 F2
,CEIL(f1) AS f2	-----
FROM scalar;	-2.4 -2. -2.400E+0 -2.000E+0
	0.0 0. +0.000E+0 +0.000E+0
	1.8 2. +1.800E+0 +2.000E+0

Figure 318, CEIL function examples

NOTE: Usually, when DB2 converts a number from one format to another, any extra digits on the right are truncated, not rounded. For example, the output of INTEGER(123.9) is 123. Use the CEIL or ROUND functions to avoid truncation.

CHAR

The CHAR function has a multiplicity of uses. The result is always a fixed-length character value, but what happens to the input along the way depends upon the input type:

- For character input, the CHAR function acts a bit like the SUBSTR function, except that it can only truncate starting from the left-most character. The optional length parameter, if provided, must be a constant or keyword.
- Date-time input is converted into an equivalent character string. Optionally, the external format can be explicitly specified (i.e. ISO, USA, EUR, JIS, or LOCAL).
- Integer and double input is converted into a left-justified character string.
- Decimal input is converted into a right-justified character string with leading zeros. The format of the decimal point can optionally be provided. The default decimal point is a dot. The '+' and '-' symbols are not allowed as they are used as sign indicators.

Below is a syntax diagram:

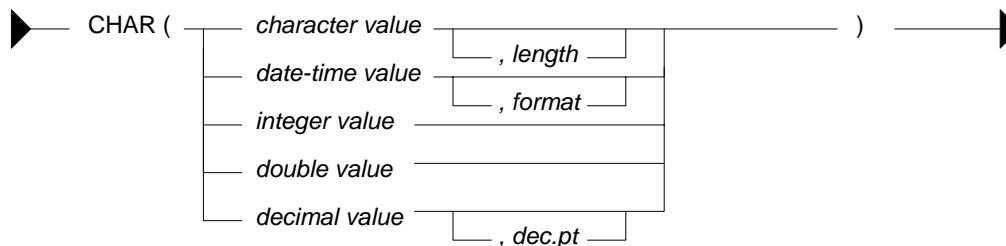


Figure 319, CHAR function syntax

Below are some examples of the CHAR function in action:

SELECT	name	ANSWER
	,CHAR(name,3)	=====
	,comm	NAME 2 COMM 4 5
	,CHAR(comm)	-----
	,CHAR(comm,'@')	James Jam 128.20 00128.20 00128@20
FROM	staff	Koonitz Koo 1386.70 01386.70 01386@70
WHERE	id BETWEEN 80	Plotz Plo - - -
	AND 100	
	ORDER BY id;	

Figure 320, CHAR function examples - characters and numbers

The CHAR function treats decimal numbers quite differently from integer and real numbers. In particular, it right-justifies the former (with leading zeros), while it left-justifies the latter (with trailing blanks). The next example illustrates this point:


```

                                ANSWER
                                =====
                                INT          CHAR_INT  CHAR_FLT    CHAR_DEC
                                -----
WITH temp1 (n) AS
(VALUE (3))
UNION ALL
SELECT n * n
FROM temp1
WHERE n < 9000
)
SELECT n          AS int
      ,CHAR(INT(n)) AS char_int
      ,CHAR(FLOAT(n)) AS charflt
      ,CHAR(DEC(n)) AS char_dec
FROM temp1;

```

INT	CHAR_INT	CHAR_FLT	CHAR_DEC
3	3	3.0E0	00000000003.
9	9	9.0E0	00000000009.
81	81	8.1E1	00000000081.
6561	6561	6.561E3	0000006561.
43046721	43046721	4.3046721E7	00043046721.

Figure 321, CHAR function examples - positive numbers

Negative numeric input is given a leading minus sign. This messes up the alignment of digits in the column (relative to any positive values). In the following query, a leading blank is put in front of all positive numbers in order to realign everything:

```

                                ANSWER
                                =====
                                N1          I1          I2          D1          D2
                                -----
WITH temp1 (n1, n2) AS
(VALUE (SMALLINT(+3))
 ,SMALLINT(-7))
UNION ALL
SELECT n1 * n2
      ,n2
FROM temp1
WHERE n1 < 300
)
SELECT n1
      ,CHAR(n1) AS i1
      ,CASE
        WHEN n1 < 0 THEN CHAR(n1)
        ELSE '+' CONCAT CHAR(n1)
      END AS i2
      ,CHAR(DEC(n1)) AS d1
      ,CASE
        WHEN n1 < 0 THEN CHAR(DEC(n1))
        ELSE '+' CONCAT CHAR(DEC(n1))
      END AS d2
FROM temp1;

```

N1	I1	I2	D1	D2
3	3	+3	00003.	+00003.
-21	-21	-21	-00021.	-00021.
147	147	+147	00147.	+00147.
-1029	-1029	-1029	-01029.	-01029.
7203	7203	+7203	07203.	+07203.

Figure 322, Align CHAR function output - numbers

Both the I2 and D2 fields above will have a trailing blank on all negative values - that was added during the concatenation operation. The RTRIM function can be used to remove it.

DATE-TIME Conversion

The CHAR function can be used to convert a date-time value to character. If the input is **not** a timestamp, the output layout can be controlled using the format option:

- ISO: International Standards Organization.
- USA: American.
- EUR: European, which is usually the same as ISO.
- JIS: Japanese Industrial Standard, which is usually the same as ISO.
- LOCAL: Whatever your computer is set to.

Below are some DATE examples:

```

                                ANSWER
                                =====
SELECT  CHAR(CURRENT DATE,ISO) AS iso      ==> 2005-11-30
        ,CHAR(CURRENT DATE,EUR) AS eur    ==> 30.11.2005
        ,CHAR(CURRENT DATE,JIS) AS jis    ==> 2005-11-30
        ,CHAR(CURRENT DATE,USA) AS usa    ==> 11/30/2005
FROM    sysibm.sysdummy1;

```

Figure 323, CHAR function examples - date value

Below are some TIME examples:

```

                                ANSWER
                                =====
SELECT  CHAR(CURRENT TIME,ISO) AS iso      ==> 19.42.21
        ,CHAR(CURRENT TIME,EUR) AS eur    ==> 19.42.21
        ,CHAR(CURRENT TIME,JIS) AS jis    ==> 19:42:21
        ,CHAR(CURRENT TIME,USA) AS usa    ==> 07:42 PM
FROM    sysibm.sysdummy1;

```

Figure 324, CHAR function examples - time value

A timestamp cannot be formatted to anything other than ISO output:

```

SELECT  CHAR(CURRENT TIMESTAMP)           ANSWER
FROM    sysibm.sysdummy1;                =====
                                           2005-11-30-19.42.21.873002

```

Figure 325, CHAR function example - timestamp value

WARNING: Converting a date or time value to character, and then ordering the set of matching rows can result in unexpected orders. See page 411 for details.

CHAR vs. DIGITS - A Comparison

Numeric input can be converted to character using either the DIGITS or the CHAR function, though the former does not support float. Both functions work differently, and neither gives perfect output. The CHAR function doesn't properly align up positive and negative numbers, while the DIGITS function loses both the decimal point and sign indicator:

```

SELECT  d2                                ANSWER
        ,CHAR(d2) AS cd2                  =====
        ,DIGITS(d2) AS dd2                D2  CD2  DD2
FROM    (SELECT DEC(d1,4,1) AS d2
        FROM  scalar
        )AS xxx                          -2.4 -002.4 0024
ORDER BY 1;                               0.0 000.0 0000
                                           1.8 001.8 0018

```

Figure 326, DIGITS vs. CHAR

NOTE: Neither the DIGITS nor the CHAR function do a great job of converting numbers to characters. See page 377 for some user-defined functions that can be used instead.

CHR

Converts integer input in the range 0 through 255 to the equivalent ASCII character value. An input value above 255 returns 255. The ASCII function (see above) is the inverse of the CHR function.

```

SELECT  'A'                               AS "c"
        ,ASCII('A') AS "c>n"
        ,CHR(ASCII('A')) AS "c>n>c"
        ,CHR(333) AS "nl"
FROM    staff
WHERE   id = 10;
                                ANSWER
                                =====
                                C  C>N  C>N>C  NL
                                -  ---  -----  --
                                A  65  A      Ŀ

```

Figure 327, 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:

<pre>SELECT c1 ,CLOB(c1) AS cc1 ,CLOB(c1,3) AS cc2 FROM scalar;</pre>	<pre>ANSWER ===== C1 CC1 CC2 ----- ABCDEF ABCDEF ABC ABCD ABCD ABC AB AB AB</pre>
---	---

Figure 328, CLOB function examples

NOTE: The DB2BATCH command processor dies a nasty death whenever it encounters a CLOB field in the output. If possible, convert to VARCHAR first to avoid this problem.

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.

<pre>SELECT id ,comm ,COALESCE(comm,0) FROM staff WHERE id < 30 ORDER BY id;</pre>	<pre>ANSWER ===== ID COMM 3 -- ---- - 10 - 0.00 20 612.45 612.45</pre>
--	---

Figure 329, 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:

<pre>WITH temp1(c1,c2,c3) AS (VVALUES (CAST(NULL AS SMALLINT) ,CAST(NULL AS SMALLINT) ,CAST(10 AS SMALLINT))) SELECT COALESCE(c1,c2,c3) AS cc1 ,CASE WHEN c1 IS NOT NULL THEN c1 WHEN c2 IS NOT NULL THEN c2 WHEN c3 IS NOT NULL THEN c3 END AS cc2 FROM TEMP1;</pre>	<pre>ANSWER ===== CC1 CC2 ---- --- 10 10</pre>
---	--

Figure 330, 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:

<pre>SELECT COUNT(*) AS #rows ,MIN(id) AS min_id ,COALESCE(MIN(id),-1) AS ccc_id FROM staff WHERE id < 5;</pre>	<pre>ANSWER ===== #ROWS MIN_ID CCC_ID ----- - 0 - -1</pre>
---	---

Figure 331, 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 332, 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
(VVALUES    ('A', 'YYY')                              =====
            , ('AE', 'OOO')                            COL1 COL2 COL3
            , ('AE', 'YYY')                              ---
)
SELECT      col1                                       AE   OOO AE000
            , col2                                       AE   YYY AEYYY
            , col1 CONCAT col2 AS col3                 A    YYY AYYY
FROM        temp1
ORDER BY   col3;

```

Figure 333, 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
(VVALUES    ('A', 'YYY')                              =====
            , ('AE', 'OOO')                            COL1 COL2 COL3
            , ('AE', 'YYY')                              ---
)
SELECT      col1                                       A    YYY A YYY
            , col2                                       AE   OOO AE000
            , CHAR(col1,2) CONCAT                      AE   YYY AEYYY
              CHAR(col2,3) AS col3
FROM        temp1
ORDER BY   col3;

```

Figure 334, 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.

<pre> WITH temp1(n1) AS (VALUES (0) UNION ALL SELECT n1 + 10 FROM temp1 WHERE n1 < 90) SELECT n1 ,DEC(RADIANS(n1) , 4, 3) AS ran ,DEC(COS(RADIANS(n1)) , 4, 3) AS cos ,DEC(SIN(RADIANS(n1)) , 4, 3) AS sin FROM temp1; </pre>	<pre> ANSWER ===== N1 RAN COS SIN -- --- --- --- 0 0.000 1.000 0.000 10 0.174 0.984 0.173 20 0.349 0.939 0.342 30 0.523 0.866 0.500 40 0.698 0.766 0.642 50 0.872 0.642 0.766 60 1.047 0.500 0.866 70 1.221 0.342 0.939 80 1.396 0.173 0.984 90 1.570 0.000 1.000 </pre>
--	---

Figure 335, RADIANS, COS, and SIN functions example

COSH

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

COT

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

DATE

Converts the input into a date value. The nature of the conversion process depends upon the input type and length:

- Timestamp and date input have the date part extracted.
- Char or varchar input that is a valid string representation of a date or a timestamp (e.g. "1997-12-23") is converted as is.
- Char or varchar input that is seven bytes long is assumed to be a Julian date value in the format `yyyynnn` where `yyyy` is the year and `nnn` is the number of days since the start of the year (in the range 001 to 366).
- Numeric input is assumed to have a value which represents the number of days since the date "0001-01-01" inclusive. All numeric types are supported, but the fractional part of a value is ignored (e.g. 12.55 becomes 12 which converts to "0001-01-12").

► `DATE (expression)` ►

Figure 336, DATE function syntax

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

<pre> SELECT ts1 ,DATE(ts1) AS dt1 FROM scalar; </pre>	<pre> ANSWER ===== TS1 DT1 ----- 1996-04-22-23.58.58.123456 1996-04-22 1996-08-15-15.15.15.151515 1996-08-15 0001-01-01-00.00.00.000000 0001-01-01 </pre>
--	---

Figure 337, DATE function example - timestamp input

```

WITH temp1(n1) AS
(VALUES (000001)
        ,(728000)
        ,(730120))
SELECT n1
       ,DATE(n1) AS d1
FROM   temp1;

```

ANSWER	
=====	
N1	D1

1	0001-01-01
728000	1994-03-13
730120	2000-01-01

Figure 338, 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
       ,DAY(dt1) AS day1
FROM   scalar
WHERE  DAY(dt1) > 10;

```

ANSWER	
=====	
DT1	DAY1

1996-04-22	22
1996-08-15	15

Figure 339, 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 range from -99 to +99, though only -31 to +31 actually make any sense:

```

SELECT dt1
       ,DAY(dt1) AS day1
       ,dt1 - '1996-04-30' AS dur2
       ,DAY(dt1 - '1996-04-30') AS day2
FROM   scalar
WHERE  DAY(dt1) > 10
ORDER BY dt1;

```

ANSWER			
=====			
DT1	DAY1	DUR2	DAY2

1996-04-22	22	-8.	-8
1996-08-15	15	315.	15

Figure 340, 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
       ,DAYNAME(dt1) AS dy1
       ,LENGTH(DAYNAME(dt1)) AS dy2
FROM   scalar
WHERE  DAYNAME(dt1) LIKE '%a%y'
ORDER BY dt1;

```

ANSWER		
=====		
DT1	DY1	DY2

0001-01-01	Monday	6
1996-04-22	Monday	6
1996-08-15	Thursday	8

Figure 341, 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.

<pre> SELECT dt1 ,DAYOFWEEK(dt1) AS dwk ,DAYNAME(dt1) AS dnm FROM scalar ORDER BY dwk ,dnm; </pre>	<pre> ANSWER ===== DT1 DWK DNM ----- 0001-01-01 2 Monday 1996-04-22 2 Monday 1996-08-15 5 Thursday </pre>
--	--

Figure 342, 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.

<pre> WITH temp1 (n) AS (VALUES (0) UNION ALL SELECT n+1 FROM temp1 WHERE n < 9), temp2 (dt1) AS (VALUES (DATE('1999-12-25')) , (DATE('2000-12-24'))), temp3 (dt2) AS (SELECT dt1 + n DAYS FROM temp1 ,temp2) SELECT CHAR(dt2,ISO) AS date ,SUBSTR(DAYNAME(dt2),1,3) AS day ,WEEK(dt2) AS w ,DAYOFWEEK(dt2) AS d ,WEEK_ISO(dt2) AS wi ,DAYOFWEEK_ISO(dt2) AS i FROM temp3 ORDER BY 1; </pre>	<pre> ANSWER ===== DATE DAY W D WI I ----- 1999-12-25 Sat 52 7 51 6 1999-12-26 Sun 53 1 51 7 1999-12-27 Mon 53 2 52 1 1999-12-28 Tue 53 3 52 2 1999-12-29 Wed 53 4 52 3 1999-12-30 Thu 53 5 52 4 1999-12-31 Fri 53 6 52 5 2000-01-01 Sat 1 7 52 6 2000-01-02 Sun 2 1 52 7 2000-01-03 Mon 2 2 1 1 2000-12-24 Sun 53 1 51 7 2000-12-25 Mon 53 2 52 1 2000-12-26 Tue 53 3 52 2 2000-12-27 Wed 53 4 52 3 2000-12-28 Thu 53 5 52 4 2000-12-29 Fri 53 6 52 5 2000-12-30 Sat 53 7 52 6 2000-12-31 Sun 54 1 52 7 2001-01-01 Mon 1 2 1 1 2001-01-02 Tue 1 3 1 2 </pre>
---	--

Figure 343, 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.

<pre> SELECT dt1 ,DAYOFYEAR(dt1) AS dyr FROM scalar ORDER BY dyr; </pre>	<pre> ANSWER ===== DT1 DYR ----- 0001-01-01 1 1996-04-22 113 1996-08-15 228 </pre>
---	--

Figure 344, 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
        ,DAYS(dt1) AS dy1
FROM    scalar
ORDER BY dy1
        ,dt1;

```

ANSWER	
DT1	DY1
0001-01-01	1
1996-04-22	728771
1996-08-15	728886

Figure 345, 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.

DBPARTITIONNUM

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

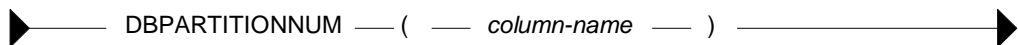


Figure 346, DBPARTITIONNUM function syntax

```

SELECT  DBPARTITIONNUM(id) AS dbnum
FROM    staff
WHERE   id = 10;

```

ANSWER	
DBNUM	
0	

Figure 347, DBPARTITIONNUM function example

The DBPARTITIONNUM function will generate a SQL error if the column/row used can not be related directly back to specific row in a real table. Therefore, one can not use this function on fields in GROUP BY statements, nor in some views. It can also cause an error when used in an outer join, and the target row failed to match in the join.

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.

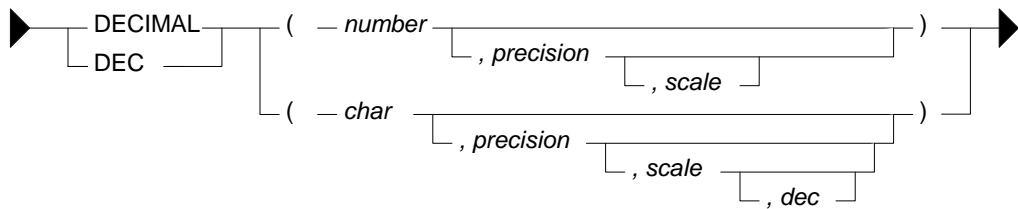


Figure 348, DECIMAL function syntax


```

WITH templ(n1,n2,c1,c2) AS
(VALUES (123
        ,1E2
        , '123.4'
        , '567$8' ))
SELECT DEC(n1,3)      AS dec1
      ,DEC(n2,4,1)   AS dec2
      ,DEC(c1,4,1)   AS dec3
      ,DEC(c2,4,1,'$') AS dec4
FROM   templ;

```

ANSWER			
DEC1	DEC2	DEC3	DEC4
123.	100.0	123.4	567.8

Figure 349, DECIMAL function examples

WARNING: Converting a floating-point number to decimal may get different results from converting the same number to integer. See page 416 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.

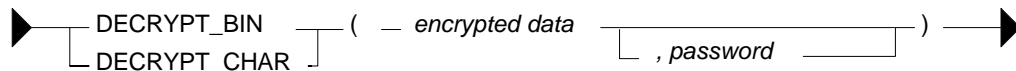


Figure 350, DECRYPT function syntax

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 351, 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      (a.name,b.name) AS df      Sanders S536 Sneider S536 4
FROM        staff a           Sanders S536 Smith S530 3
            ,staff b           Sanders S536 Lundquist L532 2
WHERE       a.id = 10         Sanders S536 Daniels D542 1
            AND b.id > 150    Sanders S536 Molinare M456 1
            AND b.id < 250    Sanders S536 Scoutten S350 1
ORDER BY   df DESC           Sanders S536 Abrahams A165 0
            ,n2 ASC;         Sanders S536 Kermisch K652 0
            ,n2 ASC;         Sanders S536 Lu L000 0

```

Figure 352, *DIFFERENCE* function example

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

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 353, *DIGITS* function examples

The CHAR function can sometimes be used as alternative to the DIGITS function. Their output differs slightly - see page 377 for a comparison.

NOTE: Neither the DIGITS nor the CHAR function do a great job of converting numbers to characters. See page 377 for some user-defined functions that can be used instead.

DLCOMMENT

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

DLINKTYPE

Returns the linktype value from a DATALINK value.

DLNEWCOPY

Returns a DATALINK value which has an attribute indicating that the referenced file has changed.

DLPREVIOUSCOPY

Returns a DATALINK value which has an attribute indicating that the previous version of the file should be restored.

DLREPLACECONTENT

Returns a DATALINK value. When the function is used in an UPDATE or INSERT the contents of the target file is replaced by another.

DLURLCOMPLETE

Returns the URL value from a DATALINK value with a link type of URL.

DLURLCOMPLETEONLY

Returns the data location attribute from a DATALINK value with a link type of URL.

DLURLCOMPLETEWRITE

Returns the complete URL value from a DATALINK value with a link type 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.

DLURLPATHWRITE

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 includes a write token if the DATALINK value comes from a DATALINK column with write permission.

DLURLSCHEME

Returns the scheme from a DATALINK value with a link type 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 templ(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 templ;	-2.00000000E+005	+2.40000000E+000

Figure 354, DOUBLE function examples

See page 416 for a discussion on floating-point number manipulation.

ENCRYPT

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

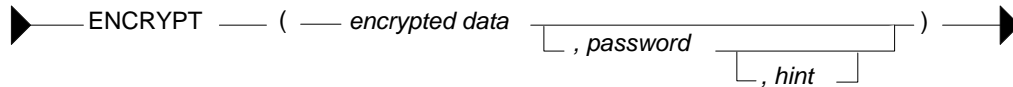


Figure 355, *DECRYPT* function syntax

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 356, *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
(VALUES (0)
 UNION ALL
 SELECT n1 + 1
 FROM temp1
 WHERE n1 < 10)
SELECT n1
,EXP(n1) AS e1
,SMALLINT(EXP(n1)) AS e2
FROM temp1;

```

ANSWER		
N1	E1	E2
0	+1.000000000000000E+0	1
1	+2.71828182845904E+0	2
2	+7.38905609893065E+0	7
3	+2.00855369231876E+1	20
4	+5.45981500331442E+1	54
5	+1.48413159102576E+2	148
6	+4.03428793492735E+2	403
7	+1.09663315842845E+3	1096
8	+2.98095798704172E+3	2980
9	+8.10308392757538E+3	8103
10	+2.20264657948067E+4	22026

Figure 357, EXP function examples

FLOAT

Same as DOUBLE.

FLOOR

Returns the next largest integer value that is smaller than or equal to the input (e.g. 5.945 returns 5.000). The output field type will equal the input field type.

```

SELECT d1
, FLOOR(d1) AS d2
, f1
, FLOOR(f1) AS f2
FROM scalar;

```

ANSWER (float output shortened)			
D1	D2	F1	F2
-2.4	-3.	-2.400E+0	-3.000E+0
0.0	+0.	+0.000E+0	+0.000E+0
1.8	+1.	+1.800E+0	+1.000E+0

Figure 358, 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 universal time, 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
10	20011017191648990521000000.	
20	20011017191648990615000000.	
30	20011017191648990642000000.	
40	20011017191648990669000000.	

NOTE: 2ND FIELD =>
IS UNPRINTABLE. =>

Figure 359, 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.

Generate Unique Timestamps

The `GENERATE_UNIQUE` output can be processed using the `TIMESTAMP` function to obtain a unique timestamp value. Adding the `CURRENT TIMEZONE` special register to the `TIMESTAMP` output will convert it to local time:

```
SELECT      CURRENT_TIMESTAMP                AS ts1
           ,TIMESTAMP(GENERATE_UNIQUE())    AS ts2
           ,TIMESTAMP(GENERATE_UNIQUE()) + CURRENT TIMEZONE AS ts3
FROM        sysibm.sysdummy1;
```

ANSWER

```
=====
TS1:  2007-01-19-18.12.33.587000
TS2:  2007-01-19-22.12.28.434960
TS3:  2007-01-19-18.12.28.434953
```

Figure 360, Covert `GENERATE_UNIQUE` output to timestamp

This code can be useful if one is doing a multi-row insert, and one wants each row inserted to have a distinct timestamp value. However, there are a few qualifications:

- The timestamp values generated will be unique in themselves. But concurrent users may also generate the same values. There is no guarantee of absolute uniqueness.
- Converting the universal-time value to local-time does not always return a value is equal to the `CURRENT_TIMESTAMP` special register. As is illustrated above, the result can differ by a few seconds. This may cause business problems if one is relying on the value to be the "true time" when something happened.

Making Random

One thing that DB2 lacks is a random number generator that makes unique values. However, if we flip the characters returned in the `GENERATE_UNIQUE` output, we have something fairly close to what is needed. Unfortunately, DB2 also lacks a `REVERSE` function, so the data flipping has to be done the hard way.

```
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
```

ANSWER

```
=====
U1          U2
-----
20000901131649119940000000  04991194613110900002
20000901131649119793000000  39791194613110900002
20000901131649119907000000  70991194613110900002
20000901131649119969000000  96991194613110900002
```

Figure 361, `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.

Using REVERSE Function

One can refer to a user-defined reverse function (see page 391 for the definition code) to flip the U1 value, and thus greatly simplify the query:

```
SELECT  u1
        ,SUBSTR(reverse(CHAR(u1)),7,20) AS u2
FROM    (SELECT HEX(GENERATE_UNIQUE()) AS u1
        FROM STAFF
        WHERE ID < 50) AS xxx
ORDER BY U2;
```

Figure 362, *GENERATE_UNIQUE* output, characters reversed using function

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
        )AS xxx
ORDER BY id;
```

ANSWER		
=====		
ID	NAME	HINT
-- -----		
10	Sanders	MY BROTHER
20	Pernal	MY BROTHER

Figure 363, *GETHINT* function example

GRAPHIC

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

HASHEDVALUE

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

```
SELECT  HASHEDVALUE(id) AS hvalue
FROM    staff
WHERE   id = 10;
```

ANSWER
=====
HVALUE

0

Figure 364, *HASHEDVALUE* function example

The `DBPARTITIONNUM` function will generate a SQL error if the column/row used can not be related directly back to specific row in a real table. Therefore, one can not use this function on fields in `GROUP BY` statements, nor in some views. It can also cause an error when used in an outer join, and the target row failed to match in the join.

HEX

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

```

WITH temp1(n1) AS
(VALUES (-3)
 UNION ALL
 SELECT n1 + 1
 FROM temp1
 WHERE n1 < 3)
SELECT SMALLINT(n1)      AS s
   ,HEX(SMALLINT(n1))   AS shx
   ,HEX(DEC(n1,4,0))   AS dhx
   ,HEX(DOUBLE(n1))    AS fhx
FROM temp1;

```

ANSWER			
S	SHX	DHX	FHX
-3	FDFE	00003D	000000000000008C0
-2	FEFF	00002D	000000000000000C0
-1	FFFF	00001D	000000000000F0BF
0	0000	00000C	0000000000000000
1	0100	00001C	000000000000F03F
2	0200	00002C	0000000000000040
3	0300	00003C	00000000000000840

Figure 365, HEX function examples, numeric data

```

SELECT c1
   ,HEX(c1) AS chx
   ,v1
   ,HEX(v1) AS vhx
FROM scalar;

```

ANSWER			
C1	CHX	V1	VHX
ABCDEF	414243444546	ABCDEF	414243444546
ABCD	414243442020	ABCD	41424344
AB	414220202020	AB	4142

Figure 366, HEX function examples, character & varchar

```

SELECT dt1
   ,HEX(dt1) AS dthx
   ,tm1
   ,HEX(tm1) AS tmhx
FROM scalar;

```

ANSWER			
DT1	DTHX	TM1	TMHX
1996-04-22	19960422	23:58:58	235858
1996-08-15	19960815	15:15:15	151515
0001-01-01	00010101	00:00:00	000000

Figure 367, 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
   ,HOUR(tm1) AS hr
FROM scalar
ORDER BY tm1;

```

ANSWER	
TM1	HR
00:00:00	0
15:15:15	15
23:58:58	23

Figure 368, 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 263 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);

WITH temp (idval) AS
(VALUES (IDENTITY_VAL_LOCAL()))
SELECT *
FROM temp;

```

ANSWER	
IDVAL	
1.	

Figure 369, IDENTITY_VAL_LOCAL function usage

INSERT

Insert one string in the middle of another, replacing a portion of what was already there. If the value to be inserted is either longer or shorter than the piece being replaced, the remainder of the data (on the right) is shifted either left or right accordingly in order to make a good fit.

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

Figure 370, 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 371, 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 indicator, 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 372, INTEGER function examples

JULIAN_DAY

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

WITH templ(dt1) AS	ANSWER		
(VALUES ('0001-01-01-00.00.00')	=====		
, ('1752-09-10-00.00.00')	DT	DY	DJ
, ('2007-06-03-00.00.00')	-----		
, ('2007-06-03-23.59.59'))	0001-01-01		1 1721426
SELECT DATE(dt1) AS dt	1752-09-10	639793	2361218
, DAYS(dt1) AS dy	2007-06-03	732830	2454255
, JULIAN_DAY(dt1) AS dj	2007-06-03	732830	2454255
FROM templ;			

Figure 373, JULIAN_DAY function example

Julian Days, A History

I happen to be a bit of an Astronomy nut, so what follows is a rather extended description of Julian Days - their purpose, and history (taken from the web).

The Julian Day calendar is used in Astronomy to relate ancient and modern astronomical observations. The Babylonians, Egyptians, Greeks (in Alexandria), and others, kept very detailed records of astronomical events, but they all used different calendars. By converting all such observations to Julian Days, we can compare and correlate them.

For example, a solar eclipse is said to have been seen at Ninevah on Julian day 1,442,454 and a lunar eclipse is said to have been observed at Babylon on Julian day number 1,566,839. These numbers correspond to the Julian Calendar dates -763-03-23 and -423-10-09 respectively). Thus the lunar eclipse occurred 124,384 days after the solar eclipse.

The Julian Day number system was invented by Joseph Justus Scaliger (born 1540-08-05 J in Agen, France, died 1609-01-21 J in Leiden, Holland) in 1583. Although the term Julian Calendar derives from the name of Julius Caesar, the term Julian day number probably does not. Evidently, this system was named, not after Julius Caesar, but after its inventor's father, Julius Caesar Scaliger (1484-1558).

The younger Scaliger combined three traditionally recognized temporal cycles of 28, 19 and 15 years to obtain a great cycle, the Scaliger cycle, or Julian period, of 7980 years (7980 is the least common multiple of 28, 19 and 15). The length of 7,980 years was chosen as the product of 28 times 19 times 15; these, respectively, are:

- The number of years when dates recur on the same days of the week.
- The lunar or Metonic cycle, after which the phases of the Moon recur on a particular day in the solar year, or year of the seasons.
- The cycle of indiction, originally a schedule of periodic taxes or government requisitions in ancient Rome.

The first Scaliger cycle began with Year 1 on -4712-01-01 (Julian) and will end after 7980 years on 3267-12-31 (Julian), which is 3268-01-22 (Gregorian). 3268-01-01 (Julian) is the first day of Year 1 of the next Scaliger cycle.

Astronomers adopted this system and adapted it to their own purposes, and they took noon GMT -4712-01-01 as their zero point. For astronomers a day begins at noon and runs until the next noon (so that the nighttime falls conveniently within one "day"). Thus they defined the Julian day number of a day as the number of days (or part of a day) elapsed since noon GMT on January 1st, 4713 B.C.E.

This was not to the liking of all scholars using the Julian day number system, in particular, historians. For chronologists who start "days" at midnight, the zero point for the Julian day number system is 00:00 at the start of -4712-01-01 J, and this is day 0. This means that 2000-01-01 G is 2,451,545 JD.

Since most days within about 150 years of the present have Julian day numbers beginning with "24", Julian day numbers within this 300-odd-year period can be abbreviated. In 1975 the convention of the modified Julian day number was adopted: Given a Julian day number JD, the modified Julian day number MJD is defined as $MJD = JD - 2,400,000.5$. This has two purposes:

- Days begin at midnight rather than noon.

- For dates in the period from 1859 to about 2130 only five digits need to be used to specify the date rather than seven.

MJD 0 thus corresponds to JD 2,400,000.5, which is twelve hours after noon on JD 2,400,000 = 1858-11-16. Thus MJD 0 designates the midnight of November 16th/17th, 1858, so day 0 in the system of modified Julian day numbers is the day 1858-11-17.

The following SQL statement uses the JULIAN_DAY function to get the Julian Date for certain days. The same calculation is also done using hand-coded SQL.

```

SELECT  bd
        ,JULIAN_DAY(bd)
        ,(1461 * (YEAR(bd) + 4800 + (MONTH(bd)-14)/12))/4
        +( 367 * (MONTH(bd)- 2 - 12*((MONTH(bd)-14)/12)))/12
        -( 3 * ((YEAR(bd) + 4900 + (MONTH(bd)-14)/12)/100))/4
        +DAY(bd) - 32075
FROM    (SELECT birthdate AS bd
        FROM    employee
        WHERE   midinit = 'R'
        ) AS xxx
ORDER BY bd;

```

ANSWER		
=====		
BD	2	3

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

Figure 374, JULIAN_DAY function examples

Julian Dates

Many computer users think of the "Julian Date" as a date format that has a layout of "yynnn" or "yyyynnn" where "yy" is the year and "nnn" is the number of days since the start of the same. A more correct use of the term "Julian Date" refers to the current date according to the calendar as originally defined by Julius Caesar - which has a leap year on every fourth year. In the US/UK, this calendar was in effect until "1752-09-14". The days between the 3rd and 13th of September in 1752 were not used in order to put everything back in sync. In the 20th and 21st centuries, to derive the Julian date one must subtract 13 days from the relevant Gregorian date (e.g.1994-01-22 becomes 1994-01-07).

The following SQL illustrates how to convert a standard DB2 Gregorian Date to an equivalent Julian Date (calendar) and a Julian Date (output format):

```

WITH templ(dt1) AS
(VALUES ('2007-01-01')
        , ('2007-01-02')
        , ('2007-12-31'))
SELECT DATE(dt1) AS dt
        ,DATE(dt1) - 13 DAYS AS dj1
        ,YEAR(dt1) * 1000 + DAYOFYEAR(dt1) AS dj2
FROM    templ;

```

ANSWER		
=====		
DT	DJ1	DJ2

2007-01-01	2006-12-19	2007001
2007-01-02	2006-12-20	2007002
2007-12-31	2007-12-18	2007365

Figure 375, 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

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

```

SELECT name
       ,LCASE(name) AS lname
       ,UCASE(name) AS uname
FROM   staff
WHERE  id < 30;

```

```

ANSWER
=====
NAME      LNAME      UNAME
-----
Sanders   sanders    SANDERS
Pernal    pernal     PERNAL

```

Figure 376, LCASE function example

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 templ(c1) AS
(VVALUES (' ABC')
        ,(' ABC ')
        ,('ABC '))
SELECT c1
       ,LEFT(c1,4)           AS c2
       ,LENGTH(LEFT(c1,4)) AS l2
FROM   templ;

```

```

ANSWER
=====
C1      C2      L2
-----
ABC     AB      4
ABC     ABC     4
ABC     ABC     4

```

Figure 377, LEFT function examples

If the input is either char or varchar, the output is varchar(4000). A column this long is a nuisance to work with. Where possible, use the SUBSTR function to get around this problem.

LENGTH

Returns an integer value with the internal length of the expression (except for double-byte string types, which return the length in characters). The value will be the same for all fields in a column, except for columns containing varying-length strings.

```

SELECT LENGTH(d1)
       ,LENGTH(f1)
       ,LENGTH(s1)
       ,LENGTH(c1)
       ,LENGTH(RTRIM(c1))
FROM   scalar;

```

```

ANSWER
=====
1      2      3      4      5
-----
2      8      2      6      6
2      8      2      6      4
2      8      2      6      2

```

Figure 378, LENGTH function examples

LN or LOG

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

```

WITH templ(n1) AS
(VVALUES (1),(123),(1234)
        ,(12345),(123456))
SELECT n1
       ,LOG(n1) AS l1
FROM   templ;

```

```

ANSWER
=====
N1      L1
-----
1       +0.0000000000000000E+000
123     +4.81218435537241E+000
1234    +7.11801620446533E+000
12345   +9.42100640177928E+000
123456  +1.17236400962654E+001

```

Figure 379, LOG function example

LOCATE

Returns an integer value with the absolute starting position of the first occurrence of the first string within the second string. If there is no match the result is zero. The optional third parameter indicates where to start the search.

▶ LOCATE (*—find-string—*, *look-in-string* , *start-pos.*) ▶

Figure 380, 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).

<pre>SELECT c1 ,LOCATE('D', c1) ,LOCATE('D', c1,2) ,LOCATE('EF',c1) ,LOCATE('A', c1,2) FROM scalar;</pre>	<pre>ANSWER ===== C1 2 3 4 5 ----- ABCDEF 4 4 5 0 ABCD 4 4 0 0 AB 0 0 0 0</pre>
---	---

Figure 381, 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.

<pre>WITH templ(n1) AS (VALUES (1),(123),(1234) ,(12345),(123456)) SELECT n1 ,LOG10(n1) AS l1 FROM templ;</pre>	<pre>ANSWER ===== N1 L1 ----- 1 +0.0000000000000000E+000 123 +2.08990511143939E+000 1234 +3.09131515969722E+000 12345 +4.09149109426795E+000 123456 +5.09151220162777E+000</pre>
--	---

Figure 382, 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.

<pre>WITH templ(c1) AS (VALUES (' ABC') ,(' ABC ') ,('ABC ')) SELECT c1 ,LTRIM(c1) AS c2 ,LENGTH(LTRIM(c1)) AS l2 FROM templ;</pre>	<pre>ANSWER ===== C1 C2 L2 ----- ABC ABC 3 ABC ABC 4 ABC ABC 5</pre>
--	---

Figure 383, 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 384, 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 385, 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 templ (ms) AS
(SELECT MIDNIGHT_SECONDS(ts1)
 FROM   scalar
)
SELECT ms
      ,SUBSTR(DIGITS(ms/3600
      ,SUBSTR(DIGITS((ms-((MS/3600)*3600))/60),9) || ':' ||
      ,SUBSTR(DIGITS(ms-((MS/60)*60),9) AS tm
FROM   templ
ORDER BY 1;

```

Figure 386, Convert MIDNIGHT_SECONDS output back to a time value

NOTE: The following two identical timestamp values: "2005-07-15.24.00.00" and "2005-07-16.00.00.00" will return different MIDNIGHT_SECONDS results. See the chapter titled "Quirks in SQL" on page 403 for a detailed discussion of this issue.

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 387, 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 388, 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;	0001-01-01 1 January
	1996-04-22 4 April
	1996-08-15 8 August

Figure 389, 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	ANSWER
(VALUES (DECIMAL(1234,10)	=====
,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 390, 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:

INPUT#1	INPUT#2	RESULT " * " OPERATOR	RESULT MULTIPLY_ALT	<--MULTIPLY_ALT--> SCALE TRUNCATD	PRECISION TRUNCATD
DEC(05,00)	DEC(05,00)	DEC(10,00)	DEC(10,00)	NO	NO
DEC(10,05)	DEC(11,03)	DEC(21,08)	DEC(21,08)	NO	NO
DEC(20,15)	DEC(21,13)	DEC(31,28)	DEC(31,18)	YES	NO
DEC(26,23)	DEC(10,01)	DEC(31,24)	DEC(31,19)	YES	NO
DEC(31,03)	DEC(15,08)	DEC(31,11)	DEC(31,03)	YES	YES

Figure 391, Decimal multiplication - same output lengths

NULLIF

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

SELECT	ANSWER
s1	=====
,NULLIF(s1,0)	S1 2 C1 4
,c1	----
,NULLIF(c1,'AB')	-2 -2 ABCDEF ABCDEF
FROM scalar	0 - ABCD ABCD
WHERE NULLIF(0,0) IS NULL;	1 1 AB -

Figure 392, 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	ANSWER
PARTITION(id) AS pp	=====
FROM staff	PP
WHERE id = 10;	--
	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	ANSWER
c1	=====
,POSSTR(c1,' ') AS p1	C1 P1 P2 P3
,POSSTR(c1,'CD') AS p2	----
,POSSTR(c1,'cd') AS p3	AB 3 0 0
FROM scalar	ABCD 5 3 0
ORDER BY 1;	ABCDEF 0 3 0

Figure 393, 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 394, 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 395, POWER function examples

QUARTER

Returns an integer value in the range 1 to 4 that represents the quarter of the year from a date or timestamp (or equivalent) value.

RADIANS

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

RAISE_ERROR

Causes the SQL statement to stop and return a user-defined error message when invoked. There are a lot of usage restrictions involving this function, see the SQL Reference for details.

► RAISE_ERROR (sqlstate , error-message) ►

Figure 396, 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 397, RAISE_ERROR function example

The SIGNAL statement (see page 79) is the statement equivalent of this function.

RAND

WARNING: Using the RAND function in a predicate can result in unpredictable results. See page 406 for a detailed description of this issue. To randomly sample the rows in a table reliably and efficiently, use the TABLESAMPLE feature. See page 372 for details.

Returns a pseudo-random floating-point value in the range of zero to one inclusive. An optional seed value can be provided to get reproducible random results. This function is especially useful when one is trying to create somewhat realistic sample data.

Usage Notes

- The RAND function returns any one of 32K distinct floating-point values in the range of zero to one inclusive. Note that many equivalent functions in other languages (e.g. SAS) return many more distinct values over the same range.
- The values generated by the RAND function are evenly distributed over the range of zero to one inclusive.
- A seed can be provided to get reproducible results. The seed can be any valid number of type integer. Note that the use of a seed alone does not give consistent results. Two different SQL statements using the same seed may return different (but internally consistent) sets of pseudo-random numbers.
- If the seed value is zero, the initial result will also be zero. All other seed values return initial values that are not the same as the seed. Subsequent calls of the RAND function in the same statement are not affected.
- If there are multiple references to the RAND function in the same SQL statement, the seed of the first RAND invocation is the one used for all.
- If the seed value is not provided, the pseudo-random numbers generated will usually be unpredictable. However, if some prior SQL statement in the same thread has already invoked the RAND function, the newly generated pseudo-random numbers "may" continue where the prior ones left off.

Typical Output Values

The following recursive SQL generates 100,000 random numbers using two as the seed value. The generated data is then summarized using various DB2 column functions:

```

WITH temp (num, ran) AS
  (VALUES (INT(1)
          ,RAND(2))
   UNION ALL
   SELECT num + 1
          ,RAND()
   FROM   temp
   WHERE  num < 100000
  )
SELECT COUNT(*)           AS #rows           ==> 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 398, Sample output from RAND function

Observe that less than 32K distinct numbers were generated. Presumably, this is because the RAND function uses a 2-byte carry. Also observe that the values range from a minimum of zero to a maximum of one.

WARNING: Unlike most, if not all, other numeric functions in DB2, the RAND function returns different results in different flavors of DB2.

Reproducible Random Numbers

The RAND function creates pseudo-random numbers. This means that the output looks random, but it is actually made using a very specific formula. If the first invocation of the function uses a seed value, all subsequent invocations will return a result that is explicitly derived from the initial seed. To illustrate this concept, the following statement selects six random numbers. Because of the use of the seed, the same six values will always be returned when this SQL statement is invoked (when invoked on my machine):

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

Figure 399, 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 400, 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
(VALUES (0
        ,SMALLINT(RAND(2)*35)*10
        ,DECIMAL(RAND()*10000,7,2))
UNION ALL
SELECT col1 + 1
        ,SMALLINT(RAND()*35)*10
        ,DECIMAL(RAND()*10000,7,2)
FROM   temp1
WHERE  col1 + 1 < 10
)
SELECT *
FROM   temp1;

```

ANSWER		
COL1	COL2	COL3
0	0	9342.32
1	250	8916.28
2	310	5430.76
3	150	5996.88
4	110	8066.34
5	50	5589.77
6	130	8602.86
7	340	184.94
8	310	5441.14
9	70	9267.55

Figure 401, 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
(VALUES (0
        ,RAND(2)
        ,RAND()+(RAND()/1E5) )
UNION ALL
SELECT col1 + 1
        ,RAND()
        ,RAND() +(RAND()/1E5)
FROM   temp1
WHERE  col1 + 1 < 30000
)
SELECT COUNT(*)           AS col#1
        ,COUNT(DISTINCT ran1) AS ran#1
        ,COUNT(DISTINCT ran2) AS ran#2
FROM   temp1;

```

ANSWER		
COL#1	RAN#1	RAN#2
30000	19698	29998

Figure 402, 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 than the RAND function. With a bit of simple data manipulation (see page 133), 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 406 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
        ,name
FROM     staff
WHERE    RAND() < 0.1
ORDER BY id;

```

ANSWER	
ID	NAME
140	Fraye
190	Sneider
290	Quill

Figure 403, 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 inner-most nested table expression) by using the RAND function to assign random values to each matching row. Subsequently, the ROW_NUMBER function is used to sequence each random value. Finally, those rows with the five lowest row numbers are selected:

```

SELECT      id                                ANSWER
            ,name                            =====
FROM        (SELECT      s2.*                ID  NAME
              ,ROW_NUMBER() OVER(ORDER BY r1) AS r2  --- -----
              FROM        (SELECT      s1.*
                            ,RAND() AS r1
                            FROM        staff s1
                            WHERE       id <= 100
                            )AS s2
              )as s3
WHERE       r2 <= 5
ORDER BY   id;

```

Figure 404, 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 405, 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 406, REAL and other numeric function examples

REPEAT

Repeats a character string "n" times.

```

▶———— REPEAT ——— ( — string-to-repeat — , #times — ) —————▶

```

Figure 407, REPEAT function syntax

```

SELECT  id
        , CHAR( REPEAT( name, 3 ), 40)
FROM    staff
WHERE   id < 40
ORDER  BY id;

```

ANSWER	
=====	
ID	

10	SandersSandersSanders
20	PernalPernalPernal
30	MarenghiMarenghiMarenghi

Figure 408, REPEAT function example

REPLACE

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

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

Figure 409, REPLACE function syntax

```

SELECT  c1
        , REPLACE( c1, 'AB', 'XY' ) AS r1
        , REPLACE( c1, 'BA', 'XY' ) AS r2
FROM    scalar;

```

ANSWER		
=====		
C1	R1	R2

ABCDEF	XYCDEF	ABCDEF
ABCD	XYCD	ABCD
AB	XY	AB

Figure 410, 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
        , REPLACE( REPLACE(
        , REPLACE( REPLACE( c1,
        'AB', 'XY' ), 'ab', 'XY' ),
        'Ab', 'XY' ), 'aB', 'XY' )
FROM    scalar;

```

ANSWER	
=====	
C1	R1

ABCDEF	XYCDEF
ABCD	XYCD
AB	XY

Figure 411, 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 templ(c1) AS
  (VALUES ( ' ABC' )
        , ( ' ABC ' )
        , ( 'ABC ' ) )
SELECT  c1
        , RIGHT( c1, 4 ) AS c2
        , LENGTH( RIGHT( c1, 4 ) ) as l2
FROM    templ;

```

ANSWER		
=====		
C1	C2	L2

ABC	ABC	4
ABC	ABC	4
ABC	BC	4

Figure 412, 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
(VALUES (123.400)
      ,( 23.450)
      ,( 3.456)
      ,( .056))
SELECT d1
      ,DEC(ROUND(d1,+2),6,3) AS p2
      ,DEC(ROUND(d1,+1),6,3) AS p1
      ,DEC(ROUND(d1,+0),6,3) AS p0
      ,DEC(ROUND(d1,-1),6,3) AS n1
      ,DEC(ROUND(d1,-2),6,3) AS n2
FROM   temp1;

```

Figure 413, ROUND function examples

RTRIM

Trims the right-most blanks of a character string.

```

SELECT c1
      ,RTRIM(c1)           AS r1
      ,LENGTH(c1)         AS r2
      ,LENGTH(RTRIM(c1)) AS r3
FROM   scalar;
                                ANSWER
                                =====
                                C1          R1          R2  R3
                                -----
                                ABCDEF    ABCDEF    6  6
                                ABCD      ABCD      6  4
                                AB        AB        6  2

```

Figure 414, 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
      ,SIGN(d1)
      ,f1
      ,SIGN(f1)
FROM   scalar;
                                ANSWER (float output shortened)
                                =====
                                D1          2          F1          4
                                -----
                                -2.4      -1.000E+0   -2.400E+0   -1.000E+0
                                0.0        +0.000E+0   +0.000E+0   +0.000E+0
                                1.8        +1.000E+0   +1.800E+0   +1.000E+0

```

Figure 415, 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 templ(n1) AS
(VALUES (0)
 UNION ALL
 SELECT n1 + 10
 FROM templ
 WHERE n1 < 80)
SELECT n1
      ,DEC(RADIANS(n1),4,3) AS ran
      ,DEC(SIN(RADIANS(n1)),4,3) AS sin
      ,DEC(TAN(RADIANS(n1)),4,3) AS tan
FROM templ;

```

ANSWER				
=====				
N1	RAN	SIN	TAN	

0	0.000	0.000	0.000	
10	0.174	0.173	0.176	
20	0.349	0.342	0.363	
30	0.523	0.500	0.577	
40	0.698	0.642	0.839	
50	0.872	0.766	1.191	
60	1.047	0.866	1.732	
70	1.221	0.939	2.747	
80	1.396	0.984	5.671	

Figure 416, SIN function example

SINH

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

SMALLINT

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

```

SELECT d1
      ,SMALLINT(d1)
      ,SMALLINT('+123')
      ,SMALLINT('-123')
      ,SMALLINT(' 123 ')
FROM scalar;

```

ANSWER					
=====					
D1	2	3	4	5	

-2.4	-2	123	-123	123	
0.0	0	123	-123	123	
1.8	1	123	-123	123	

Figure 417, SMALLINT function examples

SNAPSHOT Functions

The various SNAPSHOT functions can be used to analyze the system. They are beyond the scope of this book. Refer instead to the DB2 System Monitor Guide and Reference.

SOUNDEX

Returns a 4-character code representing the sound of the words in the argument. Use the DIFFERENCE function to convert words to soundex values and then compare.

```

SELECT a.name AS n1
      ,SOUNDEX(a.name) AS s1
      ,b.name AS n2
      ,SOUNDEX(b.name) AS s2
      ,DIFFERENCE
      (a.name,b.name) AS df
FROM staff a
     ,staff b
WHERE a.id = 10
      AND b.id > 150
      AND b.id < 250
ORDER BY df DESC
      ,n2 ASC;

```

ANSWER					
=====					
N1	S1	N2	S2	DF	

Sanders	S536	Sneider	S536	4	
Sanders	S536	Smith	S530	3	
Sanders	S536	Lundquist	L532	2	
Sanders	S536	Daniels	D542	1	
Sanders	S536	Molinare	M456	1	
Sanders	S536	Scoutten	S350	1	
Sanders	S536	Abrahams	A165	0	
Sanders	S536	Kermisch	K652	0	
Sanders	S536	Lu	L000	0	

Figure 418, SOUNDEX function example

SOUNDEX Formula

There are several minor variations on the SOUNDEX algorithm. Below is one example:

- The first letter of the name is left unchanged.

- The letters W and H are ignored.
- The vowels, A, E, I, O, U, and Y are not coded, but are used as separators (see last item).

- The remaining letters are coded as:

B, P, F, V	1
C, G, J, K, Q, S, X, Z	2
D, T	3
L	4
M, N	5
R	6

- Letters that follow letters with same code are ignored unless a separator (see the third item above) precedes them.

The result of the above calculation is a four byte value. The first byte is a character as defined in step one. The remaining three bytes are digits as defined in steps two through four. Output longer than four bytes is truncated. If the output is not long enough, it is padded on the right with zeros. The maximum number of distinct values is 8,918.

NOTE: The SOUNDEX function is something of an industry standard that was developed several decades ago. Since that time, several other similar functions have been developed. You may want to investigate writing your own DB2 function to search for similar-sounding names.

SPACE

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

WITH templ(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 templ;				3 3 X

Figure 419, SPACE function examples

SQRT

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

WITH templ(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 templ;				0.000 0.000
				1.000 1.000
				2.000 1.414

Figure 420, SQRT function example

STRIP

Removes leading, trailing, or both (the default), characters from a sting. If no strip character is provided, leading and/or trailing blank characters are removed.

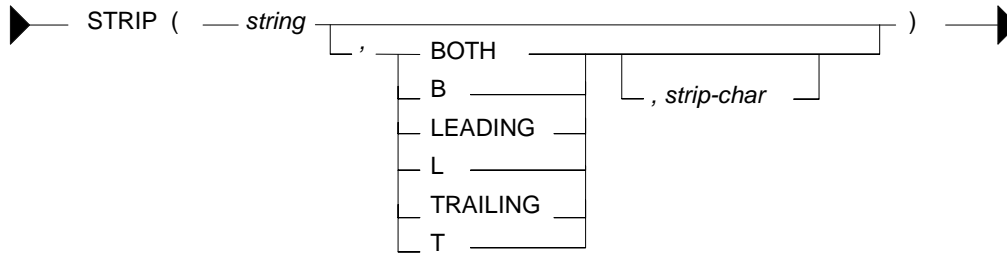


Figure 421, STRIP function syntax

Observe in the following query that the last example removes leading "A" characters:

```

WITH temp1(c1) AS
  (VALUES (' ABC')
         , (' ABC ')
         , ('ABC '))

SELECT c1
      , STRIP(c1)
      , LENGTH(STRIP(c1))
      , STRIP(c1, LEADING)
      , LENGTH(STRIP(c1, LEADING))
      , STRIP(c1, LEADING, 'A')
FROM   temp1;

```

ANSWER					
C1	C2	L2	C3	L3	C4
ABC	ABC	3	ABC	3	ABC
ABC	ABC	3	ABC	4	ABC
ABC	ABC	3	ABC	5	BC

Figure 422, STRIP function example

The TRIM function work the same way.

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.

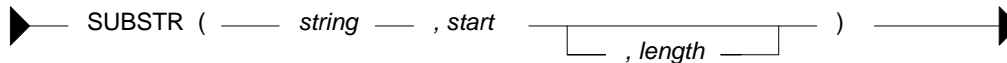


Figure 423, SUBSTR function syntax

If the length is provided, and it is longer than the field length, a SQL error results. The following statement illustrates this. Note that in this example the DAT1 field has a "field length" of 9 (i.e. the length of the longest input string).

```

WITH temp1 (len, dat1) AS
  (VALUES ( 6, '123456789')
         , ( 4, '12345' )
         , ( 16, '123' )
         )

SELECT   len
        , dat1
        , LENGTH(dat1)           AS ldat
        , SUBSTR(dat1,1,len)     AS subdat
FROM     temp1;

```

ANSWER			
LEN	DAT1	LDAT	SUBDAT
6	123456789	9	123456
4	12345	5	1234
			<error>

Figure 424, SUBSTR function - error because length parm too long

The best way to avoid the above problem is to simply write good code. If that sounds too much like hard work, try the following SQL:

```

WITH temp1 (len, dat1) AS
(VALUES ( 6, '123456789')
, ( 4, '12345' )
, ( 16, '123' )
)
SELECT len
, dat1
, LENGTH(dat1) AS ldat
, SUBSTR(dat1,1,CASE
    WHEN len < LENGTH(dat1) THEN len
    ELSE LENGTH(dat1)
    END ) AS subdat
FROM temp1;

```

```

ANSWER
=====
LEN DAT1      LDAT SUBDAT
-----
6 123456789   9 123456
4 12345       5 1234
16 123        3 123

```

Figure 425, *SUBSTR* function - avoid error using CASE (see previous)

In the above SQL a CASE statement is used to compare the LEN value against the length of the DAT1 field. If the former is larger, it is replaced by the length of the latter.

If the input is varchar, and no length value is provided, the output is varchar. However, if the length is provided, the output is of type char - with padded blanks (if needed):

```

SELECT name
, LENGTH(name) AS len
, SUBSTR(name,5) AS s1
, LENGTH(SUBSTR(name,5)) AS l1
, SUBSTR(name,5,3) AS s2
, LENGTH(SUBSTR(name,5,3)) AS l2
FROM staff
WHERE id < 60;

```

```

ANSWER
=====
NAME      LEN S1  L1 S2  L2
-----
Sanders   7  ers  3  ers  3
Pernal    6  al   2  al   3
Marenghi  8  ng   4  ng   3
O'Brien  7  ien  3  ien  3
Hanes     5  s    1  s    3

```

Figure 426, *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
, a.dept
, a.salary
, b.deptsal
FROM staff a
, TABLE
( SELECT b.dept
, SUM(b.salary) AS deptsal
FROM staff b
WHERE b.dept = a.dept
GROUP BY b.dept
) AS b
WHERE a.id < 40
ORDER BY a.id;

```

```

ANSWER
=====
ID DEPT SALARY  DEPTSAL
--
10 20 18357.50 64286.10
20 20 18171.25 64286.10
30 38 17506.75 77285.55

```

Figure 427, *Fullselect* with external table reference

See page 281 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;
CREATE ALIAS emp2 FOR emp1;

SELECT tabschema
      ,tabname
      ,card
FROM   syscat.tables
WHERE  tabname = TABLE_NAME('emp2','graeme');

```

ANSWER		
=====		
TABSHEMA	TABNAME	CARD

graeme	employee	-1

Figure 428, 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)
AS VALUES (11, 'AAA', 'BBB');

CREATE ALIAS fred2 FOR fred1;
CREATE ALIAS fred3 FOR fred2;

DROP VIEW fred1;

WITH temp1 (tab_sch, tab_nme) AS
(VALUES (TABLE_SCHEMA('fred3','graeme'),TABLE_NAME('fred3')),
        (TABLE_SCHEMA('xxxxx')
          ,TABLE_NAME('xxxxx','xxx')))
SELECT *
FROM   temp1;

```

ANSWER	
=====	
TAB_SCH	TAB_NME

graeme	fred1
graeme	xxxxx

Figure 429, TABLE_SCHEMA and TABLE_NAME functions example

TAN

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

TANH

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

TIME

Converts the input into a time value.

TIMESTAMP

Converts the input(s) into a timestamp value.

Argument Options

- If only one argument is provided, it must be (one of):
 - A timestamp value.
 - A character representation of a timestamp (the microseconds are optional).
 - A 14 byte string in the form: YYYYMMDDHHMMSS.
- If both arguments are provided:
 - The first must be a date, or a character representation of a date.
 - The second must be a time, or a character representation of a time.

```
SELECT  TIMESTAMP('1997-01-11-22.44.55.000000')
        ,TIMESTAMP('1997-01-11-22.44.55.000')
        ,TIMESTAMP('1997-01-11-22.44.55')
        ,TIMESTAMP('19970111224455')
        ,TIMESTAMP('1997-01-11','22.44.55')
FROM    staff
WHERE   id = 10;
```

*Figure 430, TIMESTAMP function examples***TIMESTAMP_FORMAT**

Takes an input string with the format: "YYYY-MM-DD HH:MM:SS" and converts it into a valid timestamp value. The `VARCHAR_FORMAT` function does the inverse.

```
WITH templ (ts1) AS
  (VALUES ('1999-12-31 23:59:59')
         , ('2002-10-30 11:22:33')
  )
SELECT  ts1
        ,TIMESTAMP_FORMAT(ts1,'YYYY-MM-DD HH24:MI:SS') AS ts2
FROM    templ
ORDER BY ts1;
```

		ANSWER
TS1	TS2	
1999-12-31 23:59:59	1999-12-31-23.59.59.000000	
2002-10-30 11:22:33	2002-10-30-11.22.33.000000	

Figure 431, TIMESTAMP_FORMAT function example

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

TIMESTAMP_ISO

Returns a timestamp in the ISO format (yyyy-mm-dd hh:mm:ss.nnnnnn) converted from the IBM internal format (yyyy-mm-dd-hh.mm.ss.nnnnnn). If the input is a date, zeros are inserted in the time part. If the input is a time, the current date is inserted in the date part and zeros in the microsecond section.

```
SELECT  tm1
        ,TIMESTAMP_ISO(tm1)
FROM    scalar;
```

		ANSWER
TM1	2	
23:58:58	2000-09-01-23.58.58.000000	
15:15:15	2000-09-01-15.15.15.000000	
00:00:00	2000-09-01-00.00.00.000000	

Figure 432, 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')),
temp2 (ts1,ts2) AS
  (SELECT  TIMESTAMP(ts1)
          ,TIMESTAMP(ts2)
    FROM    temp1),
temp3 (ts1,ts2,df) AS
  (SELECT  ts1
          ,ts2
          ,CHAR(TS1 - TS2) AS df
    FROM    temp2)
SELECT df
      ,TIMESTAMPDIFF(16,df) AS dif
      ,DAYS(ts1) - DAYS(ts2) AS dys
FROM    temp3;

```

	ANSWER
	=====
	DF DIF DYS

	00010000000001.000000 365 366
	00001130235959.000000 360 366

Figure 433, *TIMESTAMPDIFF* function example

WARNING: Some the interval types return estimates, not definitive differences, so should be used with care. For example, to get the difference between two timestamps in days, use the DAYS function as shown above. It is always correct.

Roll Your Own

The following user-defined function will get the difference, in microseconds, between two timestamp values. It can be used as an alternative to the above:

```

CREATE FUNCTION ts_diff_works(in_hi TIMESTAMP,in_lo TIMESTAMP)
RETURNS BIGINT
RETURN (BIGINT(DAYS(in_hi))          * 86400000000
      + BIGINT(MIDNIGHT_SECONDS(in_hi)) * 1000000
      + BIGINT(MICROSECOND(in_hi)))
      - (BIGINT(DAYS(in_lo))          * 86400000000
      + BIGINT(MIDNIGHT_SECONDS(in_lo)) * 1000000
      + BIGINT(MICROSECOND(in_lo)));

```

Figure 434, *Function to get difference between two timestamps*

TO_CHAR

This function is a synonym for VARCHAR_FORMAT (see page 161). It converts a timestamp value into a string using a template to define the output layout.

TO_DATE

This function is a synonym for `TIMESTAMP_FORMAT` (see page 157). It converts a character string value into a timestamp using a template to define the input layout.

TRANSLATE

Converts individual characters in either a character or graphic input string from one value to another. It can also convert lower case data to upper case.

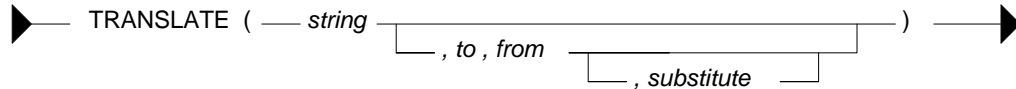


Figure 435, *TRANSLATE* function syntax

Usage Notes

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

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

Figure 436, *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 437, *REPLACE* vs. *TRANSLATE*

TRIM

See STRIP function on page 153.

TRUNC or TRUNCATE

Truncates (not rounds) the rightmost digits of an input number (1st argument). If the second argument is positive, it truncates to the right of the decimal place. If the second value is negative, it truncates to the left. A second value of zero truncates to integer. The input and output types will equal. To round instead of truncate, use the ROUND function.

```

                                ANSWER
                                =====
                                D1      POS2      POS1      ZERO      NEG1      NEG2
                                -----
WITH templ(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   templ
ORDER BY 1 DESC;

```

Figure 438, TRUNCATE function examples

TYPE_ID

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

TYPE_NAME

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

TYPE_SCHEMA

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

UCASE or UPPER

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

```

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 439, 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
        ,LENGTH(c1)           AS  l1
        ,VARCHAR(c1)         AS  v2
        ,LENGTH(VARCHAR(c1)) AS  l2
        ,VARCHAR(c1,4)       AS  v3
FROM    scalar;

```

ANSWER				
=====				
C1	L1	V2	L2	V3

ABCDEF	6	ABCDEF	6	ABCD
ABCD	6	ABCD	6	ABCD
AB	6	AB	6	AB

Figure 440, VARCHAR function examples

VARCHAR_FORMAT

Converts a timestamp value into a string with the format: "YYYY-MM-DD HH:MM:SS". The `TIMESTAMP_FORMAT` function does the inverse.

```

WITH temp1 (ts1) AS
  (VALUES (TIMESTAMP('1999-12-31-23.59.59'))
         , (TIMESTAMP('2002-10-30-11.22.33'))
  )
SELECT  ts1
        ,VARCHAR_FORMAT(ts1,'YYYY-MM-DD HH24:MI:SS') AS ts2
FROM    temp1
ORDER BY ts1;

```

ANSWER		
=====		
TS1	TS2	

1999-12-31-23.59.59.000000	1999-12-31	23:59:59
2002-10-30-11.22.33.000000	2002-10-30	11:22:33

Figure 441, VARCHAR_FORMAT function example

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

VARGRAPHIC

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

VEBLOB_CP_LARGE

This is an undocumented function that IBM has included.

VEBLOB_CP_LARGE

This is an undocumented function that IBM has included.

WEEK

Returns a value in the range 1 to 53 or 54 that represents the week of the year, where a week begins on a Sunday, or on the first day of the year. Valid input types are a date, a timestamp, or an equivalent character value. The output is of type integer.

```

SELECT  WEEK( DATE('2000-01-01')) AS w1
        ,WEEK( DATE('2000-01-02')) AS w2
        ,WEEK( DATE('2001-01-02')) AS w3
        ,WEEK( DATE('2000-12-31')) AS w4
        ,WEEK( DATE('2040-12-31')) AS w5
FROM    sysibm.sysdummy1;

```

ANSWER				
=====				
W1	W2	W3	W4	W5

1	2	1	54	53

Figure 442, 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 410).

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	=====
(VALUE (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)	1999-01-05 Tue 2 3 1 2
,SUBSTR(DAYNAME(dt2),1,3)	1999-01-06 Wed 2 4 1 3
,WEEK(dt2)	2000-12-27 Wed 53 4 52 3
,DAYOFWEEK(dt2)	2000-12-28 Thu 53 5 52 4
,WEEK_ISO(dt2)	2000-12-29 Fri 53 6 52 5
,DAYOFWEEK_ISO(dt2)	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 443, 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;	-----
	1996-04-22 1996 17
	1996-08-15 1996 33
	0001-01-01 1 1

Figure 444, YEAR and WEEK functions example

"+" PLUS

The PLUS function is same old plus sign that you have been using since you were a kid. One can use it the old fashioned way, or as if it were normal a DB2 function - with one or two input items. If there is a single input item, then the function acts as the unary "plus" operator. If there are two items, the function adds them:

```

SELECT      id
           ,salary
           ,"+"(salary)      AS s2
           ,"+"(salary,id) AS s3
FROM        staff
WHERE       id < 40
ORDER BY   id;

```

ANSWER			
ID	SALARY	S2	S3
10	18357.50	18357.50	18367.50
20	18171.25	18171.25	18191.25
30	17506.75	17506.75	17536.75

Figure 445, PLUS function examples

Both the PLUS and MINUS functions can be used to add and subtract numbers, and also date and time values. For the latter, one side of the equation has to be a date/time value, and the other either a date or time duration (a numeric representation of a date/time), or a specified date/time type. To illustrate, below are three different ways to add one year to a date:

```

SELECT      empno
           ,CHAR(birthdate,ISO)
           ,CHAR(birthdate + 1 YEAR,ISO)
           ,CHAR("+"(birthdate,DEC(00010000,8)),ISO)
           ,CHAR("+"(birthdate,DOUBLE(1),SMALLINT(1)),ISO) AS bdate4
FROM        employee
WHERE       empno < '000040'
ORDER BY   empno;

```

ANSWER				
EMPNO	BDATE1	BDATE2	BDATE3	BDATE4
000010	1933-08-24	1934-08-24	1934-08-24	1934-08-24
000020	1948-02-02	1949-02-02	1949-02-02	1949-02-02
000030	1941-05-11	1942-05-11	1942-05-11	1942-05-11

Figure 446, Adding one year to date value

"-" MINUS

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

```

SELECT      id
           ,salary
           ,"-"(salary)      AS s2
           ,"-"(salary,id) AS s3
FROM        staff
WHERE       id < 40
ORDER BY   id;

```

ANSWER			
ID	SALARY	S2	S3
10	18357.50	-18357.50	18347.50
20	18171.25	-18171.25	18151.25
30	17506.75	-17506.75	17476.75

Figure 447, MINUS function examples

"*" MULTIPLY

The MULTIPLY function is used to multiply two numeric values:

```

SELECT      id
           ,salary
           ,salary * id      AS s2
           ,""(salary,id) AS s3
FROM        staff
WHERE       id < 40
ORDER BY   id;

```

ANSWER			
ID	SALARY	S2	S3
10	18357.50	183575.00	183575.00
20	18171.25	363425.00	363425.00
30	17506.75	525202.50	525202.50

Figure 448, MULTIPLY function examples

"/" DIVIDE

The DIVIDE function is used to divide two numeric values:

```

SELECT  id
        ,salary
        ,salary / id      AS s2
        ,"/"(salary,id) AS s3
FROM    staff
WHERE   id < 40
ORDER  BY id;

```

Figure 449, DIVIDE function examples

```

ANSWER
=====
ID SALARY  S2      S3
---
10 18357.50 1835.750 1835.750
20 18171.25  908.562  908.562
30 17506.75  583.558  583.558

```

"||" CONCAT

Same as the CONCAT function:

```

SELECT  id
        ,name || 'Z'      AS n1
        ,name CONCAT 'Z' AS n2
        ,"||"(name, 'Z') AS n3
        ,CONCAT(name, 'Z') AS n4
FROM    staff
WHERE   LENGTH(name) < 5
ORDER  BY id;

```

Figure 450, CONCAT function examples

```

ANSWER
=====
ID N1      N2      N3      N4
---
110 NganZ  NganZ  NganZ  NganZ
210 LuZ   LuZ    LuZ    LuZ
270 LeaZ  LeaZ   LeaZ   LeaZ

```

User Defined Functions

Many problems that are really hard to solve using raw SQL become surprisingly easy to address, once one writes a simple function. This chapter will cover some of the basics of user-defined functions. These can be very roughly categorized by their input source, their output type, and the language used:

- External scalar functions use an external process (e.g. a C program), and possibly also an external data source, to return a single value.
- External table functions use an external process, and possibly also an external data source, to return a set of rows and columns.
- Internal sourced functions are variations of an existing DB2 function
- Internal scalar functions use compound SQL code to return a single value.
- Internal table functions use compound SQL code to return a set of rows and columns

This chapter will briefly go over the last three types of function listed above. See the official DB2 documentation for more details.

WARNING: As of the time of writing, there is a known bug in DB2 that causes the prepare cost of a dynamic SQL statement to go up exponentially when a user defined function that is written in the SQL language is referred to multiple times in a single SQL statement.

Sourced Functions

A sourced function is used to redefine an existing DB2 function so as to in some way restrict or enhance its applicability. Below is the basic syntax:

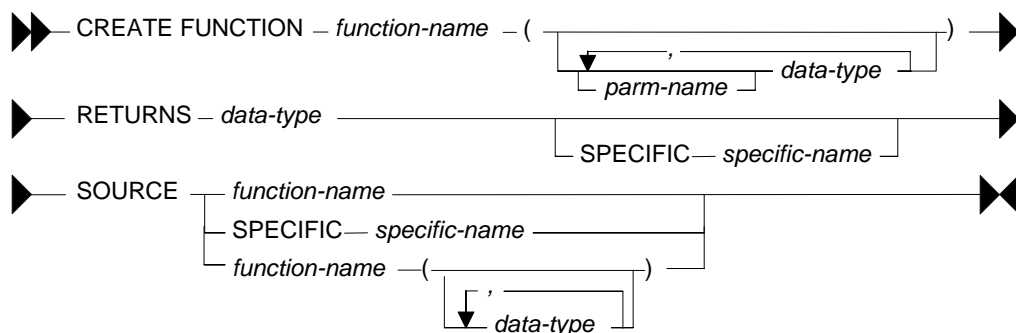


Figure 451, Sourced function syntax

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

```

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

```

Figure 452, Create sourced function

Here is an example of the function in use:

```

SELECT      id          AS ID
           ,DIGITS(id) AS I2
           ,digi_int(id) AS I3
FROM        staff
WHERE       id < 40
ORDER BY   id;

```

ANSWER		
=====		
ID	I2	I3
-- -- -- -- --		
10	00010	00010
20	00020	00020
30	00030	00030

Figure 453, Using sourced function - works

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

```

SELECT      id
           ,digi_int(INT(id))
FROM        staff
WHERE       id < 50;

```

ANSWER	
=====	
<error>	

Figure 454, Using sourced function - fails

Sourced functions are especially useful when one has created a distinct (data) type, because these do not come with any of the usual DB2 functions. To illustrate, in the following example a distinct type is created, then a table using the type, then two rows are inserted:

```

CREATE DISTINCT TYPE us_dollars AS DEC(7,2) WITH COMPARISONS;

CREATE TABLE customers
(ID SMALLINT NOT NULL
,balance us_dollars NOT NULL);

INSERT INTO customers VALUES (1 ,111.11),(2 ,222.22);

SELECT *
FROM customers
ORDER BY ID;

```

ANSWER		
=====		
ID	balance	
-- -- -- -- --		
1	111.11	
2	222.22	

Figure 455, Create distinct type and test table

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

```

SELECT      id
           ,balance * 10
FROM        customers
ORDER BY   id;

```

ANSWER	
=====	
<error>	

Figure 456, Do multiply - fails

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

```

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

```

Figure 457, Create sourced function

Now we can do the multiply:

```

SELECT      id
           ,balance * 10 AS newbal
FROM        customers
ORDER BY   id;

```

ANSWER		
=====		
ID	NEWBAL	
-- -- -- -- --		
1	1111.10	
2	2222.20	

Figure 458, Do multiply - works

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

```

SELECT  id
        , "*" (balance,10) AS newbal
FROM    customers
ORDER BY id;

```

ANSWER
=====
ID NEWBAL

1 1111.10
2 2222.20

Figure 459, Do multiply - works

Scalar Functions

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

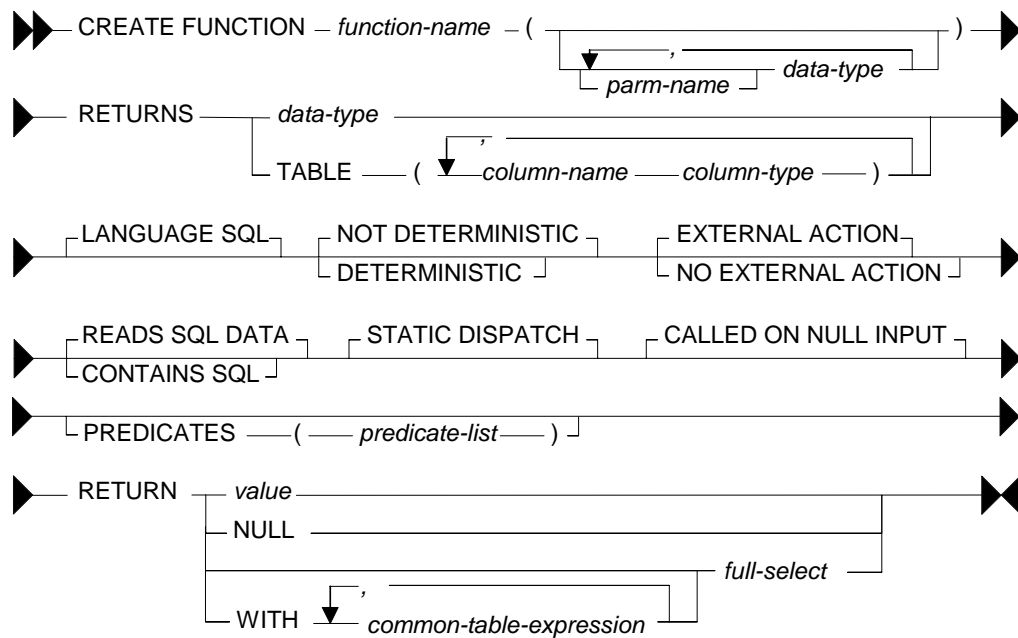


Figure 460, Scalar and Table function syntax

Description

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

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

Null Output

If a function returns a value (as opposed to a table), that value will always be nullable, regardless of whether or not the returned value can ever actually be null. This may cause problems if one is not prepared to handle a null indicator. To illustrate, the following function will return a nullable value that never be null:

```
CREATE FUNCTION Test() RETURNS CHAR(5) RETURN 'abcde';
```

Figure 461, Function returns nullable, but never null, value

Input and Output Limits

One can have multiple scalar functions with the same name and different input/output data types, but not with the same name and input/output types, but with different lengths. So if one wants to support all possible input/output lengths for, say, varchar data, one has to define the input and output lengths to be the maximum allowed for the field type.

For varchar input, one would need an output length of 32,672 bytes to support all possible input values. But this is a problem, because it is very close to the maximum allowable table (row) length in DB2, which is 32,677 bytes.

Decimal field types are even more problematic, because one needs to define both a length and a scale. To illustrate, imagine that one defines the input as being of type decimal(31,12). The following input values would be treated thus:

- A decimal(10,5) value would be fine.
- A decimal(31,31) value would lose precision.
- A decimal(31,0) value may fail because it is too large.

See page 377 for a detailed description of this problem.

Examples

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

```
CREATE FUNCTION returns_zero() RETURNS SMALLINT RETURN 0;
```

SELECT	id	AS id	ANSWER
	,returns_zero()	AS zz	=====
FROM	staff		ID ZZ
			-- --
WHERE	id = 10;		10 0

Figure 462, Simple function usage

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


```

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

SELECT   id          AS id          ANSWER
        ,calc(SMALLINT(id)) AS c1    =====
        ,calc(INTEGER (id)) AS C2    -- ---- --
FROM     staff
WHERE    id < 30
ORDER BY id;
        10 100  50
        20 200 100

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

```

Figure 463, Two functions with same name

Below is an example of a function that is not deterministic, which means that the function result can not be determined based on the input:

```

CREATE FUNCTION rnd(inval INT)
RETURNS SMALLINT
NOT DETERMINISTIC
RETURN RAND() * 50;

SELECT   id          AS id          ANSWER
        ,rnd(1) AS RND             =====
FROM     staff
WHERE    id < 40
ORDER BY id;
        10  37
        20   8
        30  42

```

Figure 464, Not deterministic function

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

```

CREATE FUNCTION get_sal(inval SMALLINT)
RETURNS DECIMAL(7,2)
RETURN SELECT salary
        FROM staff
        WHERE id = inval;

SELECT   id          AS id          ANSWER
        ,get_sal(id) AS salary      =====
FROM     staff
WHERE    id < 40
ORDER BY id;
        10 18357.50
        20 18171.25
        30 17506.75

```

Figure 465, Function using query

More complex SQL statements are also allowed - as long as the result (in a scalar function) is just one row/column value. In the next example, the either the maximum salary in the same department is obtained, or the maximum salary for the same year - whatever is higher:

```

CREATE FUNCTION max_sal(inval SMALLINT)
RETURNS DECIMAL(7,2)
RETURN WITH
    ddd (max_sal) AS
    (SELECT MAX(S2.salary)
     FROM   staff S1
           ,staff S2
     WHERE  S1.id   = inval
           AND S1.dept = s2.dept)
    ,yyy (max_sal) AS
    (SELECT MAX(S2.salary)
     FROM   staff S1
           ,staff S2
     WHERE  S1.id   = inval
           AND S1.years = s2.years)
SELECT CASE
    WHEN ddd.max_sal > yyy.max_sal
    THEN ddd.max_sal
    ELSE yyy.max_sal
END
FROM   ddd, yyy;

```

```

SELECT   id           AS id
        ,salary       AS SAL1
        ,max_sal(id) AS SAL2
FROM     staff
WHERE    id < 40
ORDER BY id;

```

```

ANSWER
=====
ID SAL1      SAL2
--  -
10 18357.50 22959.20
20 18171.25 18357.50
30 17506.75 19260.25

```

Figure 466, Function using common table expression

A scalar or table function cannot change any data, but it can be used in a DML statement. In the next example, a function is used to remove all "e" characters from the name column:

```

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

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

```

Figure 467, Function used in update

Compound SQL Usage

A function can use compound SQL, with the following limitations:

- The statement delimiter, if needed, cannot be a semi-colon.
- No DML statements are allowed.

Below is an example of a scalar function that uses compound SQL to reverse the contents of a text string:

```

--#SET DELIMITER !
CREATE FUNCTION reverse(instr VARCHAR(50))
RETURNS VARCHAR(50)
BEGIN ATOMIC
  DECLARE outstr VARCHAR(50) DEFAULT '';
  DECLARE curbyte SMALLINT DEFAULT 0;
  SET curbyte = LENGTH(RTRIM(instr));
  WHILE curbyte >= 1 DO
    SET outstr = outstr || SUBSTR(instr,curbyte,1);
    SET curbyte = curbyte - 1;
  END WHILE;
  RETURN outstr;
END!

```

IMPORTANT
=====

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

```

SELECT   id          AS id
        ,name        AS name1
        ,reverse(name) AS name2
FROM     staff
WHERE    id < 40
ORDER BY id!

```

ANSWER
=====

ID	NAME1	NAME2
10	Sanders	srednaS
20	Pernal	lanreP
30	Marenghi	ihgneraM

Figure 468, Function using compound SQL

Because compound SQL is a language with basic logical constructs, one can add code that does different things, depending on what input is provided. To illustrate, in the next example the possible output values are as follows:

- If the input is null, the output is set to null.
- If the length of the input string is less than 6, an error is flagged.
- If the length of the input string is less than 7, the result is set to -1.
- Otherwise, the result is the length of the input string.

Now for the code:

```

--#SET DELIMITER !
CREATE FUNCTION check_len(instr VARCHAR(50))
RETURNS SMALLINT
BEGIN ATOMIC
  IF instr IS NULL THEN
    RETURN NULL;
  END IF;
  IF length(instr) < 6 THEN
    SIGNAL SQLSTATE '75001'
    SET MESSAGE_TEXT = 'Input string is < 6';
  ELSEIF length(instr) < 7 THEN
    RETURN -1;
  END IF;
  RETURN length(instr);
END!

```

IMPORTANT
=====

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

```

SELECT   id          AS id
        ,name        AS name1
        ,check_len(name) AS name2
FROM     staff
WHERE    id < 60
ORDER BY id!

```

ANSWER
=====

ID	NAME1	NAME2
10	Sanders	7
20	Pernal	-1
30	Marenghi	8
40	O'Brien	7
	<error>	

Figure 469, Function with error checking logic

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

Table Functions

A table function is very similar to a scalar function, except that it returns a set of rows and columns, rather than a single value. Here is an example:

```
CREATE FUNCTION get_staff()
RETURNS TABLE (ID SMALLINT
                ,name VARCHAR(9)
                ,YR SMALLINT)
RETURN SELECT id
              ,name
              ,years
              FROM staff;

SELECT *
FROM TABLE(get_staff()) AS s
WHERE id < 40
ORDER BY id;
```

ANSWER		
=====		
ID	NAME	YR
--	-----	--
10	Sanders	7
20	Pernal	8
30	Marenghi	5

Figure 470, Simple table function

NOTE: See page 167 for the create table function syntax diagram.

Description

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

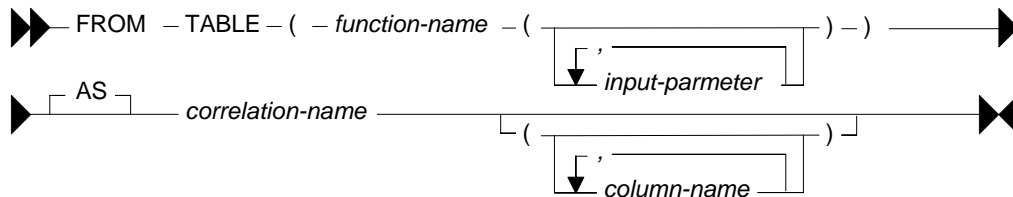


Figure 471, Table function usage - syntax

Note the following:

- The TABLE keyword, the function name (obviously), the two sets of parenthesis , and a correlation name, are all required.
- If the function has input parameters, they are all required, and their type must match.
- Optionally, one can list all of the columns that are returned by the function, giving each an assigned name

Below is an example of a function that uses all of the above features:

```
CREATE FUNCTION get_st(inval INTEGER)
RETURNS TABLE (id SMALLINT
                ,name VARCHAR(9)
                ,yr SMALLINT)
RETURN SELECT id
              ,name
              ,years
              FROM staff
              WHERE id = inval;

SELECT *
FROM TABLE(get_st(30)) AS sss (id, nnn, yy);
```

ANSWER		
=====		
ID	NNN	YY
--	-----	--
30	Marenghi	5

Figure 472, Table function with parameters

Examples

A table function returns a table, but it doesn't have to touch a table. To illustrate, the following function creates the data on the fly:

```
CREATE FUNCTION make_data()
  RETURNS TABLE (KY SMALLINT
                 ,DAT CHAR(5))
  RETURN WITH templ (k#) AS (VALUES (1),(2),(3))
         SELECT k#
            ,DIGITS(SMALLINT(k#))
         FROM   templ;

SELECT *
FROM   TABLE(make_data()) AS ttt;
```

ANSWER			
=====			
KY	DAT		
--	----		
1	00001		
2	00002		
3	00003		

Figure 473, Table function that creates data

The next example uses compound SQL to first flag an error if one of the input values is too low, then find the maximum salary and related ID in the matching set of rows, then fetch the same rows - returning the two previously found values at the same time:

```
CREATE FUNCTION staff_list(lo_key INTEGER
                          ,lo_sal INTEGER)
  RETURNS TABLE (id SMALLINT
                 ,salary DECIMAL(7,2)
                 ,max_sal DECIMAL(7,2)
                 ,id_max SMALLINT)

LANGUAGE SQL
READS SQL DATA
EXTERNAL ACTION
DETERMINISTIC
BEGIN ATOMIC
  DECLARE hold_sal DECIMAL(7,2) DEFAULT 0;
  DECLARE hold_key SMALLINT;
  IF lo_sal < 0 THEN
    SIGNAL SQLSTATE '75001'
    SET MESSAGE_TEXT = 'Salary too low';
  END IF;
  FOR get_max AS
    SELECT id AS in_key
           ,salary As in_sal
    FROM   staff
    WHERE  id >= lo_key
  DO
    IF in_sal > hold_sal THEN
      SET hold_sal = in_sal;
      SET hold_key = in_key;
    END IF;
  END FOR;
  RETURN
  SELECT id
         ,salary
         ,hold_sal
         ,hold_key
  FROM   staff
  WHERE  id >= lo_key;

END!
```

ANSWER			
=====			
ID	SALARY	MAX_SAL	ID_MAX
----	-----	-----	-----
70	16502.83	22959.20	160
80	13504.60	22959.20	160
90	18001.75	22959.20	160
100	18352.80	22959.20	160
110	12508.20	22959.20	160

```
SELECT *
FROM   TABLE(staff_list(66,1)) AS ttt
WHERE  id < 111
ORDER BY id!
```

Figure 474, Table function with compound SQL

Useful User-Defined Functions

In this section we will describe some simple functions that are generally useful, and that people have asked for over the years. In addition to the functions listed here, there are also the following elsewhere in this book:

- Check character input is a numeric value - page 375
- Convert numeric data to character (right justified) - page 377.
- Locate string in input, a block at a time - page 300.
- Pause SQL statement (by looping) for "n" seconds - page 394.
- Sort character field contents - page 393.

Julian Date Functions

The function below converts a DB2 date into a Julian date (format) value:

```
CREATE FUNCTION julian_out(inval DATE)
  RETURNS CHAR(7)
  RETURN  RTRIM(CHAR(YEAR(inval)))
          || SUBSTR(DIGITS(DAYOFYEAR(inval)),8);
```

	ANSWER		
	=====		=====
	EMPNO	H_DATE	J_DATE
	-----	-----	-----
SELECT empno	000010	1965-01-01	1965001
,CHAR(hiredate,ISO) AS h_date			
,JULIAN_OUT(hiredate) AS j_date			
FROM employee	000020	1973-10-10	1973283
WHERE empno < '000050'	000030	1975-04-05	1975095
ORDER BY empno;			

Figure 475, Convert Date into Julian Date

The next function does the opposite:

```
CREATE FUNCTION julian_in(inval CHAR(7))
  RETURNS DATE
  RETURN  DATE('0001-01-01')
          + (INT(SUBSTR(inval,1,4)) - 1) YEARS
          + (INT(SUBSTR(inval,5,3)) - 1) DAYS;
```

Figure 476, Convert Julian Date into Date

Get Prior Date

Imagine that one wanted to get all rows where some date is for the prior year - relative to the current year. This is easy to code:

```
SELECT empno
       ,hiredate
FROM   employee
WHERE  YEAR(hiredate) = YEAR(CURRENT DATE) - 1;
```

Figure 477, Select rows where hire-date = prior year

Get Prior Month

One can use the DAYS function to get the same data for the prior day. But one cannot use the MONTH function to do the equivalent for the prior month because at the first of the year the month number goes back to one.

One can address this issue by writing a simple function that multiplies the year-number by 12, and then adds the month-number:

```
CREATE FUNCTION year_month(inval DATE)
RETURNS INTEGER
RETURN (YEAR(inval) * 12) + MONTH(inval);
```

Figure 478, Create year-month function

We can use this function thus:

```
SELECT empno
       ,hiredate
FROM   employee
WHERE  YEAR_MONTH(hiredate) = YEAR_MONTH(CURRENT DATE) - 1;
```

Figure 479, Select rows where hire-date = prior month

Get Prior Week

Selecting rows for the prior week is complicated by the fact that both the US and ISO definitions of a week begin at one at the start of the year (see page 410). If however we choose to define a week as a set of seven contiguous days, regardless of the date, we can create a function to do the job. In the example below we shall assume that a week begins on a Sunday:

```
CREATE FUNCTION sunday_week(inval DATE)
RETURNS INTEGER
RETURN DAYS(inval) / 7;
```

Figure 480, Create week-number function

The next function assumes that a week begins on a Monday:

```
CREATE FUNCTION monday_week(inval DATE)
RETURNS INTEGER
RETURN (DAYS(inval) - 1) / 7;
```

Figure 481, Create week-number function

Both the above functions convert the input date into a day-number value, then subtract (if needed) to get to the right day of the week, then divide by seven to get a week-number. The result is the number of weeks since the beginning of the current era.

The next query shows the two functions in action:

WITH	ANSWER
temp1 (num,dt) AS	=====
(VALUES (1	DATE DAY WK IS SUN_WK MON_WK
,DATE('2004-12-29'))	----- -- -- --
UNION ALL	2004-12-29 Wed 53 53 104563 104563
SELECT num + 1	2004-12-30 Thu 53 53 104563 104563
,dt + 1 DAY	2004-12-31 Fri 53 53 104563 104563
FROM temp1	2005-01-01 Sat 1 53 104563 104563
WHERE num < 15	2005-01-02 Sun 2 53 104564 104563
)	2005-01-03 Mon 2 1 104564 104564
temp2 (dt,dy) AS	2005-01-04 Tue 2 1 104564 104564
(SELECT dt	2005-01-05 Wed 2 1 104564 104564
,SUBSTR(DAYNAME(dt),1,3)	2005-01-06 Thu 2 1 104564 104564
FROM temp1	2005-01-07 Fri 2 1 104564 104564
)	2005-01-08 Sat 2 1 104564 104564
SELECT CHAR(dt,ISO) AS date	2005-01-09 Sun 3 1 104565 104564
,dy AS day	2005-01-10 Mon 3 2 104565 104565
,WEEK(dt) AS wk	2005-01-11 Tue 3 2 104565 104565
,WEEK_ISO(dt) AS is	2005-01-12 Wed 3 2 104565 104565
,sunday_week(dt) AS sun_wk	
,monday_week(dt) AS mon_wk	
FROM temp2	
ORDER BY 1;	

Figure 482, Use week-number functions

Generating Numbers

The next function returns a table of rows. Each row consists of a single integer value, starting at zero, and going up to the number given in the input. At least one row is always returned. If the input value is greater than zero, the number of rows returned equals the input value plus one:

```
CREATE FUNCTION NumList(max_num INTEGER)
RETURNS TABLE(num INTEGER)
LANGUAGE SQL
RETURN
  WITH templ (num) AS
  (VALUES (0)
   UNION ALL
   SELECT num + 1
   FROM templ
   WHERE num < max_num
  )
  SELECT num
  FROM templ;
```

Figure 483, Create num-list function

Below are some queries that use the above function:

	ANSWERS
SELECT * FROM TABLE(NumList(-1)) AS xxx;	=====
	0
SELECT * FROM TABLE(NumList(+0)) AS xxx;	0
SELECT * FROM TABLE(NumList(+3)) AS xxx;	0 1 2 3
SELECT * FROM TABLE(NumList(CAST(NULL AS INTEGER))) AS xxx;	0

Figure 484, Using num-list function

NOTE: If this function did not always return one row, we might have to use a left-outer-join when joining to it. Otherwise the calling row might disappear from the answer-set because no row was returned.

To illustrate the function's usefulness, consider the following query, which returns the start and end date for a given set of activities:

	ANSWER
SELECT actno ,emstartdate ,emenddate ,DAYS(emenddate) - DAYS(emstartdate) AS #days	=====
FROM emp_act act	ACTNO EMSTARTDATE EMENDATE #DAYS
WHERE empno = '000260'	-----
AND projno = 'AD3113'	70 1982-06-15 1982-07-01 16
AND actno < 100	80 1982-03-01 1982-04-15 45
AND emptime = 0.5	
ORDER BY actno;	

Figure 485, Select activity start & end date

Imagine that we wanted take the above output, and generate a row for each day between the start and end dates. To do this we first have to calculate the number of days between a given start and end, and then join to the function using that value:


```

SELECT  actno
        ,#days
        ,num
        ,emstdate + num DAYS AS new_date
FROM    (SELECT  actno
              ,emstdate
              ,emendate
              ,DAYS(emendate) -
              DAYS(emstdate) AS #days
        FROM    emp_act act
        WHERE   empno = '000260'
        AND    projno = 'AD3113'
        AND    actno < 100
        AND    emptime = 0.5
        )AS aaa
        ,TABLE(NumList(#days)) AS ttt
ORDER BY actno
        ,num;

```

ANSWER				
=====				
ACTNO	#DAYS	NUM	NEW_DATE	

70	16	0	1982-06-15	
70	16	1	1982-06-16	
70	16	2	1982-06-17	
70	16	3	1982-06-18	
70	16	4	1982-06-19	
70	16	5	1982-06-20	
70	16	6	1982-06-21	
70	16	7	1982-06-22	
70	16	8	1982-06-23	
70	16	9	1982-06-24	
70	16	10	1982-06-25	
etc...				

Figure 486, Generate one row per date between start & end dates (1 of 2)

In the above query the #days value equals the number of days between the start and end dates. If the two dates equal, the #days value will be zero. In this case we will still get a row because the function will return a single zero value. If this were not the case (i.e. the function returned no rows if the input value was less than one), we would have to code a left-outer-join with a fake ON statement:

```

SELECT  actno
        ,#days
        ,num
        ,emstdate + num DAYS AS new_date
FROM    (SELECT  actno
              ,emstdate
              ,emendate
              ,DAYS(emendate) -
              DAYS(emstdate) AS #days
        FROM    emp_act act
        WHERE   empno = '000260'
        AND    projno = 'AD3113'
        AND    actno < 100
        AND    emptime = 0.5
        )AS aaa
LEFT OUTER JOIN
        TABLE(NumList(#days)) AS ttt
ON      1 = 1
ORDER BY actno
        ,num;

```

ACTNO	#DAYS	NUM	NEW_DATE	

70	16	0	1982-06-15	
70	16	1	1982-06-16	
70	16	2	1982-06-17	
70	16	3	1982-06-18	
70	16	4	1982-06-19	
70	16	5	1982-06-20	
70	16	6	1982-06-21	
70	16	7	1982-06-22	
70	16	8	1982-06-23	
70	16	9	1982-06-24	
70	16	10	1982-06-25	
etc...				

Figure 487, Generate one row per date between start & end dates (2 of 2)

Check Data Value Type

The following function checks to see if an input value is character, where character is defined as meaning that **all** bytes are "A" through "Z" or blank. It converts (if possible) all bytes to blank using the TRANSLATE function, and then checks to see if the result is blank:

```

CREATE FUNCTION ISCHAR (inval VARCHAR(250))
RETURNS SMALLINT
LANGUAGE SQL
RETURN
CASE
  WHEN TRANSLATE(UPPER(inval), ' ', 'ABCDEFGHIJKLMNOPQRSTUVWXYZ') = ' '
  THEN 1
  ELSE 0
END;

```

Figure 488, Check if input value is character

The next function is similar to the prior, except that it looks to see if all bytes in the input are in the range of "0" through "9", or blank:

```
CREATE FUNCTION ISNUM (inval VARCHAR(250))
RETURNS SMALLINT
LANGUAGE SQL
RETURN
CASE
    WHEN TRANSLATE(inval, ' ', '01234567890') = ' '
    THEN 1
    ELSE 0
END;
```

Figure 489, Check if input value is numeric

Below is an example of the above two functions in action:

<pre>WITH temp (indata) AS (VALUES ('ABC'), ('123'), ('3.4') , ('-44'), ('A1 '), (' ')) SELECT indata AS indata , ISCHAR(indata) AS c , ISNUM(indata) AS n FROM temp;</pre>	<pre>ANSWER ===== INDATA C N ----- - - ABC 1 0 123 0 1 3.4 0 0 -44 0 0 A1 0 0 1 1</pre>
---	--

Figure 490, Example of functions in use

The above ISNUM function is a little simplistic. It doesn't check for all-blanks, or embedded blanks, decimal input, or sign indicators. The next function does all of this, and also indicates what type of number was found:

```
CREATE FUNCTION ISNUM2 (inval VARCHAR(255))
RETURNS CHAR(4)
LANGUAGE SQL
RETURN
CASE
    WHEN inval = ' '
    THEN ' '
    WHEN LOCATE(' ', RTRIM(LTRIM(inval))) > 0
    THEN ' '
    WHEN TRANSLATE(inval, ' ', '01234567890') = inval
    THEN ' '
    WHEN TRANSLATE(inval, ' ', '01234567890') = ' '
    THEN 'INT '
    WHEN TRANSLATE(inval, ' ', '+01234567890') = ' '
    AND LOCATE('+', LTRIM(inval)) = 1
    AND LENGTH(REPLACE(inval, '+', '')) = LENGTH(inval) - 1
    THEN 'INT+'
    WHEN TRANSLATE(inval, ' ', '-01234567890') = ' '
    AND LOCATE('-', LTRIM(inval)) = 1
    AND LENGTH(REPLACE(inval, '-', '')) = LENGTH(inval) - 1
    THEN 'INT-'
    WHEN TRANSLATE(inval, ' ', '.01234567890') = ' '
    AND LENGTH(REPLACE(inval, '.', '')) = LENGTH(inval) - 1
    THEN 'DEC '
    WHEN TRANSLATE(inval, ' ', '+.01234567890') = ' '
    AND LOCATE('+', LTRIM(inval)) = 1
    AND LENGTH(REPLACE(inval, '+', '')) = LENGTH(inval) - 1
    AND LENGTH(REPLACE(inval, '.', '')) = LENGTH(inval) - 1
    THEN 'DEC+'
    ELSE ' '
END;
```

Figure 491, Check if input value is numeric - part 1 of 2

```

WHEN  TRANSLATE(inval, ' ', '-.01234567890') = ' '
AND   LOCATE('-',LTRIM(inval))              = 1
AND   LENGTH(REPLACE(inval, '-', ''))       = LENGTH(inval) - 1
AND   LENGTH(REPLACE(inval, '.', ''))       = LENGTH(inval) - 1
THEN  'DEC-'
ELSE  ' '
END;

```

Figure 492, Check if input value is numeric - part 2 of 2

The first three WHEN checks above are looking for non-numeric input:

- The input is blank.
- The input has embedded blanks.
- The input does not contain any digits.

The final five WHEN checks look for a specific types of numeric input. They are all similar in design, so we can use the last one (looking of negative decimal input) to illustrate how they all work:

- Check that the input consists only of digits, dots, the minus sign, and blanks.
- Check that the minus sign is the left-most non-blank character.
- Check that there is only one minus sign in the input.
- Check that there is only one dot in the input.

Below is an example of the above function in use:

WITH temp (indata) AS			ANSWER
(VALUES ('ABC'), ('123'), ('3.4'))			=====
, ('-44'), ('+11'), ('-1-'))			INDATA TYPE NUMBER
, ('12+'), ('+.1'), ('-0.'))			-----
, (' '), ('1 1'), ('. '))			ABC -
SELECT indata AS indata			123 INT 123.00
, ISNUM2(indata) AS type			3.4 DEC 3.40
, CASE			-44 INT- -44.00
WHEN ISNUM2(indata) <> ''			+11 INT+ 11.00
THEN DEC(indata, 5, 2)			-1- -
ELSE NULL			12+ -
END AS number			+1 DEC+ 0.10
FROM temp;			-0. DEC- 0.00
			-
			1 1 -
			.
			-

Figure 493, Example of function in use

Order By, Group By, and Having

Order By

The ORDER BY statement is used to sequence output rows. The syntax goes as follows:

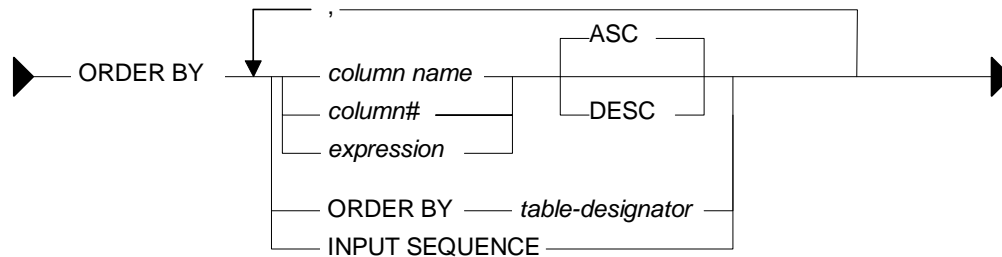


Figure 494, ORDER BY syntax

Notes

One can order on any one of the following:

- A named column, or an expression, neither of which need to be in the select list.
- An unnamed column - identified by its number in the list of columns selected.
- The ordering sequence of a specific nested subselect.
- For an insert, the order in which the rows were inserted (see page 67).

Also note:

- One can have multiple ORDER BY statements in a query, but only one per subselect.
- Specifying the same field multiple times in an ORDER BY list is allowed, but silly. Only the first specification of the field will have any impact on the output order.
- If the ORDER BY column list does not uniquely identify each row, any rows with duplicate values will come out in random order. This is almost always the wrong thing to do when the data is being displayed to an end-user.
- Use the TRANSLATE function to order data regardless of case. Note that this trick may not work consistently with some European character sets.
- NULL values sort high.

Sample Data

The following view is used throughout this section:

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

Figure 495, ORDER BY sample data definition

Order by Examples

The following query presents the output in ascending order:

```

SELECT  col1
        ,col2
FROM    seq_data
ORDER BY col1 ASC
        ,col2;

```

ANSWER		SEQ_DATA
=====		+-----+
COL1	COL2	COL1 COL2
-----		+-----+
ab	xy	ab xy
ac	XY	AB xy
Ab	12	ac XY
AB	xy	AB XY
AB	XY	Ab 12

Figure 496, Simple ORDER BY

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

```

SELECT  col1
        ,col2
FROM    seq_data
ORDER BY TRANSLATE(col1) ASC
        ,TRANSLATE(col2) ASC;

```

ANSWER	
=====	
COL1	COL2

Ab	12
ab	xy
AB	XY
AB	xy
ac	XY

Figure 497, 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
FROM    seq_data
ORDER BY col1
        ,col2;

```

ANSWER	
=====	
COL2	

xy	
XY	
12	
xy	
XY	

Figure 498, 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
        ,col2
FROM    seq_data
ORDER BY SUBSTR(col1,2) DESC
        ,col2;

```

ANSWER	
=====	
COL1	COL2

ac	XY
AB	xy
AB	XY
Ab	12
ab	xy

Figure 499, 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    coll
          ,HEX(coll) AS hex1
          ,col2
          ,HEX(col2) AS hex2
FROM      seq_data
ORDER BY  HEX(coll)
          ,HEX(col2)

```

ANSWER			
=====			
COL1	HEX1	COL2	HEX2

AB	4142	XY	5859
AB	4142	xy	7879
Ab	4162	12	3132
ab	6162	xy	7879
ac	6163	XY	5859

Figure 500, ORDER BY in bit-data sequence

ORDER BY subselect

One can order by the result of a nested ORDER BY, thus enabling one to order by a column that is not in the input - as is done below:

```

SELECT    coll
FROM      (SELECT    coll
          FROM      seq_data
          ORDER BY  col2
          ) AS xxx
ORDER BY  ORDER OF xxx;

```

ANSWER	SEQ_DATA	
=====	+-----+	
COL1	COL1	COL2
----	----	----
Ab	ab	xy
ab	AB	xy
AB	ac	XY
ac	AB	XY
AB	Ab	12
+-----+		

Figure 501, ORDER BY nested ORDER BY

In the next example the ordering of the innermost subselect is used, in part, to order the final output. This is done by first referring it to directly, and then indirectly:

```

SELECT    *
FROM      (SELECT    *
          FROM      (SELECT    *
                    FROM      seq_data
                    ORDER BY  col2
                    )AS xxx
          ORDER BY  ORDER OF xxx
                    ,SUBSTR(coll,2)
          )AS yyy
ORDER BY  ORDER OF yyy
          ,coll;

```

ANSWER	
=====	
COL1	COL2
----	----
Ab	12
ab	xy
AB	xy
AB	XY
ac	XY

Figure 502, Multiple nested ORDER BY statements

ORDER BY inserted rows

One can select from an insert statement (see page 67) to see what was inserted. Order by the INSERT SEQUENCE to display the rows in the order that they were inserted:

```

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

```

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

Figure 503, ORDER BY insert input sequence

NOTE: The INPUT SEQUENCE phrase only works in an insert statement. It can be listed in the ORDER BY part of the statement, but not in the SELECT part. The select cannot be a nested table expression.

Group By and Having

The GROUP BY and GROUPING SETS statements are used to group individual rows into combined sets based on the value in one, or more, columns. The related ROLLUP and CUBE statements are short-hand forms of particular types of GROUPING SETS statement.

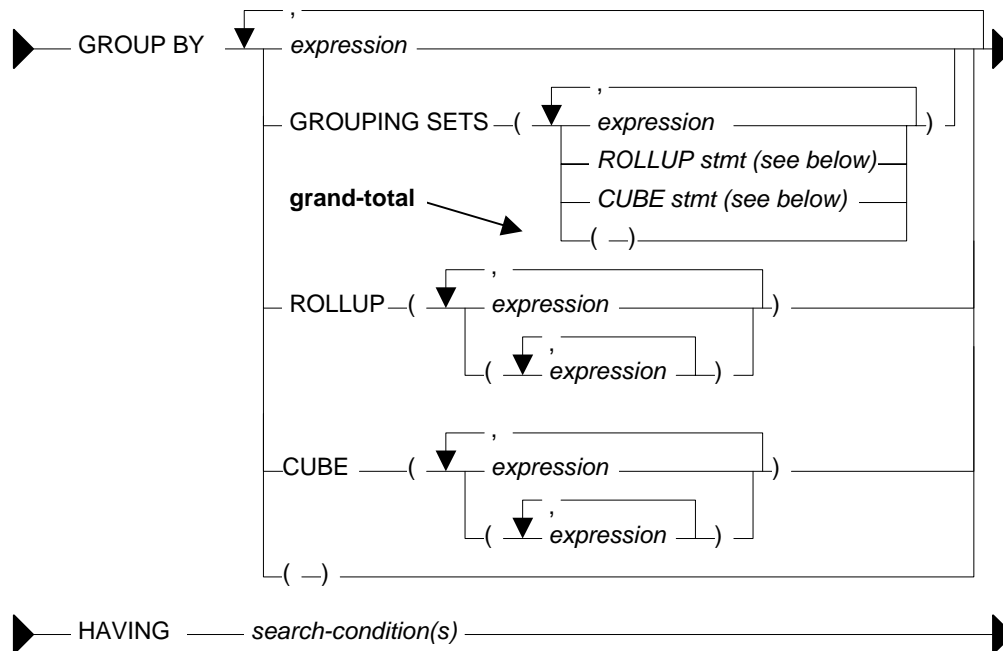


Figure 504, GROUP BY syntax

Rules and Restrictions

- There can only be one GROUP BY per SELECT. Multiple select statements in the same query can each have their own GROUP BY.
- Every field in the SELECT list must either be specified in the GROUP BY, or must have a column function applied against it.
- The result of a simple GROUP BY is always a distinct set of rows, where the unique identifier is whatever fields were grouped on.
- Only expressions returning constant values (e.g. a column name, a constant) can be referenced in a GROUP BY. For example, one cannot group on the RAND function as its result varies from one call to the next. To reference such a value in a GROUP BY, resolve it beforehand using a nested-table-expression.
- Variable length character fields with differing numbers on trailing blanks are treated as equal in the GROUP. The number of trailing blanks, if any, in the result is unpredictable.
- When grouping, all null values in the GROUP BY fields are considered equal.
- There is no guarantee that the rows resulting from a GROUP BY will come back in any particular order. If this is a problem, use an ORDER BY.

GROUP BY Flavors

A typical GROUP BY that encompasses one or more fields is actually a subset of the more general GROUPING SETS command. In a grouping set, one can do the following:

- Summarize the selected data by the items listed such that one row is returned per unique combination of values. This is an ordinary GROUP BY.
- Summarize the selected data using multiple independent fields. This is equivalent to doing multiple independent GROUP BY statements - with the separate results combined into one using UNION ALL statements.
- Summarize the selected data by the items listed such that one row is returned per unique combination of values, and also get various sub-totals, plus a grand-total. Depending on what exactly is wanted, this statement can be written as a ROLLUP, or a CUBE.

To illustrate the above concepts, imagine that we want to group some company data by team, department, and division. The possible sub-totals and totals that we might want to get are:

```
GROUP BY division, department, team
GROUP BY division, department
GROUP BY division
GROUP BY division, team
GROUP BY department, team
GROUP BY department
GROUP BY team
GROUP BY ()      <= grand-total
```

Figure 505, Possible groupings

If we wanted to get the first three totals listed above, plus the grand-total, we could write the statement one of three ways:

```
GROUP BY division, department, team
UNION ALL
GROUP BY division, department
UNION ALL
GROUP BY division
UNION ALL
GROUP BY ()

GROUP BY GROUPING SETS ((division, department, team)
                        ,(division, department)
                        ,(division)
                        ,())

GROUP BY ROLLUP (division, department, team)
```

Figure 506, Three ways to write the same GROUP BY

Usage Warnings

Before we continue, be aware of the following:

- Single vs. double parenthesis is a very big deal in grouping sets. When using the former, one is listing multiple independent groupings, while with the latter one is listing the set of items in a particular grouping.
- Repetition matters - sometimes. In an ordinary GROUP BY duplicate references to the same field has no impact on the result. By contrast, in a GROUPING SET, ROLLUP, or CUBE statement, duplicate references can often result in the same set of data being retrieved multiple times.

GROUP BY Sample Data

The following view will be used throughout this section:

```
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;
```

```
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 507, 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.

Sample Queries

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

```
SELECT  d1, dept, sex
        ,SUM(salary)          AS salary
        ,SMALLINT(COUNT(*)) AS #rows
FROM    employee_view
WHERE   dept <> 'ABC'
GROUP BY d1, dept, sex
HAVING  dept > 'A0'
        AND (SUM(salary) > 100
             OR MIN(salary) > 10
             OR COUNT(*) <> 22)
ORDER BY d1, dept, sex;
```

```
ANSWER
=====
D1 DEPT SEX SALARY #ROWS
--- ---
A  A00  F   52750     1
A  A00  M   75750     2
B  B01  M   41250     1
C  C01  F   90470     3
D  D11  F   73430     3
D  D11  M  148670     6
```

Figure 508, Simple GROUP BY

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

```
SELECT  sex
        ,SUM(salary)          AS salary
        ,SMALLINT(COUNT(*)) AS #rows
FROM    employee_view
WHERE   sex IN ('F','M')
GROUP BY dept
        ,sex
ORDER BY sex;
```

```
ANSWER
=====
SEX SALARY #ROWS
--- ---
F   52750     1
F   90470     3
F   73430     3
M   75750     2
M   41250     1
M  148670     6
```

Figure 509, 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
WHERE     dl <> 'X'
GROUP BY  SUBSTR(dept,3,1)
HAVING    COUNT(*) <> 99;
          128500           3
          353820          13

```

Figure 510, *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
          ,SUM(salary)        AS salary
          ,SMALLINT(COUNT(*)) AS #rows
FROM      employee_view
GROUP BY  SUBSTR(dept,3,1)
ORDER BY  wpart DESC;
          1      353820    13
          0      128500    3

```

Figure 511, *GROUP BY on derived field, shown*

GROUPING SETS Statement

The **GROUPING SETS** statement enable one to get multiple **GROUP BY** result sets using a single statement. It is important to understand the difference between nested (i.e. in secondary parenthesis), and non-nested **GROUPING SETS** sub-phrases:

- A nested list of columns works as a simple **GROUP BY**.
- A non-nested list of columns works as separate simple **GROUP BY** statements, which are then combined in an implied **UNION ALL**.

```

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 512, *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)
          ,GROUPING SETS (B)
          ,GROUPING SETS (C)
          is equivalent to    GROUP BY A
                              ,B
                              ,C

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

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

```

Figure 513, *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 514, 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 515, Mixing simple GROUP BY expressions and GROUPING SETS

A single GROUPING SETS statement can contain multiple sets of (implied) GROUP BY phrases. 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 516, 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
                                grand-totl
ROLLUP(A,B,C)

```

Figure 517, 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
                                     ,C
is equivalent to                UNION ALL
                                GROUP BY A
                                UNION ALL
                                GROUP BY B
                                UNION ALL
                                GROUP BY C
                                UNION ALL
                                grand-totl
CUBE(A,B,C)

```

Figure 518, 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 519, 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
        ,dept
        ,sex
        ,SUM(salary)          AS sal
        ,SMALLINT(COUNT(*)) AS #r
        ,GROUPING(d1)        AS fl
        ,GROUPING(dept)     AS fd
        ,GROUPING(sex)      AS fs
FROM    employee_view
GROUP BY GROUPING SETS (d1)
        ,GROUPING SETS (dept,sex)
ORDER BY d1
        ,dept
        ,sex;

```

ANSWER							
D1	DEPT	SEX	SAL	#R	F1	FD	FS
A	A00	-	128500	3	0	0	1
A	-	F	52750	1	0	1	0
A	-	M	75750	2	0	1	0
B	B01	-	41250	1	0	0	1
B	-	M	41250	1	0	1	0
C	C01	-	90470	3	0	0	1
C	-	F	90470	3	0	1	0
D	D11	-	222100	9	0	0	1
D	-	F	73430	3	0	1	0
D	-	M	148670	6	0	1	0

Figure 520, 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
        ,dept
        ,sex
        ,SUM(salary)          AS sal
        ,SMALLINT(COUNT(*)) AS #r
        ,GROUPING(d1)        AS fl
        ,GROUPING(dept)     AS fd
        ,GROUPING(sex)      AS fs
FROM    employee_view
GROUP BY d1
        ,dept
        ,GROUPING SETS ((dept,sex))
ORDER BY d1
        ,dept
        ,sex;

```

ANSWER							
D1	DEPT	SEX	SAL	#R	F1	FD	FS
A	A00	F	52750	1	0	0	0
A	A00	M	75750	2	0	0	0
B	B01	M	41250	1	0	0	0
C	C01	F	90470	3	0	0	0
D	D11	F	73430	3	0	0	0
D	D11	M	148670	6	0	0	0

Figure 521, Repeated field essentially ignored

```

SELECT  d1
        ,dept
        ,sex
        ,SUM(salary)          AS sal
        ,SMALLINT(COUNT(*)) AS #r
        ,GROUPING(d1)        AS fl
        ,GROUPING(dept)     AS fd
        ,GROUPING(sex)      AS fs
FROM    employee_view
GROUP BY d1
        ,DEPT
        ,GROUPING SETS (dept,sex)
ORDER BY d1
        ,dept
        ,sex;

```

ANSWER							
D1	DEPT	SEX	SAL	#R	F1	FD	FS
A	A00	F	52750	1	0	0	0
A	A00	M	75750	2	0	0	0
A	A00	-	128500	3	0	0	1
B	B01	M	41250	1	0	0	0
B	B01	-	41250	1	0	0	1
C	C01	F	90470	3	0	0	0
C	C01	-	90470	3	0	0	1
D	D11	F	73430	3	0	0	0
D	D11	M	148670	6	0	0	0
D	D11	-	222100	9	0	0	1

Figure 522, 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 523, 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 524, 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 525, 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 526, ROLLUP vs. GROUPING SETS

One way to understand the above is to convert the two ROLLUP statement into equivalent grouping sets, and them "multiply" them - ignoring any grand-totals except when they are on both sides of the equation:

```

ROLLUP(A)                *  ROLLUP(B,C)          =  GROUPING SETS((A,B,C)
                                     ,(A,B)
                                     ,(A)
                                     ,(B,C)
                                     ,(B)
                                     ,())

GROUPING SETS((A)        *  GROUPING SETS((B,C)   =
      ,())                , (B)
                                     ,(B)
                                     ,())

```

Figure 527, Multiplying GROUPING SETS

SQL Examples

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

```

SELECT dept
       ,SUM(salary)           AS salary
       ,SMALLINT(COUNT(*))   AS #rows
       ,GROUPING(dept)      AS fd
FROM   employee_view
GROUP BY dept
ORDER BY dept;

```

ANSWER			
DEPT	SALARY	#ROWS	FD
A00	128500	3	0
B01	41250	1	0
C01	90470	3	0
D11	222100	9	0

Figure 528, 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
       ,SUM(salary)           AS salary
       ,SMALLINT(COUNT(*))   AS #rows
       ,GROUPING(dept)      AS FD
FROM   employee_view
GROUP BY ROLLUP(dept)
ORDER BY dept;

```

ANSWER			
DEPT	SALARY	#ROWS	FD
A00	128500	3	0
B01	41250	1	0
C01	90470	3	0
D11	222100	9	0
-	482320	16	1

Figure 529, 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
       ,SUM(salary)           AS salary
       ,SMALLINT(COUNT(*))   AS #rows
       ,GROUPING(dept)      AS fd
FROM   employee_view
GROUP BY dept
UNION ALL
SELECT CAST(NULL AS CHAR(3)) AS dept
       ,SUM(salary)           AS salary
       ,SMALLINT(COUNT(*))   AS #rows
       ,CAST(1 AS INTEGER)   AS fd
FROM   employee_view
ORDER BY dept;

```

ANSWER			
DEPT	SALARY	#ROWS	FD
A00	128500	3	0
B01	41250	1	0
C01	90470	3	0
D11	222100	9	0
-	482320	16	1

Figure 530, 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
       ,SUM(salary)           AS salary
       ,SMALLINT(COUNT(*))   AS #rows
       ,GROUPING(dept)      AS fd
FROM   employee_view
GROUP BY dept
       ,ROLLUP(dept)
ORDER BY dept;

```

ANSWER			
DEPT	SALARY	#ROWS	FD
A00	128500	3	0
A00	128500	3	0
B01	41250	1	0
B01	41250	1	0
C01	90470	3	0
C01	90470	3	0
D11	222100	9	0
D11	222100	9	0

Figure 531, 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 532, 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 533, 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 534, 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
        ,dept
        ,SUM(salary)           AS salary
        ,SMALLINT(COUNT(*))   AS #rows
        ,GROUPING(dept)       AS fd
        ,GROUPING(sex)        AS fs
FROM    employee_view
GROUP BY ROLLUP(sex
                ,dept)
ORDER BY sex
        ,dept;

```

ANSWER					
SEX	DEPT	SALARY	#ROWS	FD	FS
F	A00	52750	1	0	0
F	C01	90470	3	0	0
F	D11	73430	3	0	0
F	-	216650	7	1	0
M	A00	75750	2	0	0
M	B01	41250	1	0	0
M	D11	148670	6	0	0
M	-	265670	9	1	0
-	-	482320	16	1	1

Figure 535, 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
        ,dept
        ,SUM(salary)           AS salary
        ,SMALLINT(COUNT(*))   AS #rows
        ,GROUPING(dept)       AS fd
        ,GROUPING(sex)        AS fs
FROM    employee_view
GROUP BY GROUPING SETS ((sex, dept)
                        ,(sex)
                        ,())
ORDER BY sex
        ,dept;

```

ANSWER					
SEX	DEPT	SALARY	#ROWS	FD	FS
F	A00	52750	1	0	0
F	C01	90470	3	0	0
F	D11	73430	3	0	0
F	-	216650	7	1	0
M	A00	75750	2	0	0
M	B01	41250	1	0	0
M	D11	148670	6	0	0
M	-	265670	9	1	0
-	-	482320	16	1	1

Figure 536, ROLLUP on SEX, then DEPT

The next example has two independent rollups:

- The first generates a summary row for each sex.
- The second generates a summary row for each work-department.

The two together make a (single) combined summary row of all matching data. This query is the same as a UNION of the two individual rollups, but it has the advantage of being done in a single pass of the data. The result is the same as a CUBE of the two fields:

```

SELECT  sex
        ,dept
        ,SUM(salary)           AS salary
        ,SMALLINT(COUNT(*))   AS #rows
        ,GROUPING(dept)       AS fd
        ,GROUPING(sex)        AS fs
FROM    employee_view
GROUP BY ROLLUP(sex)
        ,ROLLUP(dept)
ORDER BY sex
        ,dept;

```

ANSWER					
SEX	DEPT	SALARY	#ROWS	FD	FS
F	A00	52750	1	0	0
F	C01	90470	3	0	0
F	D11	73430	3	0	0
F	-	216650	7	1	0
M	A00	75750	2	0	0
M	B01	41250	1	0	0
M	D11	148670	6	0	0
M	-	265670	9	1	0
-	A00	128500	3	0	1
-	B01	41250	1	0	1
-	C01	90470	3	0	1
-	D11	222100	9	0	1
-	-	482320	16	1	1

Figure 537, 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 538, 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 539, 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 540, 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 541, 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 542, 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 543, CUBE vs. GROUPING SETS

As with the ROLLUP 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 544, 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)
                                , (A,B,C), (A,B), (A,C), (A)
                                , (B,C), (B), (B,C), (B)
                                , (B,C), (B), (C), ())
                                , CUBE(B,C)
    
```

Figure 545, 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  dl
        ,dept
        ,sex
        ,INT(SUM(salary)) AS sal
        ,SMALLINT(COUNT(*)) AS #r
        ,GROUPING(dl) AS fl
        ,GROUPING(dept) AS fd
        ,GROUPING(sex) AS fs
FROM    employee_view
GROUP BY CUBE(dl, dept, sex)
ORDER BY dl
        ,dept
        ,sex;

```

ANSWER							
Dl	DEPT	SEX	SAL	#R	F1	FD	FS
A	A00	F	52750	1	0	0	0
A	A00	M	75750	2	0	0	0
A	A00	-	128500	3	0	0	1
A	-	F	52750	1	0	1	0
A	-	M	75750	2	0	1	0
A	-	-	128500	3	0	1	1
B	B01	M	41250	1	0	0	0
B	B01	-	41250	1	0	0	1
B	-	M	41250	1	0	1	0
B	-	-	41250	1	0	1	1
C	C01	F	90470	3	0	0	0
C	C01	-	90470	3	0	0	1
C	-	F	90470	3	0	1	0
C	-	-	90470	3	0	1	1
D	D11	F	73430	3	0	0	0
D	D11	M	148670	6	0	0	0
D	D11	-	222100	9	0	0	1
D	-	F	73430	3	0	1	0
D	-	M	148670	6	0	1	0
D	-	-	222100	9	0	1	1
-	A00	F	52750	1	1	0	0
-	A00	M	75750	2	1	0	0
-	A00	-	128500	3	1	0	1
-	B01	M	41250	1	1	0	0
-	B01	-	41250	1	1	0	1
-	C01	F	90470	3	1	0	0
-	C01	-	90470	3	1	0	1
-	D11	F	73430	3	1	0	0
-	D11	M	148670	6	1	0	0
-	D11	-	222100	9	1	0	1
-	-	F	216650	7	1	1	0
-	-	M	265670	9	1	1	0
-	-	-	482320	16	1	1	1

Figure 546, CUBE example

Here is the same query expressed as GROUPING SETS;

```

SELECT  dl
        ,dept
        ,sex
        ,INT(SUM(salary)) AS sal
        ,SMALLINT(COUNT(*)) AS #r
        ,GROUPING(dl) AS fl
        ,GROUPING(dept) AS fd
        ,GROUPING(sex) AS fs
FROM    employee_view
GROUP BY GROUPING SETS ((dl, dept, sex)
                        ,(dl,dept)
                        ,(dl,sex)
                        ,(dept,sex)
                        ,(dl)
                        ,(dept)
                        ,(sex)
                        ,())
ORDER BY dl
        ,dept
        ,sex;

```

ANSWER							
Dl	DEPT	SEX	SAL	#R	F1	FD	FS
A	A00	F	52750	1	0	0	0
A	A00	M	75750	2	0	0	0
etc... (same as prior query)							

Figure 547, CUBE expressed using multiple 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
        ,dept
        ,sex
        ,INT(SUM(salary)) AS sal
        ,SMALLINT(COUNT(*)) AS #r
        ,GROUPING(d1) AS fl
        ,GROUPING(dept) AS fd
        ,GROUPING(sex) AS fs
FROM    employee_VIEW
GROUP BY CUBE((d1, dept, sex))
ORDER BY d1
        ,dept
        ,sex;

```

ANSWER							
D1	DEPT	SEX	SAL	#R	F1	FD	FS
A	A00	F	52750	1	0	0	0
A	A00	M	75750	2	0	0	0
B	B01	M	41250	1	0	0	0
C	C01	F	90470	3	0	0	0
D	D11	F	73430	3	0	0	0
D	D11	M	148670	6	0	0	0
-	-	-	482320	16	1	1	1

Figure 548, CUBE on compound fields

The above query is resolved thus:

```

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

```

Figure 549, 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
        ,dept AS dpt
        ,sex AS sx
        ,INT(SUM(salary)) AS sal
        ,SMALLINT(COUNT(*)) AS r
FROM    employee_VIEW
GROUP BY d1
        ,dept
        ,sex
ORDER BY 1,2,3;

```

ANSWER					
D1	DPT	SX	SAL	R	
A	A00	F	52750	1	
A	A00	M	75750	2	
B	B01	M	41250	1	
C	C01	F	90470	3	
D	D11	F	73430	3	
D	D11	M	148670	6	

Figure 550, Basic GROUP BY example

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

DESIRED SUB-TOTALS	EQUIVALENT 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.	EQUIVALENT TO ,()
	=====
	GROUP BY ROLLUP(d1,dept)
	,ROLLUP(sex)

Figure 551, 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
                   GROUP BY CUBE(d1,dept,sex)
                   )AS xxx
WHERE       (g1,gd,gs) = (0,0,0)
OR         (g1,gd,gs) = (0,0,1)
OR         (g1,gd,gs) = (0,1,0)
OR         (g1,gd,gs) = (0,1,1)
OR         (g1,gd,gs) = (1,1,0)
OR         (g1,gd,gs) = (1,1,1)
ORDER BY   1,2,3;

```

ANSWER								
D1	DPT	SX	SAL	#R	G1	GD	GS	
A	A00	F	52750	1	0	0	0	
A	A00	M	75750	2	0	0	0	
A	A00	-	128500	3	0	0	1	
A	-	F	52750	1	0	1	0	
A	-	M	75750	2	0	1	0	
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 552, Get lots of sub-totals, using CUBE

In the above query, the GROUPING function (see page 89) 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 553, 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 554, 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 555, 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
  (SELECT   dept
   ,AVG(salary)
   FROM     staff
   GROUP BY dept
   HAVING   AVG(salary) > 18000
  )
SELECT   a.id
   ,a.name
   ,a.dept
FROM     staff a
   ,staff2 b
WHERE    a.dept = b.dept
ORDER BY a.id;

```

```

ANSWER
=====
ID  NAME      DEPT
---  -
160 Molinare  10
210 Lu       10
240 Daniels  10
260 Jones    10

```

Figure 556, *GROUP BY on one side of join - using common table expression*

In the next example, we shall use a fullselect:

```

SELECT   a.id
   ,a.name
   ,a.dept
FROM     staff a
   ,(SELECT   dept           AS dept
   ,AVG(salary) AS avgsal
   FROM     staff
   GROUP BY dept
   HAVING   AVG(salary) > 18000
  )AS b
WHERE    a.dept = b.dept
ORDER BY a.id;

```

```

ANSWER
=====
ID  NAME      DEPT
---  -
160 Molinare  10
210 Lu       10
240 Daniels  10
260 Jones    10

```

Figure 557, *GROUP BY on one side of join - using fullselect*

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
FROM     staff
WHERE    id < 1;

```

```

ANSWER
=====
0

```

```

SELECT   COUNT(*) AS c1
FROM     staff
WHERE    id < 1
GROUP BY id;

```

```

ANSWER
=====
no row

```

Figure 558, *COUNT and No Rows*

See page 404 for a comprehensive discussion of what happens when no rows match.

Joins

A join is used to relate sets of rows in two or more logical tables. The tables are always joined on a row-by-row basis using whatever join criteria are provided in the query. The result of a join is always a new, albeit possibly empty, set of rows.

In a join, the matching rows are joined side-by-side to make the result table. By contrast, in a union (see page 239) the matching rows are joined (in a sense) one-above-the-other to make the result table.

Why Joins Matter

The most important data in a relational database is not that stored in the individual rows. Rather, it is the implied relationships between sets of related rows. For example, individual rows in an EMPLOYEE table may contain the employee ID and salary - both of which are very important data items. However, it is the set of all rows in the same table that gives the gross wages for the whole company, and it is the (implied) relationship between the EMPLOYEE and DEPARTMENT tables that enables one to get a breakdown of employees by department and/or division.

Joins are important because one uses them to tease the relationships out of the database. They are also important because they are very easy to get wrong.

Sample Views

```
CREATE VIEW staff_v1 AS
SELECT id, name
FROM   staff
WHERE  ID BETWEEN 10 AND 30;
```

```
CREATE VIEW staff_v2 AS
SELECT id, job
FROM   staff
WHERE  id BETWEEN 20 AND 50
UNION ALL
SELECT id, 'Clerk' AS job
FROM   staff
WHERE  id = 30;
```

STAFF_V1		STAFF_V2	
ID	NAME	ID	JOB
10	Sanders	20	Sales
20	Pernal	30	Clerk
30	Marenghi	30	Mgr
		40	Sales
		50	Mgr

Figure 559, Sample Views used in Join Examples

Observe that the above two views have the following characteristics:

- Both views contain rows that have no corresponding ID in the other view.
- In the V2 view, there are two rows for ID of 30.

Join Syntax

DB2 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.

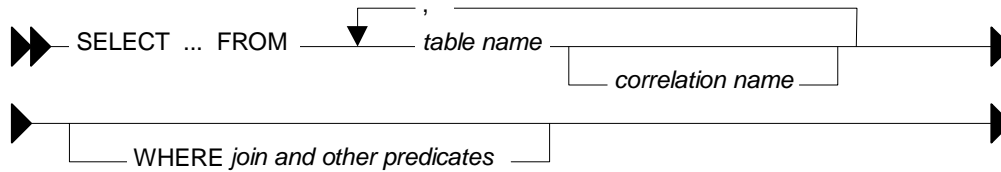


Figure 560, Join Syntax #1

Here are some sample joins:

```

SELECT    v1.id
          ,v1.name
          ,v2.job
FROM      staff_v1 v1
          ,staff_v2 v2
WHERE     v1.id = v2.id
ORDER BY v1.id
          ,v2.job;
    
```

	JOIN ANSWER
	=====
	ID NAME JOB

	20 Pernal Sales
	30 Marenghi Clerk
	30 Marenghi Mgr

Figure 561, Sample two-table join

```

SELECT    v1.id
          ,v2.job
          ,v3.name
FROM      staff_v1 v1
          ,staff_v2 v2
          ,staff_v1 v3
WHERE     v1.id = v2.id
          AND v2.id = v3.id
          AND v3.name LIKE 'M%'
ORDER BY v1.name
          ,v2.job;
    
```

	JOIN ANSWER
	=====
	ID JOB NAME

	30 Clerk Marenghi
	30 Mgr Marenghi

Figure 562, Sample three-table join

The second join style, which is suitable for both inner and outer joins, involves joining the tables two at a time, listing the type of join as one goes. ON conditions constrain the join (note: there must be at least one), while WHERE conditions are applied after the join and constrain the result.

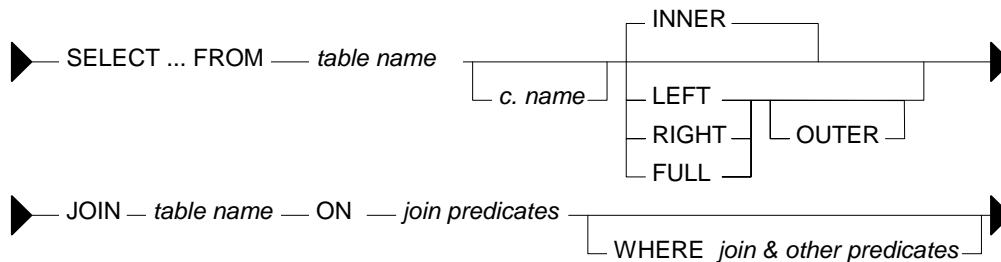


Figure 563, Join Syntax #2

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

```

SELECT    v1.id
          ,v1.name
          ,v2.job
FROM      staff_v1 v1
INNER JOIN
          staff_v2 v2
ON        v1.id = v2.id
ORDER BY v1.id
          ,v2.job;
    
```

	JOIN ANSWER
	=====
	ID NAME JOB

	20 Pernal Sales
	30 Marenghi Clerk
	30 Marenghi Mgr

Figure 564, Sample two-table inner join

```

SELECT    v1.id
          ,v2.job
          ,v3.name
FROM      staff_v1 v1
JOIN      staff_v2 v2
ON        v1.id = v2.id
JOIN      staff_v1 v3
ON        v2.id = v3.id
WHERE     v3.name LIKE 'M%'
ORDER BY v1.name
          ,v2.job;

```

STAFF_V1		STAFF_V2	
ID	NAME	ID	JOB
10	Sanders	20	Sales
20	Pernal	30	Clerk
30	Marenghi	30	Mgr
		40	Sales
		50	Mgr

```

JOIN ANSWER
=====
ID JOB   NAME
-----
30 Clerk Marenghi
30 Mgr   Marenghi

```

Figure 565, Sample three-table inner join

Query Processing Sequence

The following table lists the sequence with which various parts of a query are executed:

```

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

```

Figure 566, Query Processing Sequence

Observe that ON predicates (e.g. in an outer join) are always processed before any WHERE predicates (in the same join) are applied. Ignoring this processing sequence can cause what looks like an outer join to run as an inner join - see figure 578.

ON vs. WHERE

A join written using the second syntax style shown above can have either, or both, ON and WHERE checks. These two types of check work quite differently:

- WHERE checks are used to filter rows, and to define the nature of the join. Only those rows that match all WHERE checks are returned.
- ON checks define the nature of the join. They are used to categorize rows as either joined or not-joined, rather than to exclude rows from the answer-set, though they may do this in some situations.

Let illustrate this difference with a simple, if slightly silly, left outer join:

```

SELECT    *
FROM      staff_v1 v1
LEFT OUTER JOIN
          staff_v2 v2
ON        1 = 1
AND       v1.id = v2.id
ORDER BY v1.id
          ,v2.job;

```

ANSWER			
=====			
ID	NAME	ID	JOB

10	Sanders	-	-
20	Pernal	20	Sales
30	Marenghi	30	Clerk
30	Marenghi	30	Mgr

Figure 567, Sample Views used in Join Examples

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

```

SELECT  *
FROM    staff_v1 v1
LEFT OUTER JOIN
        staff_v2 v2
ON      1      = 1
WHERE   v1.id  = v2.id
ORDER BY v1.id
        ,v2.job;

```

ANSWER			
ID	NAME	ID	JOB
20	Pernal	20	Sales
30	Marenghi	30	Clerk
30	Marenghi	30	Mgr

Figure 568, Sample Views used in Join Examples

In the first example above, all rows were retrieved from the V1 view. Then, for each row, the two ON checks were used to find matching rows in the V2 view. In the second query, all rows were again retrieved from the V1 view. Then each V1 row was joined to every row in the V2 view using the (silly) ON check. Finally, the WHERE check (which is always done after the join) was applied to filter out all pairs that do not match on ID.

Can an ON check ever exclude rows? The answer is complicated:

- In an inner join, an ON check can exclude rows because it is used to define the nature of the join and, by definition, in an inner join only matching rows are returned.
- In a partial outer join, an ON check on the originating table does not exclude rows. It simply categorizes each row as participating in the join or not.
- In a partial outer join, an ON check on the table to be joined to can exclude rows because if the row fails the test, it does not match the join.
- In a full outer join, an ON check never excludes rows. It simply categorizes them as matching the join or not.

Each of the above principles will be demonstrated as we look at the different types of join.

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	JOB
10	Sanders	20	Sales	Join on ID =====>	20	Sales
20	Pernal	30	Clerk		30	Clerk
30	Marenghi	30	Mgr		30	Mgr
		40	Sales			
		50	Mgr			

Figure 569, Example of Inner Join

```

SELECT *
FROM   staff_v1 v1
       ,staff_v2 v2
WHERE  v1.id = v2.id
ORDER BY v1.id
        ,v2.job;
    
```

```

ANSWER
=====
ID NAME      ID JOB
-- -
20 Pernal    20 Sales
30 Marenghi  30 Clerk
30 Marenghi  30 Mgr
    
```

Figure 570, 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
        ,v2.job;
    
```

```

ANSWER
=====
ID NAME      ID JOB
-- -
20 Pernal    20 Sales
30 Marenghi  30 Clerk
30 Marenghi  30 Mgr
    
```

Figure 571, Inner Join SQL (2 of 2)

ON and WHERE Usage

In an inner join only, an ON and a WHERE check work much the same way. Both define the nature of the join, and because in an inner join, only matching rows are returned, both act to exclude all rows that do not match the join.

Below is an inner join that uses an ON check to exclude managers:

```

SELECT *
FROM   staff_v1 v1
INNER JOIN
       staff_v2 v2
ON     v1.id = v2.id
AND    v2.job <> 'Mgr'
ORDER BY v1.id
        ,v2.job;
    
```

```

ANSWER
=====
ID NAME      ID JOB
-- -
20 Pernal    20 Sales
30 Marenghi  30 Clerk
    
```

Figure 572, Inner join, using ON check

Here is the same query written using a WHERE check

```

SELECT *
FROM   staff_v1 v1
INNER JOIN
       staff_v2 v2
ON     v1.id = v2.id
WHERE  v2.job <> 'Mgr'
ORDER BY v1.id
        ,v2.job;
    
```

```

ANSWER
=====
ID NAME      ID JOB
-- -
20 Pernal    20 Sales
30 Marenghi  30 Clerk
    
```

Figure 573, Inner join, using WHERE check

Left Outer Join

A left outer join is the same as saying that I want all of the rows in the first table listed, plus any matching rows in the second table:

<pre> STAFF_V1 +-----+ ID NAME +-----+ 10 Sanders 20 Pernal 30 Marenghi +-----+ </pre>	<pre> STAFF_V2 +-----+ ID JOB +-----+ 20 Sales 30 Clerk 30 Mgr 40 Sales 50 Mgr +-----+ </pre>	<pre> =====> </pre>	<pre> LEFT-OUTER-JOIN ANSWER ===== ID NAME ID JOB -- - 10 Sanders - 20 Pernal 20 Sales 30 Marenghi 30 Clerk 30 Marenghi 30 Mgr </pre>
--	---	----------------------------	--

Figure 574, Example of Left Outer Join

```

SELECT *
FROM   staff_v1 v1
LEFT OUTER JOIN
      staff_v2 v2
ON     v1.id = v2.id
ORDER BY 1,4;

```

Figure 575, Left Outer Join SQL (1 of 2)

It is possible to code a left outer join using the standard inner join syntax (with commas between tables), but it is a lot of work:

```

SELECT v1.*                                <== This join gets all
      ,v2.*                                rows in STAFF_V1
FROM   staff_v1 v1                        that match rows
      ,staff_v2 v2                        in STAFF_V2.
WHERE  v1.id = v2.id
UNION
SELECT v1.*                                <== This query gets
      ,CAST(NULL AS SMALLINT) AS id        all the rows in
      ,CAST(NULL AS CHAR(5)) AS job       STAFF_V1 with no
FROM   staff_v1 v1                        matching rows
WHERE  v1.id NOT IN                       in STAFF_V2.
      (SELECT id FROM staff_v2)
ORDER BY 1,4;

```

Figure 576, Left Outer Join SQL (2 of 2)

ON and WHERE Usage

In any type of join, a WHERE check works as if the join is an inner join. If no row matches, then no row is returned, regardless of what table the predicate refers to. By contrast, in a left or right outer join, an ON check works differently, depending on what table field it refers to:

- If it refers to a field in the table being joined to, it determines whether the related row matches the join or not.
- If it refers to a field in the table being joined from, it determines whether the related row finds a match or not. Regardless, the row will be returned.

In the next example, those rows in the table being joined to (i.e. the V2 view) that match on ID, and that are not for a manager are joined to:

```

SELECT *                                ANSWER
FROM   staff_v1 v1                       =====
LEFT OUTER JOIN                          ID NAME      ID JOB
      staff_v2 v2                        -- ----
ON     v1.id = v2.id                      10 Sanders  -  -
AND    v2.job <> 'Mgr'                    20 Pernal   20 Sales
ORDER BY v1.id                            30 Marenghi 30 Clerk
      ,v2.job;

```

Figure 577, ON check on table being joined to

If we rewrite the above query using a WHERE check we will lose a row (of output) because the check is applied after the join is done, and a null JOB does not match:

```

SELECT *                                ANSWER
FROM   staff_v1 v1                       =====
LEFT OUTER JOIN                          ID NAME      ID JOB
      staff_v2 v2                        -- ----
ON     v1.id = v2.id                      20 Pernal   20 Sales
WHERE  v2.job <> 'Mgr'                    30 Marenghi 30 Clerk
ORDER BY v1.id
      ,v2.job;

```

Figure 578, WHERE check on table being joined to (1 of 2)

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


```

SELECT      *
FROM        staff_v1 v1
LEFT OUTER JOIN
            staff_v2 v2
ON          v1.id = v2.id
WHERE      (v2.job <> 'Mgr'
            OR v2.job IS NULL)
ORDER BY  v1.id
            ,v2.job;
    
```

ANSWER			
ID	NAME	ID	JOB
10	Sanders	-	-
20	Pernal	20	Sales
30	Marenghi	30	Clerk

Figure 579, WHERE check on table being joined to (2 of 2)

In the next example, those rows in the table being joined from (i.e. the V1 view) that match on ID and have a NAME > 'N' participate in the join. Note however that V1 rows that do not participate in the join (i.e. ID = 30) are still returned:

```

SELECT      *
FROM        staff_v1 v1
LEFT OUTER JOIN
            staff_v2 v2
ON          v1.id = v2.id
AND        v1.name > 'N'
ORDER BY  v1.id
            ,v2.job;
    
```

ANSWER			
ID	NAME	ID	JOB
10	Sanders	-	-
20	Pernal	20	Sales
30	Marenghi	-	-

Figure 580, ON check on table being joined from

If we rewrite the above query using a WHERE check (on NAME) we will lose a row because now the check excludes rows from the answer-set, rather than from participating in the join:

```

SELECT      *
FROM        staff_v1 v1
LEFT OUTER JOIN
            staff_v2 v2
ON          v1.id = v2.id
WHERE      v1.name > 'N'
ORDER BY  v1.id
            ,v2.job;
    
```

ANSWER			
ID	NAME	ID	JOB
10	Sanders	-	-
20	Pernal	20	Sales

Figure 581, WHERE check on table being joined from

Unlike in the previous example, there is no way to alter the above WHERE check to make it logically equivalent to the prior ON check. The ON and the WHERE are applied at different times and for different purposes, and thus do completely different things.

Right Outer Join

A right outer join is the inverse of a left outer join. One gets every row in the second table listed, plus any matching rows in the first table:

STAFF_V1		STAFF_V2		RIGHT-OUTER-JOIN ANSWER	
ID	NAME	ID	JOB	ID	JOB
10	Sanders	20	Sales	20	Sales
20	Pernal	30	Clerk	30	Clerk
30	Marenghi	30	Mgr	30	Mgr
		40	Sales	-	40 Sales
		50	Mgr	-	50 Mgr

Figure 582, 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
       ,v2.job;

```

ANSWER			
ID	NAME	ID	JOB
20	Pernal	20	Sales
30	Marengchi	30	Clerk
30	Marengchi	30	Mgr
-	-	40	Sales
-	-	50	Mgr

Figure 583, 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.*
      ,v2.*
FROM   staff_v1 v1
      ,staff_v2 v2
WHERE  v1.id = v2.id
UNION
SELECT CAST(NULL AS SMALLINT) AS id
      ,CAST(NULL AS VARCHAR(9)) AS name
      ,v2.*
FROM   staff_v2 v2
WHERE  v2.id NOT IN
      (SELECT id FROM staff_v1)
ORDER BY 3,4;

```

ANSWER			
ID	NAME	ID	JOB
20	Pernal	20	Sales
30	Marengchi	30	Clerk
30	Marengchi	30	Mgr
-	-	40	Sales
-	-	50	Mgr

Figure 584, Right Outer Join SQL (2 of 2)

ON and WHERE Usage

The rules for ON and WHERE usage are the same in a right outer join as they are for a left outer join (see page 208), except that the relevant tables are reversed.

Full Outer Joins

A full outer join occurs when all of the matching rows in two tables are joined, and there is also returned one copy of each non-matching row in both tables.

STAFF_V1		STAFF_V2		FULL-OUTER-JOIN ANSWER			
ID	NAME	ID	JOB	ID	NAME	ID	JOB
10	Sanders	20	Sales	10	Sanders	-	-
20	Pernal	30	Clerk	20	Pernal	20	Sales
30	Marengchi	30	Mgr	30	Marengchi	30	Clerk
		40	Sales	30	Marengchi	30	Mgr
		50	Mgr	-	-	40	Sales
				-	-	50	Mgr

Figure 585, 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
       ,v2.job;

```

ANSWER			
ID	NAME	ID	JOB
10	Sanders	-	-
20	Pernal	20	Sales
30	Marengchi	30	Clerk
30	Marengchi	30	Mgr
-	-	40	Sales
-	-	50	Mgr

Figure 586, 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 1,3,4;

```

```

ANSWER
=====
ID NAME      ID JOB
-- - - - - -
10 Sanders   - -
20 Pernal    20 Sales
30 Marenghi  30 Clerk
30 Marenghi  30 Mgr
- -          40 Sales
- -          50 Mgr

```

Figure 587, Full Outer Join SQL

The above is reasonably hard to understand when two tables are involved, and it goes down hill fast as more tables are joined. Avoid.

ON and WHERE Usage

In a full outer join, an ON check is quite unlike a WHERE check in that it never results in a row being excluded from the answer set. All it does is categorize the input row as being either matching or non-matching. For example, in the following full outer join, the ON check joins those rows with equal key values:

```

SELECT    *
FROM      staff_v1 v1
FULL OUTER JOIN
          staff_v2 v2
ON        v1.id = v2.id
ORDER BY  v1.id
          ,v2.id
          ,v2.job;

```

```

ANSWER
=====
ID NAME      ID JOB
-- - - - - -
10 Sanders   - -
20 Pernal    20 Sales
30 Marenghi  30 Clerk
30 Marenghi  30 Mgr
- -          40 Sales
- -          50 Mgr

```

Figure 588, 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    *
FROM      staff_v1 v1
FULL OUTER JOIN
          staff_v2 v2
ON        v1.id = v2.id
AND       v1.id > 20
ORDER BY  v1.id
          ,v2.id
          ,v2.job;

```

```

ANSWER
=====
ID NAME      ID JOB
-- - - - - -
10 Sanders   - -
20 Pernal    - -
30 Marenghi  30 Clerk
30 Marenghi  30 Mgr
- -          20 Sales
- -          40 Sales
- -          50 Mgr

```

Figure 589, Full Outer Join, match on keys > 20

Observe how in the above statement we added a predicate, and we got more rows! This is because in an outer join an ON predicate never removes rows. It simply categorizes them as being either matching or non-matching. If they match, it joins them. If they don't, it passes them through.

In the next example, nothing matches. Consequently, every row is returned individually. This query is logically similar to doing a UNION ALL on the two views:

```

SELECT *
FROM   staff_v1 v1
FULL OUTER JOIN
      staff_v2 v2
ON     v1.id = v2.id
AND    +1 = -1
ORDER BY v1.id
        ,v2.id
        ,v2.job;

```

ANSWER			
ID	NAME	ID	JOB
10	Sanders	-	-
20	Pernal	-	-
30	Marenghi	-	-
-	-	20	Sales
-	-	30	Clerk
-	-	30	Mgr
-	-	40	Sales
-	-	50	Mgr

Figure 590, Full Outer Join, match on keys (no rows match)

ON checks are somewhat like WHERE checks in that they have two purposes. Within a table, they are used to categorize rows as being either matching or non-matching. Between tables, they are used to define the fields that are to be joined on.

In the prior example, the first ON check defined the fields to join on, while the second join identified those fields that matched the join. Because nothing matched (due to the second predicate), everything fell into the "outer join" category. This means that we can remove the first ON check without altering the answer set:

```

SELECT *
FROM   staff_v1 v1
FULL OUTER JOIN
      staff_v2 v2
ON     +1 = -1
ORDER BY v1.id
        ,v2.id
        ,v2.job;

```

ANSWER			
ID	NAME	ID	JOB
10	Sanders	-	-
20	Pernal	-	-
30	Marenghi	-	-
-	-	20	Sales
-	-	30	Clerk
-	-	30	Mgr
-	-	40	Sales
-	-	50	Mgr

Figure 591, 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 is a Cartesian Product:

```

SELECT *
FROM   staff_v1 v1
FULL OUTER JOIN
      staff_v2 v2
ON     +1 <> -1
ORDER BY v1.id
        ,v2.id
        ,v2.job;

```

ANSWER			
ID	NAME	ID	JOB
10	Sanders	20	Sales
10	Sanders	30	Clerk
10	Sanders	30	Mgr
10	Sanders	40	Sales
10	Sanders	50	Mgr
20	Pernal	20	Sales
20	Pernal	30	Clerk
20	Pernal	30	Mgr
20	Pernal	40	Sales
20	Pernal	50	Mgr
30	Marenghi	20	Sales
30	Marenghi	30	Clerk
30	Marenghi	30	Mgr
30	Marenghi	40	Sales
30	Marenghi	50	Mgr

STAFF_V1		STAFF_V2	
ID	NAME	ID	JOB
10	Sanders	20	Sales
20	Pernal	30	Clerk
30	Marenghi	30	Mgr
		40	Sales
		50	Mgr

Figure 592, Full Outer Join, don't match on keys (all rows match)

In an outer join, WHERE predicates behave as if they were written for an inner join. In particular, they always do the following:

- WHERE predicates defining join fields enforce an inner join on those fields.
- WHERE predicates on non-join fields are applied after the join, which means that when they are used on not-null fields, they negate the outer join.

Here is an example of a WHERE join predicate turning an outer join into an inner join:

SELECT *	ANSWER
FROM staff_v1 v1	=====
FULL JOIN	ID NAME ID JOB
staff_v2 v2	-- -- --
ON v1.id = v2.id	20 Pernal 20 Sales
WHERE v1.id = v2.id	30 Marenghi 30 Clerk
ORDER BY 1,3,4;	30 Marenghi 30 Mgr

Figure 593, Full Outer Join, turned into an inner join by WHERE

To illustrate some of the complications that WHERE checks can cause, imagine that we want to do a FULL OUTER JOIN on our two test views (see below), limiting the answer to those rows where the "V1 ID" field is less than 30. There are several ways to express this query, each giving a different answer:

STAFF_V1	STAFF_V2	OUTER-JOIN CRITERIA	ANSWER
+-----+	+-----+	=====	=====
ID NAME	ID JOB	=====>	???, DEPENDS
10 Sanders	20 Sales	V1.ID = V2.ID	
20 Pernal	30 Clerk	V1.ID < 30	
30 Marenghi	30 Mgr		
+-----+	+-----+		
	40 Sales		
	50 Mgr		
	+-----+		

Figure 594, Outer join V1.ID < 30, sample data

In our first example, the "V1.ID < 30" predicate is applied after the join, which effectively eliminates all "V2" rows that don't match (because their "V1.ID" value is null):

SELECT *	ANSWER
FROM staff_v1 v1	=====
FULL JOIN	ID NAME ID JOB
staff_v2 v2	-- -- --
ON v1.id = v2.id	10 Sanders - -
WHERE v1.id < 30	20 Pernal 20 Sales
ORDER BY 1,3,4;	

Figure 595, Outer join V1.ID < 30, check applied in WHERE (after join)

In the next example the "V1.ID < 30" check is done during the outer join where it does not any eliminate rows, but rather limits those that match in the two views:

SELECT *	ANSWER
FROM staff_v1 v1	=====
FULL JOIN	ID NAME ID JOB
staff_v2 v2	-- -- --
ON v1.id = v2.id	10 Sanders - -
AND v1.id < 30	20 Pernal 20 Sales
ORDER BY 1,3,4;	30 Marenghi - -
	- - 30 Clerk
	- - 30 Mgr
	- - 40 Sales
	- - 50 Mgr

Figure 596, 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 *
FROM (SELECT *
      FROM staff_v1
      WHERE id < 30) AS v1
FULL OUTER JOIN
      staff_v2 v2
ON      v1.id = v2.id
ORDER BY 1,3,4;

```

ANSWER			
=====			
ID	NAME	ID	JOB
-- -- -- -- --			
10	Sanders	-	-
20	Pernal	20	Sales
-	-	30	Clerk
-	-	30	Mgr
-	-	40	Sales
-	-	50	Mgr

Figure 597, 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 *
FROM staff_v1 v1
FULL OUTER JOIN
      staff_v2 v2
ON      v1.id = v2.id
WHERE   v1.id < 30
      OR v1.id IS NULL
ORDER BY 1,3,4;

```

ANSWER			
=====			
ID	NAME	ID	JOB
-- -- -- -- --			
10	Sanders	-	-
20	Pernal	20	Sales
-	-	40	Sales
-	-	50	Mgr

Figure 598, 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 *
FROM staff_v1 v1
FULL OUTER JOIN
      staff_v2 v2
ON      v1.id = v2.id
WHERE   v1.id < 30
      OR v1.id = v2.id
      OR v1.id IS NULL
ORDER BY 1,3,4;

```

ANSWER			
=====			
ID	NAME	ID	JOB
-- -- -- -- --			
10	Sanders	-	-
20	Pernal	20	Sales
30	Marenghi	30	Clerk
30	Marenghi	30	Mgr
-	-	40	Sales
-	-	50	Mgr

Figure 599, Outer join V1.ID < 30, (gives wrong answer - see text)

The last two checks in the above query ensure that every V2 row is returned. But they also have the affect of returning the NAME field from the V1 table whenever there is a match. Given our intentions, this should not happen.

SUMMARY: Query WHERE conditions are applied after the join. When used in an outer join, this means that they applied to all rows from all tables. In effect, this means that any WHERE conditions in a full outer join will, in most cases, turn it into a form of inner join.

Cartesian Product

A Cartesian Product is a form of inner join, where the join predicates either do not exist, or where they do a poor job of matching the keys in the joined tables.

STAFF_V1		STAFF_V2			CARTESIAN-PRODUCT			
ID	NAME	ID	JOB		ID	NAME	ID	JOB
10	Sanders	20	Sales	=====>	10	Sanders	20	Sales
20	Pernal	30	Clerk		10	Sanders	30	Clerk
30	Marenghi	30	Mgr		10	Sanders	30	Mgr
		40	Sales		10	Sanders	40	Sales
		50	Mgr		10	Sanders	50	Mgr
					20	Pernal	20	Sales
					20	Pernal	30	Clerk
					20	Pernal	30	Mgr
					20	Pernal	40	Sales
					20	Pernal	50	Mgr
					30	Marenghi	20	Sales
					30	Marenghi	30	Clerk
					30	Marenghi	30	Mgr
					30	Marenghi	40	Sales
					30	Marenghi	50	Mgr

Figure 600, Example of Cartesian Product

Writing a Cartesian Product is simplicity itself. One simply omits the WHERE conditions:

```
SELECT *
FROM   staff_v1 v1
       ,staff_v2 v2
ORDER BY v1.id
        ,v2.id
        ,v2.job;
```

Figure 601, Cartesian Product SQL (1 of 2)

One way to reduce the likelihood of writing a full Cartesian Product is to always use the inner/outer join style. With this syntax, an ON predicate is always required. There is however no guarantee that the ON will do any good. Witness the following example:

```
SELECT *
FROM   staff_v1 v1
INNER JOIN
       staff_v2 v2
ON     'A' <> 'B'
ORDER BY v1.id
        ,v2.id
        ,v2.job;
```

Figure 602, 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				ANSWER
				=====
				ID JOB ID
				-- -- --
				20 Sales 20
				20 Sales 40
				30 Clerk 30
				30 Mgr 30
				30 Mgr 50

Figure 603, Partial Cartesian Product SQL

In the above example we joined the two views by JOB, which is not a unique key. The result was that for each JOB value, we got a mini Cartesian Product.

Cartesian Products are at their most insidious when the result of the (invalid) join is feed into a GROUP BY or DISTINCT statement that removes all of the duplicate rows. Below is an example where the only clue that things are wrong is that the count is incorrect:

```

SELECT    v2.job                                ANSWER
          ,COUNT(*) AS #rows                    =====
FROM      staff_v1 v1                           JOB    #ROWS
          ,staff_v2 v2                           -----
GROUP BY  v2.job
ORDER BY  #rows
          ,v2.job;

```

JOB	#ROWS
Clerk	3
Mgr	6
Sales	6

Figure 604, Partial Cartesian Product SQL, with GROUP BY

To really mess up with a Cartesian Product you may have to join more than one table. Note however that big tables are not required. For example, a Cartesian Product of five 100-row tables will result in 10,000,000,000 rows being returned.

HINT: A good rule of thumb to use when writing a join is that for all of the tables (except one) there should be equal conditions on all of the fields that make up the various unique keys. If this is not true then it is probable that some kind Cartesian Product is being done and the answer may be wrong.

Join Notes

Using the COALESCE Function

If you don't like working with nulls, but you need to do outer joins, then life is tough. In an outer join, fields in non-matching rows are given null values as placeholders. Fortunately, these nulls can be eliminated using the COALESCE function.

The COALESCE function can be used to combine multiple fields into one, and/or to eliminate null values where they occur. The result of the COALESCE is always the first non-null value encountered. In the following example, the two ID fields are combined, and any null NAME values are replaced with a question mark.

```

SELECT    COALESCE(v1.id,v2.id) AS id            ANSWER
          ,COALESCE(v1.name, '?') AS name        =====
FROM      staff_v1 v1                           ID NAME    JOB
          ,staff_v2 v2                           -- -----
FULL OUTER JOIN
          staff_v2 v2
ON        v1.id = v2.id
ORDER BY  v1.id
          ,v2.job;

```

ID	NAME	JOB
10	Sanders	-
20	Pernal	Sales
30	Marengi	Clerk
30	Marengi	Mgr
40	?	Sales
50	?	Mgr

Figure 605, Use of COALESCE function in outer join

Listing non-matching rows only

Imagine that we wanted to do an outer join on our two test views, only getting those rows that do not match. This is a surprisingly hard query to write.

STAFF_V1		STAFF_V2		NON-MATCHING OUTER-JOIN =====>	ANSWER =====
ID	NAME	ID	JOB		ID NAME ID JOB
10	Sanders	20	Sales		10 Sanders - -
20	Pernal	30	Clerk		- - 40 Sales
30	Marenghi	30	Mgr		- - 50 Mgr
		40	Sales		
		50	Mgr		

Figure 606, Example of outer join, only getting the non-matching rows

One way to express the above is to use the standard inner-join syntax:

```

SELECT  v1.*                                <== Get all the rows
        ,CAST(NULL AS SMALLINT) AS id        in STAFF_V1 that
        ,CAST(NULL AS CHAR(5)) AS job       have no matching
FROM    staff_v1 v1                          row in STAFF_V2.
WHERE   v1.id NOT IN
        (SELECT id FROM staff_v2)

UNION
SELECT  CAST(NULL AS SMALLINT) AS id        <== Get all the rows
        ,CAST(NULL AS VARCHAR(9)) AS name   in STAFF_V2 that
        ,v2.*                               have no matching
FROM    staff_v2 v2                          row in STAFF_V1.
WHERE   v2.id NOT IN
        (SELECT id FROM staff_v1)

ORDER BY 1,3,4;

```

Figure 607, 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
        OR v2.flag IS NULL
ORDER BY v1.id
        ,v2.id
        ,v2.job;

```

ANSWER =====					
ID	NAME	FLAG	ID	JOB	FLAG
10	Sanders	V1	-	-	-
-	-	-	40	Sales	V2
-	-	-	50	Mgr	V2

Figure 608, 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
      v2 v2
ON    v1.id = v2.id
WHERE v1.flag IS NULL
      OR v2.flag IS NULL
ORDER BY v1.id, v2.id, v2.job;

```

ANSWER =====					
ID	NAME	FLAG	ID	JOB	FLAG
10	Sanders	V1	-	-	-
-	-	-	40	Sales	V2
-	-	-	50	Mgr	V2

Figure 609, 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 *
FROM   staff_v1 v1
FULL OUTER JOIN
      staff_v2 v2
ON     v1.id = v2.id
WHERE  v1.id IS NULL
      OR v2.id IS NULL
ORDER BY v1.id
        ,v2.id
        ,v2.job;

```

STAFF_V1	
ID	NAME
10	Sanders
20	Pernal
30	Marenghi

STAFF_V2	
ID	JOB
20	Sales
30	Clerk
30	Mgr
40	Sales
50	Mgr

Figure 610, Outer Join SQL, getting only non-matching rows

Join in SELECT Phrase

Imagine that we want to get selected rows from the V1 view, and for each matching row, get the corresponding JOB from the V2 view - if there is one:

STAFF_V1	
ID	NAME
10	Sanders
20	Pernal
30	Marenghi

STAFF_V2	
ID	JOB
20	Sales
30	Clerk
30	Mgr
40	Sales
50	Mgr

```

LEFT OUTER JOIN
=====>
V1.ID = V2.ID
V1.ID <> 30

```

ANSWER			
ID	NAME	ID	JOB
10	Sanders	-	-
20	Pernal	20	Sales

Figure 611, Left outer join example

Here is one way to express the above as a query:

```

SELECT   v1.id
        ,v1.name
        ,v2.job
FROM     staff_v1 v1
LEFT OUTER JOIN
      staff_v2 v2
ON       v1.id = v2.id
WHERE    v1.id <> 30
ORDER BY v1.id ;

```

ANSWER			
ID	NAME	ID	JOB
10	Sanders	-	-
20	Pernal	20	Sales

Figure 612, 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
        ,v1.name
        ,(SELECT v2.job
          FROM   staff_v2 v2
          WHERE  v1.id = v2.id) AS jb
FROM     staff_v1 v1
WHERE    v1.id <> 30
ORDER BY v1.id;

```

ANSWER			
ID	NAME	JOB	
10	Sanders	-	
20	Pernal	Sales	

Figure 613, Outer Join done in SELECT phrase of SQL

Certain rules apply when using the above syntax:

- The nested table expression in the SELECT is applied after all other joins and sub-queries (i.e. in the FROM section of the query) are done.
- The nested table expression acts as a left outer join.
- Only one column and row (at most) can be returned by the expression.

- If no row is returned, the result is null.

Given the above restrictions, the following query will fail because more than one V2 row is returned for every V1 row (for ID = 30):

```

SELECT    v1.id
          ,v1.name
          ,(SELECT  v2.job
             FROM    staff_v2 v2
             WHERE   v1.id = v2.id) AS jb
FROM      staff_v1 v1
ORDER BY v1.id;

```

	ANSWER
	=====
	ID NAME JB
	-- -----
	10 Sanders -
	20 Pernal Sales
	<error>

Figure 614, Outer Join done in SELECT phrase of SQL - gets error

To make the above query work for all IDs, we have to decide which of the two matching JOB values for ID 30 we want. Let us assume that we want the maximum:

```

SELECT    v1.id
          ,v1.name
          ,(SELECT  MAX(v2.job)
             FROM    staff_v2 v2
             WHERE   v1.id = v2.id) AS jb
FROM      staff_v1 v1
ORDER BY v1.id;

```

	ANSWER
	=====
	ID NAME JB
	-- -----
	10 Sanders -
	20 Pernal Sales
	30 Marenghi Mgr

Figure 615, Outer Join done in SELECT phrase of SQL - fixed

The above is equivalent to the following query:

```

SELECT    v1.id
          ,v1.name
          ,MAX(v2.job) AS jb
FROM      staff_v1 v1
LEFT OUTER JOIN
          staff_v2 v2
ON        v1.id = v2.id
GROUP BY v1.id
          ,v1.name
ORDER BY v1.id ;

```

	ANSWER
	=====
	ID NAME JB
	-- -----
	10 Sanders -
	20 Pernal Sales
	30 Marenghi Mgr

Figure 616, Same as prior query - using join and GROUP BY

The above query is rather misleading because someone unfamiliar with the data may not understand why the NAME field is in the GROUP BY. Obviously, it is not there to remove any rows, it simply needs to be there because of the presence of the MAX function. Therefore, the preceding query is better because it is much easier to understand. It is also probably more efficient.

CASE Usage

The SELECT expression can be placed in a CASE statement if needed. To illustrate, in the following query we get the JOB from the V2 view, except when the person is a manager, in which case we get the NAME from the corresponding row in the V1 view:

```

SELECT    v2.id
          ,CASE
            WHEN v2.job <> 'Mgr'
            THEN v2.job
            ELSE (SELECT v1.name
                  FROM  staff_v1 v1
                  WHERE v1.id = v2.id)
          END AS j2
FROM      staff_v2 v2
ORDER BY v2.id
          ,j2;

```

	ANSWER
	=====
	ID J2
	-- -----
	20 Sales
	30 Clerk
	30 Marenghi
	40 Sales
	50 -

Figure 617, Sample Views used in Join Examples

Multiple Columns

If you want to retrieve two columns using this type of join, you need to have two independent nested table expressions:

SELECT	v2.id	ANSWER	
	,v2.job	=====	
	,(SELECT v1.name	ID JOB NAME N2	
	FROM staff_v1 v1	-- -- -- -- --	
	WHERE v2.id = v1.id)	20 Sales Pernal 6	
	,(SELECT LENGTH(v1.name) AS n2	30 Clerk Marenghi 8	
	FROM staff_v1 v1	30 Mgr Marenghi 8	
	WHERE v2.id = v1.id)	40 Sales - -	
FROM	staff_v2 v2	50 Mgr - -	
ORDER BY	v2.id		
	,v2.job;		

Figure 618, Outer Join done in SELECT, 2 columns

An easier way to do the above is to write an ordinary left outer join with the joined columns in the SELECT list. To illustrate this, the next query is logically equivalent to the prior:

SELECT	v2.id	ANSWER	
	,v2.job	=====	
	,v1.name	ID JOB NAME N2	
	,LENGTH(v1.name) AS n2	-- -- -- -- --	
FROM	staff_v2 v2	20 Sales Pernal 6	
LEFT OUTER JOIN		30 Clerk Marenghi 8	
	staff_v1 v1	30 Mgr Marenghi 8	
ON	v2.id = v1.id	40 Sales - -	
ORDER BY	v2.id	50 Mgr - -	
	,v2.job;		

Figure 619, Outer Join done in FROM, 2 columns

Column Functions

This join style lets one easily mix and match individual rows with the results of column functions. For example, the following query returns a running SUM of the ID column:

SELECT	v1.id	ANSWER	
	,v1.name	=====	
	,(SELECT SUM(x1.id)	ID NAME SUM_ID	
	FROM staff_v1 x1	-- -- -- -- --	
	WHERE x1.id <= v1.id	10 Sanders 10	
)AS sum_id	20 Pernal 30	
FROM	staff_v1 v1	30 Marenghi 60	
ORDER BY	v1.id		
	,v2.job;		

Figure 620, 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 621, 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		OUTER-JOIN CRITERIA	ANSWER		
ID	NAME	ID	JOB		ID	NAME	JOB
10	Sanders	20	Sales	V1.ID = V2.ID	10	Sanders	-
20	Pernal	30	Clerk	V2.JOB LIKE 'S%'	20	Pernal	Sales
30	Marenghi	30	Mgr		30	Marenghi	-
		40	Sales				
		50	Mgr				

Figure 622, 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				ANSWER (WRONG)		
	v1.id	v1.name	v2.job	ID	NAME	JOB
FROM	staff_v1	v1		20	Pernal	Sales
LEFT OUTER JOIN	staff_v2	v2				
ON	v1.id	=	v2.id			
WHERE	v2.job	LIKE	'S%'			
ORDER BY	v1.id					
	v2.job;					

Figure 623, 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				ANSWER		
	v1.id	v1.name	v2.job	ID	NAME	JOB
FROM	staff_v1	v1		10	Sanders	-
LEFT OUTER JOIN	(SELECT *			20	Pernal	Sales
	FROM staff_v2	v2		30	Marenghi	-
	WHERE job	LIKE	'S%'			
)AS v2					
ON	v1.id	=	v2.id			
ORDER BY	v1.id					
	v2.job;					

Figure 624, 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				ANSWER		
	v1.id	v1.name	(SELECT v2.job	ID	NAME	JOB
			FROM staff_v2	10	Sanders	-
			WHERE v1.id	20	Pernal	Sales
			AND v2.job	30	Marenghi	-
			LIKE 'S%')			
FROM	staff_v1	v1				
ORDER BY	v1.id					
	job;					

Figure 625, Outer Join, WHERE done independently - correct

Joins - Things to Remember

- You get nulls in an outer join, whether you want them or not, because the fields in non-matching rows are set to null. If they bug you, use the COALESCE function to remove them. See page 216 for an example.

- From a logical perspective, all WHERE conditions are applied after the join. For performance reasons, DB2 may apply some checks before the join, especially in an inner join, where doing this cannot affect the result set.
- All WHERE conditions that join tables act as if they are doing an inner join, even when they are written in an outer join.
- The ON checks in a full outer join never remove rows. They simply determine what rows are matching versus not (see page 211). To eliminate rows in an outer join, one must use a WHERE condition.
- The ON checks in a partial outer join work differently, depending on whether they are against fields in the table being joined to, or joined from (see page 208).
- A Cartesian Product is not an outer join. It is a poorly matching inner join. By contrast, a true outer join gets both matching rows, and non-matching rows.
- The NODENUMBER and PARTITION functions cannot be used in an outer join. These functions only work on rows in real tables.

When the join is defined in the SELECT part of the query (see page 218), it is done after any other joins and/or sub-queries specified in the FROM phrase. And it acts as if it is a left outer join.

Complex Joins

When one joins multiple tables using an outer join, one must consider carefully what exactly what one wants to do, because the answer that one gets will depend upon how one writes the query. To illustrate, the following query first gets a set of rows from the employee table, and then joins (from the employee table) to both the activity and photo tables:

SELECT	eee.empno		ANSWER	
	,aaa.projno		=====	
	,aaa.actno		EMPNO PROJNO ACTNO FORMAT	
	,ppp.photo_format AS format		-----	
FROM	employee eee		000010 MA2110 10 -	
LEFT OUTER JOIN	emp_act aaa		000070 - - -	
ON	eee.empno = aaa.empno		000130 - - bitmap	
AND	aaa.emptime = 1		000150 MA2112 60 bitmap	
AND	aaa.projno LIKE 'M%1%'		000150 MA2112 180 bitmap	
LEFT OUTER JOIN	emp_photo ppp		000160 MA2113 60 -	
ON	eee.empno = ppp.empno	←		
AND	ppp.photo_format LIKE 'b%'			
WHERE	eee.lastname LIKE '%A%'			
AND	eee.empno < '000170'			
AND	eee.empno <> '000030'			
ORDER BY	eee.empno;			

Figure 626, Join from Employee to Activity and Photo

Observe that we got photo data, even when there was no activity data. This is because both tables were joined directly from the employee table. In the next query, we will again start at the employee table, then join to the activity table, and then from the activity table join to the photo table. We will not get any photo data, if the employee has no activity:

```

SELECT      eee.empno
            ,aaa.projno
            ,aaa.actno
            ,ppp.photo_format AS format
FROM        employee   eee
LEFT OUTER JOIN
emp_act    aaa
ON          eee.empno      = aaa.empno
AND        aaa.emptime     = 1
AND        aaa.projno     LIKE 'M%1%'
LEFT OUTER JOIN
emp_photo  ppp
ON          aaa.empno      = ppp.empno
AND        ppp.photo_format LIKE 'b%'
WHERE      eee.lastname    LIKE '%A%'
          AND eee.empno    < '000170'
          AND eee.empno    <> '000030'
ORDER BY   eee.empno;

```

ANSWER			
EMPNO	PROJNO	ACTNO	FORMAT
000010	MA2110	10	-
000070	-	-	-
000130	-	-	-
000150	MA2112	60	bitmap
000150	MA2112	180	bitmap
000160	MA2113	60	-

Figure 627, Join from Employee to Activity, then from Activity to Photo

The only difference between the above two queries is the first line of the second ON.

Outer Join followed by Inner Join

Mixing and matching inner and outer joins in the same query can cause one to get the wrong answer. To illustrate, the next query has an outer join, followed by an inner join. We are trying to do the following:

- Get a list of matching employees - based on some local predicates.
- For each employee found, list their matching activities, if any (i.e. left outer join).
- For each activity found, only list it if its project-name contains the letter "Q" (i.e. inner join between activity and project).

Below is the **wrong** way to write this query. It is wrong because the final inner join (between activity and project) turns the preceding outer join into an inner join. This causes an employee to not show when there are no matching projects:

```

SELECT      eee.workdept AS dp#
            ,eee.empno
            ,aaa.projno
            ,ppp.prstaff AS staff
FROM        (SELECT *
            FROM employee
            WHERE lastname LIKE '%A%'
              AND job       <> 'DESIGNER'
              AND workdept BETWEEN 'B' AND 'E'
            )AS eee
LEFT OUTER JOIN
emp_act    aaa
ON          aaa.empno      = eee.empno
AND        aaa.emptime     <= 0.5
INNER JOIN
project    ppp
ON          aaa.projno     = ppp.projno
AND        ppp.projname   LIKE '%Q%'
ORDER BY   eee.workdept
            ,eee.empno
            ,aaa.projno;

```

ANSWER			
DP#	EMPNO	PROJNO	STAFF
C01	000030	IF1000	2.00
C01	000130	IF1000	2.00

Figure 628, Complex join - wrong

As was stated above, we really want to get all matching employees, and their related activities (projects). If an employee has no matching activities, we still want to see the employee.

The next query gets the correct answer by putting the inner join between the activity and project tables in parenthesis, and then doing an outer join to the combined result:

```

SELECT    eee.workdept AS dp#
,eee.empno
,xxx.projno
,xxx.prstaff AS staff
FROM      (SELECT *
          FROM    employee
          WHERE   lastname LIKE '%A%'
                AND job    <> 'DESIGNER'
                AND workdept BETWEEN 'B' AND 'E'
          )AS eee
LEFT OUTER JOIN
  (SELECT  aaa.empno
          ,aaa.emptime
          ,aaa.projno
          ,ppp.prstaff
          FROM    emp_act    aaa
          INNER JOIN
                project    ppp
          ON      aaa.projno = ppp.projno
          AND    ppp.projname LIKE '%Q%'
          )AS xxx
ON        xxx.empno    = eee.empno
AND       xxx.emptime <= 0.5
ORDER BY  eee.workdept
,eee.empno
,xxx.projno;

```

ANSWER			
DP#	EMPNO	PROJNO	STAFF
C01	000030	IF1000	2.00
C01	000130	IF1000	2.00
D21	000070	-	-
D21	000240	-	-

Figure 629, Complex join - right

The lesson to be learnt here is that if a subsequent inner join acts upon data in a preceding outer join, then it, in effect, turns the former into an inner join.

Simplified Nested Table Expression

The next query is the same as the prior, except that the nested-table expression has no select list, nor correlation name. In this example, any columns in tables that are inside of the nested-table expression are referenced directly in the rest of the query:

```

SELECT    eee.workdept AS dp#
,eee.empno
,aaa.projno
,ppp.prstaff AS staff
FROM      (SELECT *
          FROM    employee
          WHERE   lastname LIKE '%A%'
                AND job    <> 'DESIGNER'
                AND workdept BETWEEN 'B' AND 'E'
          )AS eee
LEFT OUTER JOIN
  (
    emp_act    aaa
  INNER JOIN
    project    ppp
  ON      aaa.projno = ppp.projno
  AND    ppp.projname LIKE '%Q%'
  )
ON        aaa.empno    = eee.empno
AND       aaa.emptime <= 0.5
ORDER BY  eee.workdept
,eee.empno
,aaa.projno;

```

ANSWER			
DP#	EMPNO	PROJNO	STAFF
C01	000030	IF1000	2.00
C01	000130	IF1000	2.00
D21	000070	-	-
D21	000240	-	-

Figure 630, Complex join - right

Sub-Query

Sub-queries are hard to use, tricky to tune, and often do some strange things. Consequently, a lot of people try to avoid them, but this is stupid because sub-queries are really, really, useful. Using a relational database and not writing sub-queries is almost as bad as not doing joins.

A sub-query is a special type of fullselect that is used to relate one table to another without actually doing a join. For example, it lets one select all of the rows in one table where some related value exists, or does not exist, in another table.

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
(t1a      CHAR(1)    NOT NULL
 ,t1b      CHAR(2)    NOT NULL
 ,PRIMARY KEY(t1a));
COMMIT;
```

```
CREATE TABLE table2
(t2a      CHAR(1)    NOT NULL
 ,t2b      CHAR(1)    NOT NULL
 ,t2c      CHAR(1));
```

```
INSERT INTO table1 VALUES ('A','AA'),('B','BB'),('C','CC');
INSERT INTO table2 VALUES ('A','A','A'),('B','A',NULL);
```

TABLE1		TABLE2		
T1A	T1B	T2A	T2B	T2C
A	AA	A	A	A
B	BB	B	A	-
C	CC			

"-" = null

Figure 631, Sample tables used in sub-query examples

Sub-query Flavours

Sub-query Syntax

A sub-query compares an expression against a fullselect. The type of comparison done is a function of which, if any, keyword is used:

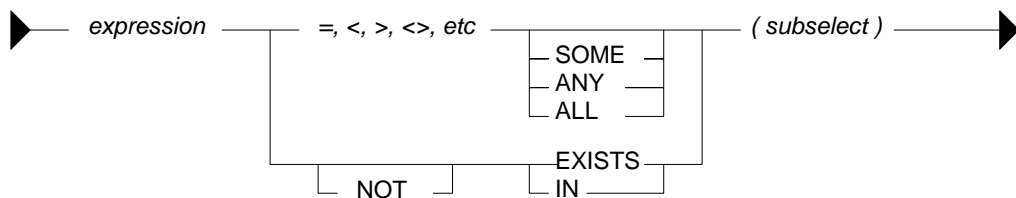


Figure 632, Sub-query syntax diagram

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 *
FROM table1
WHERE t1a =
      (SELECT t2a
       FROM table2
       WHERE t2a = 'A');

```

ANSWER
=====

	TABLE1	TABLE2
SUB-Q RESLT +----+ T2A ---- A +----+	+-----+ T1A T1B ---- ---- A AA B BB C CC +-----+	+-----+ T2A T2B T2C ---- ---- ---- A A A B A - +-----+
		"- " = null

Figure 633, No keyword sub-query, works

The next example gets a SQL error. The sub-query returns two rows, which the "=|" check cannot process. Had an "= ANY" or an "= SOME" check been used instead, the query would have worked fine:

```

SELECT *
FROM table1
WHERE t1a =
      (SELECT t2a
       FROM table2);

```

ANSWER
=====

<error>

	TABLE1	TABLE2
SUB-Q RESLT +----+ T2A ---- A B +----+	+-----+ T1A T1B ---- ---- A AA B BB C CC +-----+	+-----+ T2A T2B T2C ---- ---- ---- A A A B A - +-----+
		"- " = null

Figure 634, 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	T2A T2B T2C
FROM table2);	B BB	---	A AA	A A A
	C CC	---	B BB	B A -
		---	C CC	- - -
		---	+-----+	+-----+
				"-" = null

Figure 635, 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-query)	> MINIMUM(sub-query results)
< ANY(sub-query)	< MAXIMUM(sub-query results)
> ALL(sub-query)	> MAXIMUM(sub-query results)
< ALL(sub-query)	< MINIMUM(sub-query results)

Figure 636, 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	---	---
FROM table2	A AA	T2B
WHERE t2b >= 'A');		---
		A
		A
		+----+

Figure 637, 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	---	---
FROM table2	A AA	T2B
WHERE t2b >= 'X');	B BB	---
	C CC	+----+

Figure 638, 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 229) 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 *
FROM table1
WHERE t1a = ALL
      (SELECT t2b
       FROM table2
       WHERE t2b >= 'X' )
AND 0 <>
      (SELECT COUNT(*)
       FROM table2
       WHERE t2b >= 'X' );

```

SQ-#1	SQ-#2	TABLE1	TABLE2
RESLT	RESLT		
+-----+	+-----+	+-----+	+-----+
T2B	(*)	T1A T1B	T2A T2B T2C
-----	-----	-----	-----
0	0	A AA	A A A
-----	-----	B BB	B A -
+-----+	+-----+	C CC	+-----+
			"-" = null

Figure 639, 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 *
FROM table1
WHERE EXISTS
      (SELECT *
       FROM table2);

```

ANSWER	TABLE1	TABLE2
T1A T1B		
=====	+-----+	+-----+
A AA	T1A T1B	T2A T2B T2C
-----	-----	-----
B BB	A AA	A A A
-----	B BB	B A -
C CC	C CC	+-----+
		"-" = null

Figure 640, EXISTS sub-query, always returns a match

Below is an EXISTS check that, given our sample data, always returns false:

```

SELECT *
FROM table1
WHERE EXISTS
      (SELECT *
       FROM table2
       WHERE t2b >= 'X' );

```

ANSWER
=====
0 rows

Figure 641, 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 *
FROM   table1
WHERE  EXISTS
      (SELECT COUNT(*)
       FROM   table2
       WHERE  t2b = 'X');

```

ANSWERS		TABLE1		TABLE2		
T1A	T1B	T1A	T1B	T2A	T2B	T2C
A	AA	A	AA	A	A	A
B	BB	B	BB	B	A	-
C	CC	C	CC			

 "-" = null

Figure 642, 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 *
FROM   table1
WHERE  NOT EXISTS
      (SELECT *
       FROM   table2
       WHERE  t2c >= 'A'
              AND t2c <> t1a);

```

ANSWERS		TABLE1		TABLE2		
T1A	T1B	T1A	T1B	T2A	T2B	T2C
A	AA	A	AA	A	A	A
B	BB	B	BB	B	A	-
C	CC	C	CC			

 "-" = null

```

SELECT *
FROM   table1
WHERE  t1a = ALL
      (SELECT t2c
       FROM   table2
       WHERE  t2c >= 'A');

```

Figure 643, 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 *
FROM   table1
WHERE  NOT EXISTS
      (SELECT *
       FROM   table2
       WHERE  t2c >= 'X'
              AND t2c <> t1a);

```

ANSWERS		TABLE1		TABLE2		
T1A	T1B	T1A	T1B	T2A	T2B	T2C
A	AA	A	AA	A	A	A
B	BB	B	BB	B	A	-
C	CC	C	CC			

 "-" = null

```

SELECT *
FROM   table1
WHERE  t1a = ALL
      (SELECT t2c
       FROM   table2
       WHERE  t2c >= 'X');

```

Figure 644, 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 *
FROM table1
WHERE NOT EXISTS
  (SELECT *
   FROM table2
   WHERE t2c <> t1a);
ANSWER
=====
T1A T1B
---
A AA

```

TABLE1		TABLE2		
T1A	T1B	T2A	T2B	T2C
A	AA	A	A	A
B	BB	B	A	-
C	CC			

```

SELECT *
FROM table1
WHERE t1a = ALL
  (SELECT t2c
   FROM table2);
ANSWER
=====
no rows

```

Figure 645, 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 *
FROM table2
WHERE t2c <> 'A';
ANSWER
=====
T2A T2B T2C
---
no rows

```

```

SELECT *
FROM table2
WHERE t2c <> 'B';
ANSWER
=====
T2A T2B T2C
---
A A A

```

```

SELECT *
FROM table2
WHERE t2c <> 'C';
ANSWER
=====
T2A T2B T2C
---
A A A

```

Figure 646, List of values in T2C <> T1A value

To make a NOT EXISTS sub-query that is logically equivalent to the ALL sub-query that we have used above, one can add an additional check for null T2C values:

```

SELECT *
FROM table1
WHERE NOT EXISTS
  (SELECT *
   FROM table2
   WHERE t2c <> t1a
   OR t2c IS NULL);
ANSWER
=====
no rows

```

TABLE1		TABLE2		
T1A	T1B	T2A	T2B	T2C
A	AA	A	A	A
B	BB	B	A	-
C	CC			

Figure 647, NOT EXISTS - same as ALL

One problem with the above query is that it is not exactly obvious. Another is that the two T2C predicates will have to be fenced in with parenthesis if other predicates (on TABLE2) exist. For these reasons, use an ALL sub-query when that is what you mean to do.

IN Keyword Sub-Query

The IN sub-query check is similar to the ANY and SOME checks:

- If any row in the sub-query result matches, the answer is true.
- If the sub-query result is empty, the answer is false.
- If no row in the sub-query result matches, the answer is also false.
- If all of the values in the sub-query result are null, the answer is false.

Below is an example that compares the T1A and T2A columns. Two rows match:

<pre>SELECT * FROM table1 WHERE t1a IN (SELECT t2a FROM table2);</pre>	<pre>ANSWER ===== T1A T1B --- -- A AA B BB</pre>	<pre>TABLE1 +-----+ T1A T1B ---- ---- A AA B BB C CC +-----+</pre>	<pre>TABLE2 +-----+ T2A T2B T2C ---- ---- ---- A A A B A - +-----+ "- " = null</pre>
---	--	---	--

Figure 648, IN sub-query example, two matches

In the next example, no rows match because the sub-query result is an empty set:

<pre>SELECT * FROM table1 WHERE t1a IN (SELECT t2a FROM table2 WHERE t2a >= 'X');</pre>	<pre>ANSWER ===== 0 rows</pre>
--	--------------------------------

Figure 649, 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 return false:

<pre>SELECT * FROM table2 WHERE t2c IN (SELECT t2c FROM table2);</pre>	<pre>ANSWERS ===== T2A T2B T2C --- -- -- A A A</pre>	<pre>TABLE2 +-----+ T2A T2B T2C ---- ---- ---- A A A B A - +-----+ "- " = null</pre>
<pre>SELECT * FROM table2 WHERE t2c = ANY (SELECT t2c FROM table2);</pre>		

Figure 650, 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 is 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:

<pre>SELECT * FROM table1 WHERE t1a NOT IN (SELECT t2c FROM table2);</pre>	<pre>ANSWER ===== 0 rows</pre>	<pre>TABLE1 +-----+ T1A T1B ---- ---- A AA B BB C CC +-----+</pre>	<pre>TABLE2 +-----+ T2A T2B T2C ---- ---- ---- A A A B A - +-----+ "- " = null</pre>
---	--------------------------------	---	--

Figure 651, 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 652, 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 653, NOT EXISTS sub-query example, matches

Correlated vs. Uncorrelated Sub-Queries

An uncorrelated sub-query is one where the predicates in the sub-query part of SQL statement have no direct relationship to the current row being processed in the "top" table (hence uncorrelated). The following sub-query is uncorrelated:

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 654, 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 655, 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 *
FROM   table2 aa
WHERE  EXISTS
      (SELECT *
       FROM   table2 bb
       WHERE  aa.t2a = bb.t2b);

```

ANSWER		
T2A	T2B	T2C
A	A	A

TABLE2		
T2A	T2B	T2C
A	A	A
B	A	-

"-" = null

Figure 656, 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 *
FROM   table1
WHERE  (t1a,t1b) IN
      (SELECT t2a, t2b
       FROM   table2);

```

ANSWER		TABLE1		TABLE2		
=====		T1A	T1B	T2A	T2B	T2C
0 rows		A	AA	A	A	A
		B	BB	B	A	-
		C	CC			

"-" = null

```

SELECT *
FROM   table1
WHERE  EXISTS
      (SELECT *
       FROM   table2
       WHERE  t1a = t2a
          AND t1b = t2b);

```

ANSWER	
=====	
0 rows	

Figure 657, 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 *
FROM   table1
WHERE  EXISTS
      (SELECT *
       FROM   table2
       WHERE  t1a = t2a
          AND t1b >= t2b);

```

ANSWER		TABLE1		TABLE2		
T1A	T1B	T1A	T1B	T2A	T2B	T2C
A	AA	A	AA	A	A	A
B	BB	B	BB	B	A	-
		C	CC			

"-" = null

Figure 658, 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
       ,lastname
       ,salary
FROM   employee
WHERE  salary >
      (SELECT MAX(salary)
       FROM   employee
       WHERE  empno NOT IN
            (SELECT empno
             FROM   emp_act
             WHERE  projno LIKE 'MA%'))

ORDER BY 1;

```

```

ANSWER
=====
EMPNO  LASTNAME  SALARY
-----
000010 HAAS      52750.00
000110 LUCCHESSE 46500.00

```

Figure 659, 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
        ,MAX(deptno)  AS maxdpt
FROM     department
WHERE    deptname LIKE 'Z%'
ORDER BY 1;

```

```

ANSWER
=====
#ROWS MAXDEPT
-----
0      null

```

Figure 660, 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 *
FROM   table1 t1
WHERE  0 =
      (SELECT COUNT(*)
       FROM   table2 t2
       WHERE  t1.t1a = t2.t2c);

```

```

TABLE1      TABLE2
+-----+
| T1A | T1B | | T2A | T2B | T2C |
+-----+
| A   | AA  | | A   | A   | A   |
| B   | BB  | | B   | A   | -   |
| C   | CC  | |-----+
+-----+
"- " = null

```

```

SELECT *
FROM   table1 t1
WHERE  NOT EXISTS
      (SELECT *
       FROM   table2 t2
       WHERE  t1.t1a = t2.t2c);

```

```

ANSWER
=====
T1A T1B
----
B   BB
C   CC

```

```

SELECT *
FROM   table1
WHERE  t1a NOT IN
      (SELECT t2c
       FROM   table2
       WHERE  t2c IS NOT NULL);

```

Figure 661, 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.*
FROM table1 t1
LEFT OUTER JOIN
    table2 t2
ON t1.t1a = t2.t2c
WHERE t2.t2c IS NULL;
```

ANSWER
=====

T1A	T1B
B	BB
C	CC

Figure 662, 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 *
FROM table1 t1
WHERE EXISTS
    (SELECT *
    FROM table2 t2
    WHERE t1.t1a = t2.t2c);

SELECT *
FROM table1 t1
WHERE 1 <=
    (SELECT COUNT(*)
    FROM table2 t2
    WHERE t1.t1a = t2.t2c);

SELECT *
FROM table1
WHERE t1a = ANY
    (SELECT t2c
    FROM table2);

SELECT *
FROM table1
WHERE t1a = SOME
    (SELECT t2c
    FROM table2);

SELECT *
FROM table1
WHERE t1a IN
    (SELECT t2c
    FROM table2);
```

TABLE1		TABLE2		
T1A	T1B	T2A	T2B	T2C
A	AA	A	A	A
B	BB	B	A	-
C	CC			

ANSWER
=====

T1A	T1B
A	AA

"-" = null

Figure 663, 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
  (SELECT DISTINCT t2c
   FROM table2
  )
SELECT t1.*
FROM table1 t1
      ,t2
WHERE t1.t1a = t2.t2c;

SELECT t1.*
FROM table1 t1
      ,(SELECT DISTINCT t2c
        FROM table2
        )AS t2
WHERE t1.t1a = t2.t2c;

SELECT t1.*
FROM table1 t1
INNER JOIN
  (SELECT DISTINCT t2c
   FROM table2
  )AS t2
ON t1.t1a = t2.t2c;

```

TABLE1	
T1A	T1B
A	AA
B	BB
C	CC

TABLE2		
T2A	T2B	T2C
A	A	A
B	A	-

"-" = null

```

ANSWER
=====
T1A T1B
----
A   AA

```

Figure 664, 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 *
FROM table1 t1
WHERE 10 =
  (SELECT COUNT(*)
   FROM table2 t2
   WHERE t1.t1a = t2.t2b);

SELECT *
FROM table1
WHERE EXISTS
  (SELECT t2b
   FROM table2
   WHERE t1a = t2b
   GROUP BY t2b
   HAVING COUNT(*) = 10);

SELECT *
FROM table1
WHERE t1a IN
  (SELECT t2b
   FROM table2
   GROUP BY t2b
   HAVING COUNT(*) = 10);

```

TABLE1	
T1A	T1B
A	AA
B	BB
C	CC

TABLE2		
T2A	T2B	T2C
A	A	A
B	A	-

"-" = null

```

ANSWER
=====
0 rows

```

Figure 665, 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 233 for more notes) to both check T2B and the count at the same time:

```

SELECT *
FROM table1
WHERE (t1a,10) IN
      (SELECT t2b, COUNT(*)
       FROM table2
       GROUP BY t2b);

```

ANSWER
=====
0 rows

Figure 666, 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
  (SELECT t2b
   FROM table2
   GROUP BY t2b
   HAVING COUNT(*) = 10
  )
SELECT t1.*
FROM table1 t1
      ,t2
WHERE t1.t1a = t2.t2b;

```

TABLE1		TABLE2		
T1A	T1B	T2A	T2B	T2C
A	AA	A	A	A
B	BB	B	A	-
C	CC			

"-" = null

```

SELECT t1.*
FROM table1 t1
      ,(SELECT t2b
       FROM table2
       GROUP BY t2b
       HAVING COUNT(*) = 10
      )AS t2
WHERE t1.t1a = t2.t2b;

```

ANSWER
=====
0 rows

```

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 667, 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 227 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 *
FROM table1
WHERE t1a = ALL
      (SELECT t2b
       FROM table2);

SELECT *
FROM table1
WHERE NOT EXISTS
      (SELECT *
       FROM table2
       WHERE t1a <> t2b);

```

TABLE1		TABLE2		
T1A	T1B	T2A	T2B	T2C
A	AA	A	A	A
B	BB	B	A	-
C	CC			

ANSWER
=====

T1A	T1B
A	AA

"-" = null

Figure 668, 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 *
FROM table1
WHERE t1a = ALL
      (SELECT t2b
       FROM table2
       WHERE t2b >= 'X');

SELECT *
FROM table1
WHERE NOT EXISTS
      (SELECT *
       FROM table2
       WHERE t1a <> t2b
              AND t2b >= 'X');

```

ANSWER
=====

T1A	T1B
A	AA
B	BB
C	CC

Figure 669, 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 *
FROM table1
WHERE t1a = ALL
      (SELECT t2b
       FROM table2
       WHERE t2b >= 'X')
      AND 0 <>
      (SELECT COUNT(*)
       FROM table2
       WHERE t2b >= 'X');

SELECT *
FROM table1
WHERE t1a IN
      (SELECT MAX(t2b)
       FROM table2
       WHERE t2b >= 'X'
       HAVING COUNT(DISTINCT t2b) = 1);

```

TABLE1		TABLE2		
T1A	T1B	T2A	T2B	T2C
A	AA	A	A	A
B	BB	B	A	-
C	CC			

ANSWER
=====

0 rows

Figure 670, 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	R2	R1 UNION R2	R1 UNION ALL R2	R1 INTERSECT R2	R1 INTERSECT ALL R2	R1 EXCEPT R2	R1 EXCEPT ALL 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				
			C				
			C				
			C				
			C				
			D				
			E				

Figure 671, 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

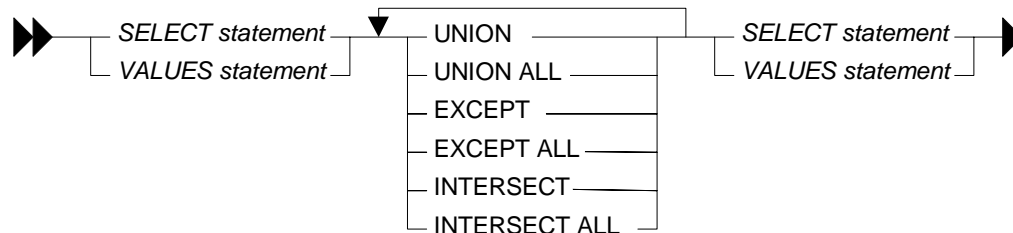


Figure 672, Union, Except, and Intersect syntax

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');

SELECT R1
FROM R1
ORDER BY R1;

SELECT R2
FROM R2
ORDER BY R2;
  
```

ANSWER
=====

R1	R2
A	A
A	A
A	B
B	B
B	B
C	C
C	D
C	
E	

Figure 673, 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
		C	D		B
		C			B
		E			B
					C
					C
					C
					D
					E

Figure 674, 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
		C	C		
		C	D		
		C			
		E			

Figure 675, 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
FROM    R1
EXCEPT
SELECT  R2
FROM    R2
ORDER BY 1;

SELECT  R1
FROM    R1
EXCEPT ALL
SELECT  R2
FROM    R2
ORDER BY 1;

```

R1	R2	R1 EXCEPT R2	R1 EXCEPT ALL R2
--	--	=====	=====
A	A	E	A
A	A		C
A	B		C
B	B		E
B	B		
C	C		
C	D		
C			
E			

Figure 676, 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
FROM    R2
EXCEPT
SELECT  R1
FROM    R1
ORDER BY 1;

SELECT  R2
FROM    R2
EXCEPT ALL
SELECT  R1
FROM    R1
ORDER BY 1;

```

R1	R2	R2 EXCEPT R1	R2 EXCEPT ALL R1
--	--	=====	=====
A	A	D	B
A	A		D
A	B		
B	B		
B	B		
C	C		
C	D		
C			
E			

Figure 677, 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
SELECT  R2      SELECT  R2      (SELECT  R2      A  A
FROM    R2      FROM    R2      FROM    R2      A  A
EXCEPT
SELECT  R2      SELECT  R2      SELECT  R2      B  B
FROM    R2      FROM    R2      FROM    R2      B  B
ORDER BY 1;    ORDER BY 1;    )ORDER BY 1;    C  C
                                                    C  D
                                                    C
                                                    E

```

ANSWER	ANSWER	ANSWER
=====	=====	=====
E	E	A
		B
		C
		E

Figure 678, Use of parenthesis in Union

Unions and Views

Imagine that one has a series of tables that track sales data, with one table for each year. One can define a view that is the UNION ALL of these tables, so that a user would see them as a single object. Such a view can support inserts, updates, and deletes, as long as each table in the view has a constraint that distinguishes it from all the others. Below is an example:

```
CREATE TABLE sales_data_2002
(sales_date      DATE          NOT NULL
,daily_seq#     INTEGER       NOT NULL
,cust_id        INTEGER       NOT NULL
,amount         DEC(10,2)     NOT NULL
,invoice#       INTEGER       NOT NULL
,sales_rep      CHAR(10)      NOT NULL
,CONSTRAINT C CHECK (YEAR(sales_date) = 2002)
,PRIMARY KEY (sales_date, daily_seq#));

CREATE TABLE sales_data_2003
(sales_date      DATE          NOT NULL
,daily_seq#     INTEGER       NOT NULL
,cust_id        INTEGER       NOT NULL
,amount         DEC(10,2)     NOT NULL
,invoice#       INTEGER       NOT NULL
,sales_rep      CHAR(10)      NOT NULL
,CONSTRAINT C CHECK (YEAR(sales_date) = 2003)
,PRIMARY KEY (sales_date, daily_seq#));

CREATE VIEW sales_data AS
SELECT *
FROM   sales_data_2002
UNION ALL
SELECT *
FROM   sales_data_2003;
```

Figure 679, Define view to combine yearly tables

Below is some SQL that changes the contents of the above view:

```
INSERT INTO sales_data VALUES ('2002-11-22',1,123,100.10,996,'SUE')
, ('2002-11-22',2,123,100.10,997,'JOHN')
, ('2003-01-01',1,123,100.10,998,'FRED')
, ('2003-01-01',2,123,100.10,999,'FRED');

UPDATE sales_data
SET   amount = amount / 2
WHERE sales_rep = 'JOHN';

DELETE
FROM   sales_data
WHERE  sales_date = '2003-01-01'
AND    daily_seq# = 2;
```

Figure 680, Insert, update, and delete using view

Below is the view contents, after the above is run:

SALES_DATE	DAILY_SEQ#	CUST_ID	AMOUNT	INVOICE#	SALES_REP
01/01/2003	1	123	100.10	998	FRED
11/22/2002	1	123	100.10	996	SUE
11/22/2002	2	123	50.05	997	JOHN

Figure 681, View contents after insert, update, delete

Materialized Query Tables

Introduction

A materialized query table contains the results of a query. The DB2 optimizer knows this and can, if appropriate, redirect a query that is against the source table(s) to use the materialized query table instead. This can make the query run much faster.

The following statement defines a materialized query table:

```
CREATE TABLE staff_summary AS
  (SELECT   dept
           ,COUNT(*) AS count_rows
           ,SUM(id) AS sum_id
   FROM     staff
   GROUP BY dept)
DATA INITIALLY DEFERRED REFRESH IMMEDIATE;
```

Figure 682, Sample materialized query table DDL

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

ORIGINAL QUERY	OPTIMIZED QUERY
=====	=====
SELECT dept	SELECT Q1.dept AS "dept"
,AVG(id)	,Q1.sum_id / Q1.count_rows
FROM staff	FROM staff_summary AS Q1
GROUP BY dept	

Figure 683, Original and optimized queries

When used appropriately, materialized query tables can cause dramatic improvements in query performance. For example, if in the above STAFF table there was, on average, about 5,000 rows per individual department, referencing the STAFF_SUMMARY table instead of the STAFF table in the sample query might be about 1,000 times faster.

DB2 Optimizer Issues

In order for a materialized query table to be considered for use by the DB2 optimizer, the following has to be true:

- The table has to be refreshed at least once.
- The table MAINTAINED BY parameter and the related DB2 special registers must correspond. For example, if the table is USER maintained, then the CURRENT REFRESH AGE special register must be set to ANY, and the CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION special register must be set to USER or ALL.

See page 246 for more details on these registers.

Usage Notes

A materialized query table is defined using a variation of the standard CREATE TABLE statement. Instead of providing an element list, one supplies a SELECT statement, and defines the refresh option.

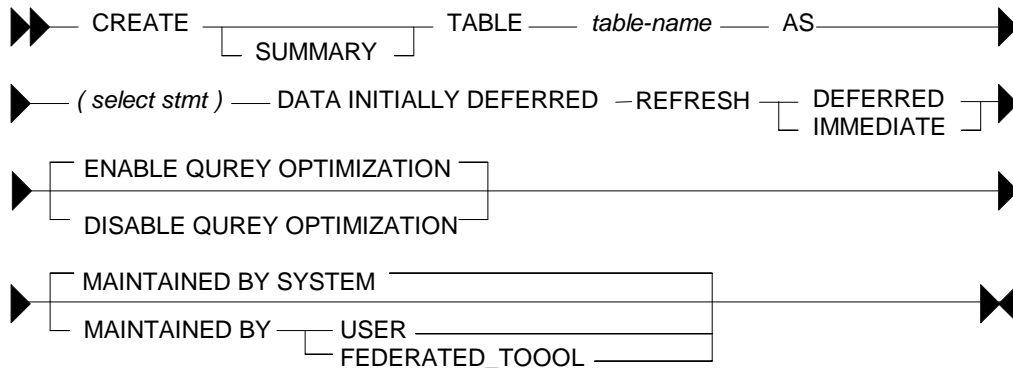


Figure 684, Materialized query table DDL, syntax diagram

Syntax Options

Refresh

- **REFRESH DEFERRED:** The data is refreshed whenever one does a **REFRESH TABLE**. At this point, DB2 will first delete all of the existing rows in the table, then run the select statement defined in the **CREATE** to (you guessed it) repopulate.
- **REFRESH IMMEDIATE:** Once created, this type of table has to be refreshed once using the **REFRESH** statement. From then on, DB2 will maintain the materialized query table in sync with the source table as changes are made to the latter.

Materialized query tables that are defined **REFRESH IMMEDIATE** are obviously more useful in that the data in them is always current. But they may cost quite a bit to maintain, and not all queries can be defined thus.

Query Optimization

- **ENABLE:** The table is used for query optimization when appropriate. This is the default. The table can also be queried directly.
- **DISABLE:** The table will not be used for query optimization. It can be queried directly.

Maintained By

- **SYSTEM:** The data in the materialized query table is maintained by the system. This is the default.
- **USER:** The user is allowed to perform insert, update, and delete operations against the materialized query table. The table cannot be refreshed. This type of table can be used when you want to maintain your own materialized query table (e.g. using triggers) to support features not provided by DB2. The table can also be defined to enable query optimization, but the optimizer will probably never use it as a substitute for a real table.
- **FEDERATED_TOOL:** The data in the materialized query table is maintained by the replication tool. Only a **REFRESH DEFERRED** table can be maintained using this option.

Options vs. Actions

The following table compares materialized query table options to subsequent actions:

MATERIALIZED QUERY TABLE		ALLOWABLE ACTIONS ON TABLE	
REFRESH	MAINTAINED BY	REFRESH TABLE	INSERT/UPDATE/DELETE
DEFERRED	SYSTEM	yes	no
	USER	no	yes
IMMEDIATE	SYSTEM	yes	no

Figure 685, Materialized query table options vs. allowable actions

Select Statement

Various restrictions apply to the select statement that is used to define the materialized query table. In general, materialized query tables defined refresh-immediate need simpler queries than those defined refresh-deferred.

Refresh Deferred Tables

- The query must be a valid SELECT statement.
- Every column selected must have a name.
- An ORDER BY is not allowed.
- Reference to a typed table or typed view is not allowed.
- Reference to declared temporary table is not allowed.
- Reference to a nickname or materialized query table is not allowed.
- Reference to a system catalogue table is not allowed. Reference to an explain table is allowed, but is impudent.
- Reference to NODENUMBER, PARTITION, or any other function that depends on physical characteristics, is not allowed.
- Reference to a datalink type is not allowed.
- Functions that have an external action are not allowed.
- Scalar functions, or functions written in SQL, are not allowed. So SUM(SALARY) is fine, but SUM(INT(SALARY)) is not allowed.

Refresh Immediate Tables

All of the above restrictions apply, plus the following:

- If the query references more than one table or view, it must define as inner join, yet not use the INNER JOIN syntax (i.e. must use old style).
- If there is a GROUP BY, the SELECT list must have a COUNT(*) or COUNT_BIG(*) column.
- Besides the COUNT and COUNT_BIG, the only other column functions supported are SUM and GROUPING - all with the DISTINCT phrase. Any field that allows nulls, and that is summed, but also have a COUNT(column name) function defined.
- Any field in the GROUP BY list must be in the SELECT list.
- The table must have at least one unique index defined, and the SELECT list must include (amongst other things) all the columns of this index.

- Grouping sets, CUBE and ROLLUP are allowed. The GROUP BY items and associated GROUPING column functions in the select list must form a unique key of the result set.
- The HAVING clause is not allowed.
- The DISTINCT clause is not allowed.
- Non-deterministic functions are not allowed.
- Special registers are not allowed.
- If REPLICATED is specified, the table must have a unique key.

Optimizer Options

A materialized query table that has been defined ENABLE QUERY OPTIMIZATION, and has been refreshed, is a candidate for use by the DB2 optimizer if, and only if, three DB2 special registers are set to match the table status:

- CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION.
- CURRENT QUERY OPTIMIZATION.
- CURRENT REFRESH AGE.

Each of the above are discussed below.

CURRENT REFRESH AGE

The refresh age special register tells the DB2 optimizer how up-to-date the data in an materialized query table has to be in order to be considered. There are only two possible values:

- 0: Only use those materialized query tables that are defined as refresh-immediate are eligible. This is the default.
- 99,999,999,999,999: Consider all valid materialized query tables. This is the same as ANY.

NOTE: The above number is a 26-digit decimal value that is a timestamp duration, but without the microsecond component. The value ANY is logically equivalent.

The database default value can be changed using the following command:

```
UPDATE DATABASE CONFIGURATION USING dft_refresh_age ANY;
```

Figure 686, Changing default refresh age for database

The database default value can be overridden within a thread using the SET REFRESH AGE statement. Here is the syntax:

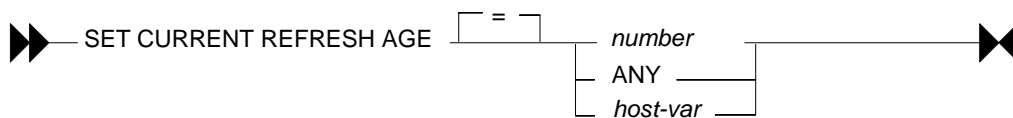


Figure 687, Set refresh age command, syntax

Below are some examples of the SET command:

```
SET CURRENT REFRESH AGE 0;
SET CURRENT REFRESH AGE = ANY;
SET CURRENT REFRESH AGE = 999999999999999;
```

Figure 688, Set refresh age command, examples

CURRENT MAINTAINED TYPES

The current maintained types special register tells the DB2 optimizer what types of materialized query table that are defined refresh deferred are to be considered - assuming that the refresh-age parameter is not set to zero:

- ALL: All refresh-deferred materialized query tables are to be considered. If this option is chosen, no other option can be used.
- NONE: No refresh-deferred materialized query tables are to be considered. If this option is chosen, no other option can be used.
- SYSTEM: System-maintained refresh-deferred materialized query tables are to be considered. This is the default.
- USER: User-maintained refresh-deferred materialized query tables are to be considered.
- FEDERATED TOOL: Federated-tool-maintained refresh-deferred materialized query tables are to be considered, but only if the CURRENT QUERY OPTIMIZATION special register is 2 or greater than 5.
- CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION: The existing values for this special register are used.

The database default value can be changed using the following command:

```
UPDATE DATABASE CONFIGURATION USING dft_refresh_age ANY;
```

Figure 689, Changing default maintained type for database

The database default value can be overridden within a thread using the SET REFRESH AGE statement. Here is the syntax:

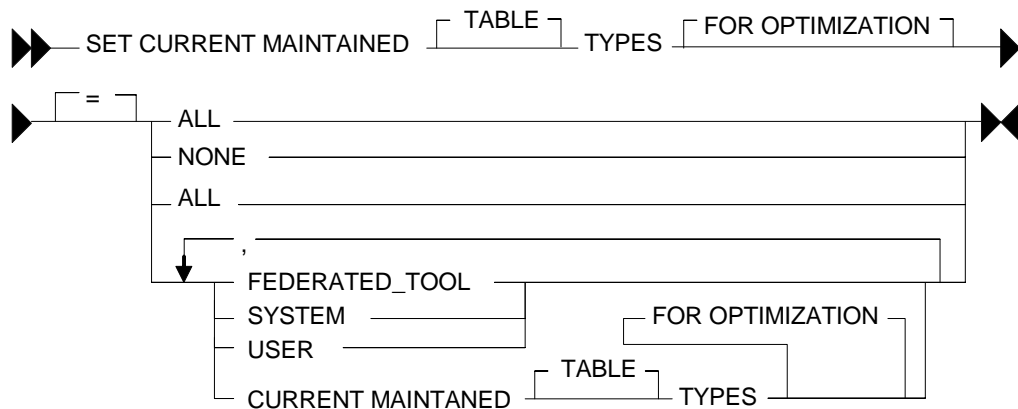


Figure 690, Set maintained type command, syntax

Below are some examples of the SET command:

```
SET CURRENT MAINTAINED          TYPES          = ALL;
SET CURRENT MAINTAINED TABLE  TYPES          = SYSTEM;
SET CURRENT MAINTAINED TABLE  TYPES FOR OPTIMIZATION = USER, SYSTEM;
```

Figure 691, Set maintained type command, examples

CURRENT QUERY OPTIMIZATION

The current query optimization special register tells the DB2 optimizer what set of optimization techniques to use. The value can range from zero to nine - except for four or eight. A value of five or above will cause the optimizer to consider using materialized query tables.

The database default value can be changed using the following command:

```
UPDATE DATABASE CONFIGURATION USING DFT_QUERYOPT 5;
```

Figure 692, Changing default maintained type for database

The database default value can be overridden within a thread using the SET CURRENT QUERY OPTIMIZATION statement. Here is the syntax:

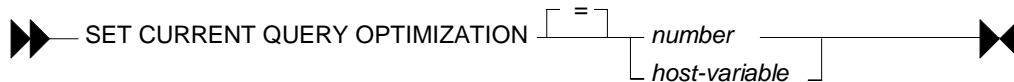


Figure 693, Set maintained type command, syntax

Below are an example of the SET command:

```
SET CURRENT QUERY OPTIMIZATION = 9;
```

figure 694, Set query optimization, example

What Matches What

Assuming that the current query optimization special register is set to five or above, the DB2 optimizer will consider using a materialized query table (instead of the base table) when any of the following conditions are true:

MQT DEFINITION		DATABASE/APPLICATION STATUS		DB2
=====		=====		USE
REFRESH	MAINTAINED-BY	REFRESH-AGE	MAINTAINED-TYPE	MQT
=====	=====	=====	=====	===
IMMEDIATE	SYSTEM	-	-	Yes
DEFERRED	SYSTEM	ANY	ALL or SYSTEM	Yes
DEFERRED	USER	ANY	ALL or USER	Yes
DEFERRED	FEDERATED-TOOL	ANY	ALL or FEDERATED-TOOL	Yes

Figure 695, When DB2 will consider using a materialized query table

Selecting Special Registers

One can select the relevant special register to see what the values are:

```
SELECT CURRENT REFRESH AGE AS age_ts
, CURRENT TIMESTAMP AS current_ts
, CURRENT QUERY OPTIMIZATION AS q_opt
FROM sysibm.sysdummy1;
```

Figure 696, Selecting special registers

Refresh Deferred Tables

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


```

CREATE TABLE staff_names AS
  (SELECT   dept
           ,COUNT(*)           AS count_rows
           ,SUM(salary)         AS sum_salary
           ,AVG(salary)         AS avg_salary
           ,MAX(salary)         AS max_salary
           ,MIN(salary)         AS min_salary
           ,STDDEV(salary)      AS std_salary
           ,VARIANCE(salary)    AS var_salary
           ,CURRENT_TIMESTAMP AS last_change
  FROM     staff
  WHERE    TRANSLATE(name) LIKE '%A%'
  AND     salary                > 10000
  GROUP BY dept
  HAVING   COUNT(*) = 1
  )DATA INITIALLY DEFERRED REFRESH DEFERRED;

```

Figure 697, Refresh deferred materialized query table DDL

Refresh Immediate Tables

A materialized query table defined REFRESH IMMEDIATE is automatically maintained in sync with the source table by DB2. As with any materialized query table, it is defined by referring to a query. Below is a table that refers to a single source table:

```

CREATE TABLE emp_summary AS
  (SELECT   emp.workdept
           ,COUNT(*)           AS num_rows
           ,COUNT(emp.salary) AS num_salary
           ,SUM(emp.salary)     AS sum_salary
           ,COUNT(emp.comm)   AS num_comm
           ,SUM(emp.comm)      AS sum_comm
  FROM     employee emp
  GROUP BY emp.workdept
  )DATA INITIALLY DEFERRED REFRESH IMMEDIATE;

```

Figure 698, Refresh immediate materialized query table DDL

Below is a query that can use the above materialized query table in place of the base table:

```

SELECT   emp.workdept
         ,DEC(SUM(emp.salary),8,2) AS sum_sal
         ,DEC(AVG(emp.salary),7,2) AS avg_sal
         ,SMALLINT(COUNT(emp.comm)) AS #comms
         ,SMALLINT(COUNT(*))       AS #emps
  FROM   employee emp
  WHERE  emp.workdept > 'C'
  GROUP BY emp.workdept
  HAVING COUNT(*) <> 5
  AND    SUM(emp.salary) > 50000
  ORDER BY sum_sal DESC;

```

Figure 699, Query that uses materialized query table (1 of 3)

The next query can also use the materialized query table. This time, the data returned from the materialized query table is qualified by checking against a sub-query:

```

SELECT   emp.workdept
         ,COUNT(*) AS #rows
  FROM   employee emp
  WHERE  emp.workdept IN
        (SELECT deptno
         FROM   department
         WHERE  deptname LIKE '%S%')
  GROUP BY emp.workdept
  HAVING   SUM(salary) > 50000;

```

Figure 700, Query that uses materialized query table (2 of 3)

This last example uses the materialized query table in a nested table expression:

```

SELECT   #emps
         ,DEC(SUM(sum_sal),9,2)   AS sal_sal
         ,SMALLINT(COUNT(*))     AS #depts
FROM     (SELECT   emp.workdept
         ,DEC(SUM(emp.salary),8,2) AS sum_sal
         ,MAX(emp.salary)        AS max_sal
         ,SMALLINT(COUNT(*))    AS #emps
         FROM     employee emp
         GROUP BY emp.workdept
        )AS XXX
GROUP BY #emps
HAVING   COUNT(*) > 1
ORDER BY #emps
FETCH FIRST 3 ROWS ONLY
OPTIMIZE FOR 3 ROWS;

```

Figure 701, Query that uses materialized query table (3 of 3)

Using Materialized Query Tables to Duplicate Data

All of the above materialized query tables have contained a GROUP BY in their definition. But this is not necessary. To illustrate, we will first create a simple table:

```

CREATE TABLE staff_all
(id          SMALLINT      NOT NULL
,name       VARCHAR(9)    NOT NULL
,job        CHAR(5)
,salary     DECIMAL(7,2)
,PRIMARY KEY(id));

```

Figure 702, Create source table

As long as the above table has a primary key, which it does, we can define a duplicate of the above using the following code:

```

CREATE TABLE staff_all_dup AS
(SELECT *
 FROM   staff_all)
DATA INITIALLY DEFERRED REFRESH IMMEDIATE;

```

Figure 703, Create duplicate data table

We can also decide to duplicate only certain rows:

```

CREATE TABLE staff_all_dup_some AS
(SELECT *
 FROM   staff_all
 WHERE  id < 30)
DATA INITIALLY DEFERRED REFRESH IMMEDIATE;

```

Figure 704, Create table - duplicate certain rows only

Imagine that we had another table that listed all those staff that we are about to fire:

```

CREATE TABLE staff_to_fire
(id          SMALLINT      NOT NULL
,name       VARCHAR(9)    NOT NULL
,dept       SMALLINT
,PRIMARY KEY(id));

```

Figure 705, Create source table

We can create materialized query table that joins the above two staff tables as long as the following is true:

- Both tables have identical primary keys (i.e. same number of columns).
- The join is an inner join on the common primary key fields.

- All primary key columns are listed in the SELECT.

Now for an example:

```
CREATE TABLE staff_combo AS
  (SELECT   aaa.id      AS id1
           ,aaa.job     AS job
           ,fff.id      AS id2
           ,fff.dept    AS dept
    FROM     staff_all   aaa
           ,staff_to_fire fff
    WHERE    aaa.id = fff.id)
DATA INITIALLY DEFERRED REFRESH IMMEDIATE;
```

Figure 706, Materialized query table on join

See page 252 for more examples of join usage.

Queries that don't use Materialized Query Table

Below is a query that can not use the EMP_SUMMARY table because of the reference to the MAX function. Ironically, this query is exactly the same as the nested table expression above, but in the prior example the MAX is ignored because it is never actually selected:

```
SELECT   emp.workdept
         ,DEC(SUM(emp.salary),8,2) AS sum_sal
         ,MAX(emp.salary)         AS max_sal
FROM     employee emp
GROUP BY emp.workdept;
```

Figure 707, Query that doesn't use materialized query table (1 of 2)

The following query can't use the materialized query table because of the DISTINCT clause:

```
SELECT   emp.workdept
         ,DEC(SUM(emp.salary),8,2) AS sum_sal
         ,COUNT(DISTINCT salary) AS #salaries
FROM     employee emp
GROUP BY emp.workdept;
```

Figure 708, Query that doesn't use materialized query table (2 of 2)

Usage Notes and Restrictions

- A materialized query table must be refreshed before it can be queried. If the table is defined refresh immediate, then the table will be maintained automatically after the initial refresh.
- Make sure to commit after doing a refresh. The refresh does not have an implied commit.
- Run RUNSTATS after refreshing a materialized query table.
- One can not load data into materialized query tables.
- One can not directly update materialized query tables.

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

```
REFRESH TABLE emp_summary;
COMMIT;

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

Figure 709, Materialized query table refresh commands

Multi-table Materialized Query Tables

Single-table materialized query tables save having to look at individual rows to resolve a GROUP BY. Multi-table materialized query tables do this, and also avoid having to resolve a join.

```
CREATE TABLE dept_emp_summary AS
  (SELECT   emp.workdept
           ,dpt.deptname
           ,COUNT(*)           AS num_rows
           ,COUNT(emp.salary) AS num_salary
           ,SUM(emp.salary)     AS sum_salary
           ,COUNT(emp.comm)    AS num_comm
           ,SUM(emp.comm)       AS sum_comm
  FROM     employee emp
           ,department dpt
  WHERE    dpt.deptno = emp.workdept
  GROUP BY emp.workdept
           ,dpt.deptname
  )DATA INITIALLY DEFERRED REFRESH IMMEDIATE;
```

Figure 710, Multi-table materialized query table DDL

The following query is resolved using the above materialized query table:

```
SELECT   d.deptname
         ,d.deptno
         ,DEC(AVG(e.salary),7,2) AS avg_sal
         ,SMALLINT(COUNT(*))    AS #emps
FROM     department d
         ,employee e
WHERE    e.workdept = d.deptno
        AND d.deptname LIKE '%S%'
GROUP BY d.deptname
         ,d.deptno
HAVING   SUM(e.comm) > 4000
ORDER BY avg_sal DESC;
```

Figure 711, Query that uses materialized query table

Here is the SQL that DB2 generated internally to get the answer:

```
SELECT   Q2.$C0 AS "deptname"
         ,Q2.$C1 AS "deptno"
         ,Q2.$C2 AS "avg_sal"
         ,Q2.$C3 AS "#emps"
FROM     (SELECT   Q1.deptname           AS $C0
               ,Q1.workdept           AS $C1
               ,DEC((Q1.sum_salary / Q1.num_salary),7,2) AS $C2
               ,SMALLINT(Q1.num_rows) AS $C3
           FROM     dept_emp_summary AS Q1
           WHERE    (Q1.deptname LIKE '%S%')
               AND  (4000 < Q1.sum_comm)
           )AS Q2
ORDER BY Q2.$C2 DESC;
```

Figure 712, DB2 generated query to use materialized query table

Rules and Restrictions

- The join must be an inner join, and it must be written in the old style syntax.
- Every table accessed in the join (except one?) must have a unique index.
- The join must not be a Cartesian product.
- The GROUP BY must include all of the fields that define the unique key for every table (except one?) in the join.

Three-table Example

```

CREATE TABLE dpt_emp_act_sumry AS
  (SELECT   emp.workdept
           ,dpt.deptname
           ,emp.empno
           ,emp.firstnme
           ,SUM(act.emptime) AS sum_time
           ,COUNT(act.emptime) AS num_time
           ,COUNT(*) AS num_rows
  FROM     department dpt
           ,employee emp
           ,emp_act act
  WHERE    dpt.deptno = emp.workdept
           AND emp.empno = act.empno
  GROUP BY emp.workdept
           ,dpt.deptname
           ,emp.empno
           ,emp.firstnme
  )DATA INITIALLY DEFERRED REFRESH IMMEDIATE;

```

Figure 713, Three-table materialized query table DDL

Now for a query that will use the above:

```

SELECT   d.deptno
         ,d.deptname
         ,DEC(AVG(a.emptime),5,2) AS avg_time
  FROM   department d
         ,employee e
         ,emp_act a
  WHERE  d.deptno = e.workdept
         AND e.empno = a.empno
         AND d.deptname LIKE '%S%'
         AND e.firstnme LIKE '%S%'
  GROUP BY d.deptno
         ,d.deptname
  ORDER BY 3 DESC;

```

Figure 714, Query that uses materialized query table

And here is the DB2 generated SQL:

```

SELECT   Q4.$C0 AS "deptno"
         ,Q4.$C1 AS "deptname"
         ,Q4.$C2 AS "avg_time"
  FROM   (SELECT   Q3.$C3 AS $C0
           ,Q3.$C2 AS $C1
           ,DEC((Q3.$C1 / Q3.$C0),5,2) AS $C2
         FROM     (SELECT   SUM(Q2.$C2) AS $C0
                   ,SUM(Q2.$C3) AS $C1
                   ,Q2.$C0 AS $C2
                   ,Q2.$C1 AS $C3
                 FROM     (SELECT   Q1.deptname AS $C0
                               ,Q1.workdept AS $C1
                               ,Q1.num_time AS $C2
                               ,Q1.sum_time AS $C3
                             FROM     dpt_emp_act_sumry AS Q1
                             WHERE    (Q1.firstnme LIKE '%S%')
                             AND      (Q1.DEPTNAME LIKE '%S%')
                           )AS Q2
                   GROUP BY Q2.$C1
                   ,Q2.$C0
                 )AS Q3
         )AS Q4
  ORDER BY Q4.$C2 DESC;

```

Figure 715, DB2 generated query to use materialized query table

Indexes on Materialized Query Tables

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

- Unique indexes are not allowed.
- The materialized query table must not be in a "check pending" status when the index is defined. Run a refresh to address this problem.

Below are some indexes for the DPT_EMP_ACT_SUMRY table that was defined above:

```
CREATE INDEX dpt_emp_act_sumx1
      ON dpt_emp_act_sumry
      (workdept
      ,deptname
      ,empno
      ,firstnme);

CREATE INDEX dpt_emp_act_sumx2
      ON dpt_emp_act_sumry
      (num_rows);
```

Figure 716, Indexes for DPT_EMP_ACT_SUMRY materialized query 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 717, 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 718, Sample query that uses NUM_ROWS index

Organizing by Dimensions

The following materialized query table is organized (clustered) by the two columns that are referred to in the GROUP BY. Under the covers, DB2 will also create a dimension index on each column, and a block index on both columns combined:

```
CREATE TABLE emp_sum AS
  (SELECT   workdept
           ,job
           ,SUM(salary)           AS sum_sal
           ,COUNT(*)            AS #emps
           ,GROUPING(workdept)   AS grp_dpt
           ,GROUPING(job)        AS grp_job
  FROM     employee
  GROUP BY CUBE(workdept
                ,job))
DATA INITIALLY DEFERRED REFRESH DEFERRED
ORGANIZE BY DIMENSIONS (workdept, job)
IN tsempsum;
```

Figure 719, Materialized query table organized by dimensions

WARNING: Multi-dimensional tables may perform very poorly when created in the default tablespace, or in a system-maintained tablespace. Use a database-maintained tablespace with the right extent size, and/or run the DB2EMPFA command.

Don't forget to run RUNSTATS!

Using Staging Tables

A staging table can be used to incrementally maintain a materialized query table that has been defined refresh deferred. Using a staging table can result in a significant performance saving (during the refresh) if the source table is very large, and is not changed very often.

NOTE: To use a staging table, the SQL statement used to define the target materialized query table must follow the rules that apply for a table that is defined refresh immediate - even though it is defined refresh deferred.

The staging table CREATE statement has the following components:

- The name of the staging table.
- A list of columns (with no attributes) in the target materialized query table. The column names do not have to match those in the target table.
- Either two or three additional columns with specific names- as provided by DB2.
- The name of the target materialized query table.

To illustrate, below is a typical materialized query table:

```
CREATE TABLE emp_sumry AS
  (SELECT   workdept           AS dept
           ,COUNT(*)         AS #rows
           ,COUNT(salary)    AS #sal
           ,SUM(salary)        AS sum_sal
  FROM     employee emp
  GROUP BY emp.workdept
  )DATA INITIALLY DEFERRED REFRESH DEFERRED;
```

Figure 720, Sample materialized query table

Here is a staging table for the above:

```

CREATE TABLE emp_sumry_s
  (dept
  ,num_rows
  ,num_sal
  ,sum_sal
  ,GLOBALTRANSID
  ,GLOBALTRANSTIME
  )FOR emp_sumry PROPAGATE IMMEDIATE;

```

Figure 721, Staging table for the above materialized query table

Additional Columns

The two, or three, additional columns that every staging table must have are as follows:

- **GLOBALTRANSID:** The global transaction ID for each propagated row.
- **GLOBALTRANSTIME:** The transaction timestamp
- **OPERATIONTYPE:** The operation type (i.e. insert, update, or delete). This column is needed if the target materialized query table does not contain a **GROUP BY** statement.

Using a Staging Table

To activate the staging table one must first use the **SET INTEGRITY** command to remove the check pending flag, and then do a full refresh of the target materialized query table. After this is done, the staging table will record all changes to the source table.

Use the refresh incremental command to apply the changes recorded in the staging table to the target materialized query table.

```

SET INTEGRITY FOR emp_sumry_s STAGING IMMEDIATE UNCHECKED;
REFRESH TABLE emp_sumry;

```

<< make changes to the source table (i.e. employee) >>

```

REFRESH TABLE emp_sumry INCREMENTAL;

```

Figure 722, Enabling and the using a staging table

A multi-row update (or insert, or delete) uses the same **CURRENT TIMESTAMP** for all rows changed, and for all invoked triggers. Therefore, the **#CHANGING_SQL** field is only incremented when a new timestamp value is detected.

Identity Columns and Sequences

Imagine that one has an INVOICE table that records invoices generated. Also imagine that one wants every new invoice that goes into this table to get an invoice number value that is part of a unique and unbroken sequence of ascending values - assigned in the order that the invoices are generated. So if the highest invoice number is currently 12345, then the next invoice will get 12346, and then 12347, and so on.

There are three ways to do this, up to a point:

- Use an identity column, which generates a unique value per row in a table.
- Use a sequence, which generates a unique value per one or more tables.
- Do it yourself, using an insert trigger to generate the unique values.

You may need to know what values were generated during each insert. There are several ways to do this:

- For all of the above techniques, embed the insert inside a select statement (see figure 737 and/or page 67). This is probably the best solution.
- For identity columns, use the IDENTITY_VAL_LOCAL function (see page 263).
- For sequences, make a NEXTVAL or PREVVAL call (see page 266).

Living With Gaps

The only way that one can be absolutely certain not to have a gap in the sequence of values generated is to create your own using an insert trigger. However, this solution is probably the least efficient of those listed here, and it certainly has the least concurrency.

There is almost never a valid business reason for requiring an unbroken sequence of values. So the best thing to do, if your users ask for such a feature, is to beat them up.

Living With Sequence Errors

For efficiency reasons, identity column and sequence values are usually handed out (to users doing inserts) in block of values, where the block size is defined using the CACHE option. If a user inserts a row, and then dithers for a bit before inserting another, it is possible that some other user (with a higher value) will insert first. In this case, the identity column or sequence value will be a good approximation of the insert sequence, but not right on.

If the users need to know the precise order with which rows were inserted, then either set the cache size to one, which will cost, or include a current timestamp value.

Identity Columns

One can define a column in a DB2 table as an "identity column". This column, which must be numeric (note: fractional fields not allowed), will be incremented by a fixed constant each time a new row is inserted. Below is a syntax diagram for that part of a CREATE TABLE statement that refers to an identity column definition:

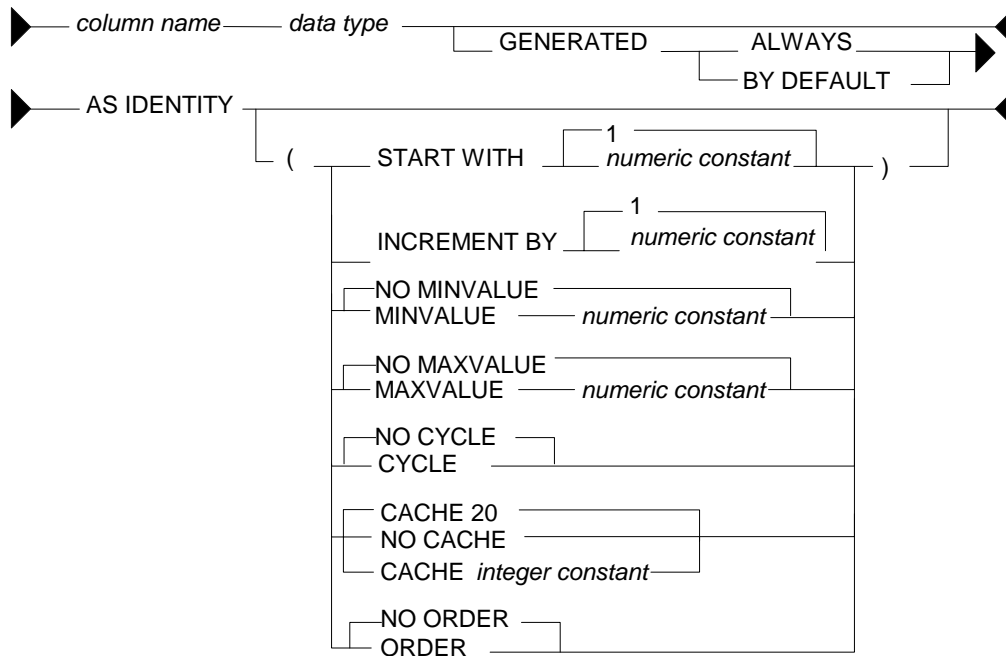


Figure 723, Identity Column syntax

Below is an example of a typical invoice table that uses an identity column that starts at one, and then goes ever upwards:

```

CREATE TABLE invoice_data
(invoice#          INTEGER          NOT NULL
 GENERATED ALWAYS AS IDENTITY
   (START WITH 1
   , INCREMENT BY 1
   , NO MAXVALUE
   , NO CYCLE
   , ORDER)
, sale_date       DATE              NOT NULL
, customer_id     CHAR(20)          NOT NULL
, product_id      INTEGER           NOT NULL
, quantity        INTEGER           NOT NULL
, price           DECIMAL(18,2)     NOT NULL
, PRIMARY KEY    (invoice#));

```

Figure 724, Identity column, sample table

Rules and Restrictions

Identity columns come in one of two general flavors:

- The value is always generated by DB2.
- The value is generated by DB2 only if the user does not provide a value (i.e. by default). This configuration is typically used when the input is coming from an external source (e.g. data propagation).

Rules

- There can only be one identity column per table.
- The field cannot be updated if it is defined "generated always".

- The column type must be numeric and must not allow fractional values. Any integer type is OK. Decimal is also fine, as long as the scale is zero. Floating point is a no-no.
- The identity column value is generated before any BEFORE triggers are applied. Use a trigger transition variable to see the value.
- A unique index is not required on the identity column, but it is a good idea. Certainly, if the value is being created by DB2, then a non-unique index is a fairly stupid idea.
- Unlike triggers, identity column logic is invoked and used during a LOAD. However, a load-replace will not reset the identity column value. Use the RESTART command (see below) to do this. An identity column is not affected by a REORG.

Syntax Notes

- **START WITH** defines the start value, which can be any valid integer value. If no start value is provided, then the default is the MINVALUE for ascending sequences, and the MAXVALUE for descending sequences. If this value is also not provided, then the default is 1.
- **INCREMENT BY** defines the interval between consecutive values. This can be any valid integer value, though using zero is pretty silly. The default is 1.
- **MINVALUE** defines (for ascending sequences) the value that the sequence will start at if no start value is provided. It is also the value that an ascending sequence will begin again at after it reaches the maximum and loops around. If no minimum value is provided, then after reaching the maximum the sequence will begin again at the start value. If that is also not defined, then the sequence will begin again at 1, which is the default start value.
- For descending sequences, it is the minimum value that will be used before the sequence loops around, and starts again at the maximum value.
- **MAXVALUE** defines (for ascending sequences) the value that a sequence will stop at, and then go back to the minimum value. For descending sequences, it is the start value (if no start value is provided), and also the restart value - if the sequence reaches the minimum and loops around.
- **CYCLE** defines whether the sequence should cycle about when it reaches the maximum value (for an ascending sequences), or whether it should stop. The default is no cycle.
- **CACHE** defines whether or not to allocate sequences values in chunks, and thus to save on log writes. The default is no cache, which means that every row inserted causes a log write (to save the current value).
- If a cache value (from 2 to 20) is provided, then the new values are assigned to a common pool in blocks. Each insert user takes from the pool, and only when all of the values are used is a new block (of values) allocated and a log write done. If the table is deactivated, either normally or otherwise, then the values in the current block are discarded, resulting in gaps in the sequence. Gaps in the sequence of values also occur when an insert is subsequently rolled back, so they cannot be avoided. But don't use the cache if you want to try and avoid them.
- **ORDER** defines whether all new rows inserted are assigned a sequence number in the order that they were inserted. The default is no, which means that occasionally a row that is inserted after another may get a slightly lower sequence number. This is the default.

Identity Column Examples

The following example uses all of the defaults to start an identity column at one, and then to go up in increments of one. The inserts will eventually die when they reach the maximum allowed value for the field type (i.e. for small integer = 32K).

```
CREATE TABLE test_data                                KEY# FIELD - VALUES ASSIGNED
(key# SMALLINT NOT NULL                               =====
 GENERATED ALWAYS AS IDENTITY                       1 2 3 4 5 6 7 8 9 10 11 etc.
,dat1 SMALLINT NOT NULL
,ts1  TIMESTAMP NOT NULL
,PRIMARY KEY(key#));
```

Figure 725, Identity column, ascending sequence

The next example defines an identity column that goes down in increments of -3:

```
CREATE TABLE test_data                                KEY# FIELD - VALUES ASSIGNED
(key# SMALLINT NOT NULL                               =====
 GENERATED ALWAYS AS IDENTITY                       6 3 0 -3 -6 -9 -12 -15 etc.
 (START WITH 6
, INCREMENT BY -3
, NO CYCLE
, NO CACHE
, ORDER)
,dat1 SMALLINT NOT NULL
,ts1  TIMESTAMP NOT NULL
,PRIMARY KEY(key#));
```

Figure 726, Identity column, descending sequence

The next example, which is amazingly stupid, goes nowhere fast. A primary key cannot be defined on this table:

```
CREATE TABLE test_data                                KEY# VALUES ASSIGNED
(key# SMALLINT NOT NULL                               =====
 GENERATED ALWAYS AS IDENTITY                       123 123 123 123 123 123 etc.
 (START WITH 123
, MAXVALUE 124
, INCREMENT BY 0
, NO CYCLE
, NO ORDER)
,dat1 SMALLINT NOT NULL
,ts1  TIMESTAMP NOT NULL);
```

Figure 727, Identity column, dumb sequence

The next example uses every odd number up to the maximum (i.e. 6), then loops back to the minimum value, and goes through the even numbers, ad-infinitum:

```
CREATE TABLE test_data                                KEY# VALUES ASSIGNED
(key# SMALLINT NOT NULL                               =====
 GENERATED ALWAYS AS IDENTITY                       1 3 5 2 4 6 2 4 6 2 4 6 etc.
 (START WITH 1
, INCREMENT BY 2
, MAXVALUE 6
, MINVALUE 2
, CYCLE
, NO CACHE
, ORDER)
,dat1 SMALLINT NOT NULL
,ts1  TIMESTAMP NOT NULL);
```

Figure 728, Identity column, odd values, then even, then stuck

Usage Examples

Below is the DDL for a simplified invoice table where the primary key is an identity column. Observe that the invoice# is always generated by DB2:

```

CREATE TABLE invoice_data
(invoice#          INTEGER                NOT NULL
      GENERATED ALWAYS AS IDENTITY
      (START WITH 100
       , INCREMENT BY 1
       , NO CYCLE
       , ORDER)
,sale_date        DATE                    NOT NULL
,customer_id      CHAR(20)                NOT NULL
,product_id       INTEGER                 NOT NULL
,quantity         INTEGER                 NOT NULL
,price            DECIMAL(18,2)           NOT NULL
,PRIMARY KEY      (invoice#));

```

Figure 729, Identity column, definition

One cannot provide a value for the invoice# when inserting into the above table. Therefore, one must either use a default placeholder, or leave the column out of the insert. An example of both techniques is given below. The second insert also selects the generated values:

```

INSERT INTO invoice_data
VALUES (DEFAULT, '2001-11-22', 'ABC', 123, 100, 10);

SELECT invoice#          ANSWER
FROM   FINAL TABLE
      (INSERT INTO invoice_data
      (sale_date, customer_id, product_id, quantity, price)
      VALUES ('2002-11-22', 'DEF', 123, 100, 10)
            , ('2003-11-22', 'GHI', 123, 100, 10));

```

Figure 730, 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	2001-11-22	ABC	123	100	10.00
101	2002-11-22	DEF	123	100	10.00
102	2003-11-22	GHI	123	100	10.00

Figure 731, Invoice table, after inserts

Altering Identity Column Options

Imagine that the application is happily collecting invoices in the above table, but your silly boss is unhappy because not enough invoices, as measured by the ever-ascending invoice# value, are being generated per unit of time. We can improve things without actually fixing any difficult business problems by simply altering the invoice# current value and the increment using the ALTER TABLE ... RESTART command:

```

ALTER TABLE invoice_data
ALTER COLUMN invoice#
  RESTART WITH 1000
  SET INCREMENT BY 2;

```

Figure 732, Invoice table, restart identity column value

Now imagine that we insert two more rows thus:

```

INSERT INTO invoice_data
VALUES (DEFAULT, '2004-11-24', 'XXX', 123, 100, 10)
      , (DEFAULT, '2004-11-25', 'YYY', 123, 100, 10);

```

Figure 733, Invoice table, more sample inserts

Our mindless management will now see this data:

INVOICE#	SALE_DATE	CUSTOMER_ID	PRODUCT_ID	QUANTITY	PRICE
100	2001-11-22	ABC	123	100	10.00
101	2002-11-22	DEF	123	100	10.00
102	2003-11-22	GHI	123	100	10.00
1000	2004-11-24	XXX	123	100	10.00
1002	2004-11-25	YYY	123	100	10.00

Figure 734, Invoice table, after second inserts

Alter Usage Notes

The identity column options can be changed using the ALTER TABLE command:

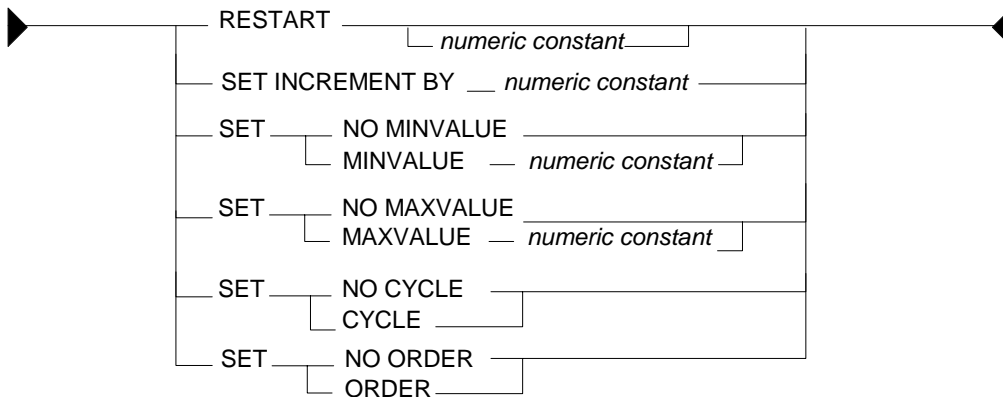


Figure 735, Identity Column alter syntax

Restarting the identity column start number to a lower number, or to a higher number if the increment is a negative value, can result in the column getting duplicate values. This can also occur if the increment value is changed from positive to negative, or vice-versa. If no value is provided for the restart option, the sequence restarts at the previously defined start value.

Gaps in Identity Column Values

If an identity column is generated always, and no cache is used, and the increment value is 1, then there will usually be no gaps in the sequence of assigned values. But gaps can occur if an insert is subsequently rolled out instead of committed. In the following example, there will be no row in the table with customer number "1" after the rollback:

```

CREATE TABLE customers
(cust#          INTEGER          NOT NULL
 GENERATED ALWAYS AS IDENTITY (NO CACHE)
 ,cname        CHAR(10)         NOT NULL
 ,ctype        CHAR(03)         NOT NULL
 ,PRIMARY KEY  (cust#));
COMMIT;

SELECT cust#
FROM FINAL TABLE
(ININSERT INTO customers
VALUES (DEFAULT, 'FRED', 'XXX'));
ROLLBACK;
ANSWER
=====
CUST#
-----
1

SELECT cust#
FROM FINAL TABLE
(ININSERT INTO customers
VALUES (DEFAULT, 'FRED', 'XXX'));
COMMIT;
ANSWER
=====
CUST#
-----
2
    
```

Figure 736, Gaps in Values, example

IDENTITY_VAL_LOCAL Function

There are two ways to find out what values were generated when one inserted a row into a table with an identity column:

- Embed the insert within a select statement (see figure 737).
- Call the IDENTITY_VAL_LOCAL function.

Certain rules apply to IDENTITY_VAL_LOCAL function usage:

- The value returned from is a decimal (31.0) field.
- The function returns null if the user has not done a single-row insert in the current unit of work. Therefore, the function has to be invoked before one does a commit. Having said this, in some versions of DB2 it seems to work fine after a commit.
- If the user inserts multiple rows into table(s) having identity columns in the same unit of work, the result will be the value obtained from the last single-row insert. The result will be null if there was none.
- Multiple-row inserts are ignored by the function. So if the user first inserts one row, and then separately inserts two rows (in a single SQL statement), the function will return the identity column value generated during the first insert.
- The function cannot be called in a trigger or SQL function. To get the current identity column value in an insert trigger, use the trigger transition variable for the column. The value, and thus the transition variable, is defined before the trigger is begun.
- If invoked inside an insert statement (i.e. as an input value), the value will be taken from the most recent (previous) single-row insert done in the same unit of work. The result will be null if there was none.
- The value returned by the function is unpredictable if the prior single-row insert failed. It may be the value from the insert before, or it may be the value given to the failed insert.
- The function is non-deterministic, which means that the result is determined at fetch time (i.e. not at open) when used in a cursor. So if one fetches a row from a cursor, and then does an insert, the next fetch may get a different value from the prior.
- The value returned by the function may not equal the value in the table - if either a trigger or an update has changed the field since the value was generated. This can only occur if the identity column is defined as being "generated by default". An identity column that is "generated always" cannot be updated.
- When multiple users are inserting into the same table concurrently, each will see their own most recent identity column value. They cannot see each other's.

If the above sounds unduly complex, it is because it is. It is often much easier to simply get the values by embedding the insert inside a select:

```

SELECT  MIN(cust#) AS minc                               ANSWER
        ,MAX(cust#) AS maxc                               =====
        ,COUNT(*) AS rows                               MINC MAXC ROWS
FROM    FINAL TABLE
(INsert INTO customers
VALUES (DEFAULT, 'FRED', 'xxx')
        ,(DEFAULT, 'DAVE', 'yyy')
        ,(DEFAULT, 'JOHN', 'zzz'));

```

Figure 737, Selecting identity column values inserted

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
(VALUES (IDENTITY_VAL_LOCAL()))
SELECT *
FROM   temp;

COMMIT;

WITH temp (id) AS
(VALUES (IDENTITY_VAL_LOCAL()))
SELECT *
FROM   temp;
```

```
<<< ANSWER
=====
      ID
-----
      1

<<< ANSWER
=====
      ID
-----
      1
```

Figure 738, *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);

SELECT   invoice#          AS inv#
        ,sale_date
        ,IDENTITY_VAL_LOCAL() AS id
FROM     invoice_table
ORDER BY 1;
COMMIT;
```

```
ANSWER
=====
INV# SALE_DATE  ID
---- -
1 11/22/2000  2
2 11/23/2000  2
3 11/24/2000  2
4 11/25/2000  2
```

Figure 739, *IDENTITY_VAL_LOCAL* function examples

One can also use the function to get the most recently inserted single row by the current user:

```
SELECT   invoice#          AS inv#
        ,sale_date
        ,IDENTITY_VAL_LOCAL() AS id
FROM     invoice_table
WHERE    id = IDENTITY_VAL_LOCAL();
```

```
ANSWER
=====
INV# SALE_DATE  ID
---- -
2 11/23/2000  2
```

Figure 740, *IDENTITY_VAL_LOCAL* usage in predicate

Sequences

A sequence is almost the same as an identity column, except that it is an object that exists outside of any particular table.

```
CREATE SEQUENCE fred                               SEQ# VALUES ASSIGNED
AS DECIMAL(31)                                    =====
START WITH 100                                    100 102 104 106 etc.
INCREMENT BY 2
NO MINVALUE
NO MAXVALUE
NO CYCLE
CACHE 20
ORDER;
```

Figure 741, Create sequence

The options and defaults for a sequence are exactly the same as those for an identity column (see page 259). Likewise, one can alter a sequence in much the same way as one would alter the status of an identity column:

```
ALTER SEQUENCE fred                               SEQ# VALUES ASSIGNED
RESTART WITH -55                                  =====
INCREMENT BY -5                                   -55 -60 -65 -70 etc.
MINVALUE -1000
MAXVALUE +1000
NO CACHE
NO ORDER
CYCLE;
```

Figure 742, Alter sequence attributes

The only sequence attribute that one cannot change with the ALTER command is the field type that is used to hold the current value.

Constant Sequence

If the increment is zero, the sequence will stay whatever value one started it with until it is altered. This can be useful if wants to have a constant that can be globally referenced:

```
CREATE SEQUENCE biggest_sale_to_date             SEQ# VALUES ASSIGNED
AS INTEGER                                        =====
START WITH 345678                                 345678, 345678, etc.
INCREMENT BY 0;
```

Figure 743, Sequence that doesn't change

Getting the Sequence Value

There is no concept of a current sequence value. Instead one can either retrieve the next or the previous value (if there is one). And any reference to the next value will invariably cause the sequence to be incremented. The following example illustrates this:

```
CREATE SEQUENCE fred;                               ANSWER
COMMIT;                                             =====
                                                    SEQ#
                                                    ----
WITH templ (n1) AS
(VALUES 1
UNION ALL
SELECT n1 + 1
FROM templ
WHERE n1 < 5
)
SELECT NEXTVAL FOR fred AS seq#
FROM templ;
```

Figure 744, Selecting the NEXTVAL

NEXTVAL and PREVVAL -Usage Notes

- One retrieves the next or previous value using a "NEXTVAL FOR sequence-name", or a "PREVVAL for sequence-name" call.
- A NEXTVAL call generates and returns the next value in the sequence. Thus, each call will consume the returned value. This remains true even if the statement that did the retrieval subsequently fails or is rolled back.
- A PREVVAL call returns the most recently generated value for the specified sequence for the current connection. Unlike when getting the next value, getting the prior value does not alter the state of the sequence, so multiple calls can retrieve the same value.
- If no NEXTVAL reference (to the target sequence) has been made for the current connection, any attempt to get the PREVVAL will result in a SQL error.

NEXTVAL and PREVVAL - Usable Statements

- SELECT INTO statement (within the select part), as long as there is no DISTINCT, GROUP BY, UNION, EXECPT, or INTERSECT.
- INSERT statement - with restrictions.
- UPDATE statement - with restrictions.
- SET host variable statement.

NEXTVAL - Usable Statements

- A trigger.

NEXTVAL and PREVVAL - Not Allowed In

- DELETE statement.
- Join condition of a full outer join.
- Anywhere in a CREATE TABLE or CREATE VIEW statement.

NEXTVAL - Not Allowed In

- CASE expression
- Join condition of a join.
- Parameter list of an aggregate function.
- SELECT statement where there is an outer select that contains a DISTINCT, GROUP BY, UNION, EXCEPT, or INTERSECT.
- Most sub-queries.

PREVVAL - Not Allowed In

- A trigger.

There are many more usage restrictions, but you presumably get the picture. See the DB2 SQL Reference for the complete list.

Usage Examples

Below a sequence is defined, then various next and previous values are retrieved:

```

CREATE SEQUENCE fred;                                ANSWERS
COMMIT;                                              =====

WITH templ (prv) AS                                  ====>          PRV
(VALUES (PREVVAL FOR fred))                          ---
SELECT *                                             <error>
FROM templ;

WITH templ (nxt) AS                                  ====>          NXT
(VALUES (NEXTVAL FOR fred))                          ---
SELECT *                                             1
FROM templ;

WITH templ (prv) AS                                  ====>          PRV
(VALUES (PREVVAL FOR fred))                          ---
SELECT *                                             1
FROM templ;

WITH templ (n1) AS                                   ====>          NXT PRV
(VALUES 1                                             --- ---
 UNION ALL
 SELECT n1 + 1
 FROM templ
 WHERE n1 < 5
 )
SELECT NEXTVAL FOR fred AS nxt
      ,PREVVAL FOR fred AS prv
FROM templ;

```

Figure 745, Use of NEXTVAL and PREVVAL expressions

One does not actually have to fetch a NEXTVAL result in order to increment the underlying sequence. In the next example, some of the rows processed are thrown away halfway thru the query, but their usage still affects the answer (of the subsequent query):

```

CREATE SEQUENCE fred;                                ANSWERS
COMMIT;                                              =====

WITH templ AS                                        ====>          ID NXT
(SELECT id
      ,NEXTVAL FOR fred AS nxt
 FROM staff
 WHERE id < 100
 )
SELECT *
FROM templ
WHERE id = 50 + (nxt * 0);

WITH templ (nxt, prv) AS                             ====>          NXT PRV
(VALUES (NEXTVAL FOR fred
      ,PREVVAL FOR fred))                          --- ---
SELECT *
FROM templ;

```

Figure 746, NEXTVAL values used but not retrieved

NOTE: The somewhat funky predicate at the end of the first query above prevents DB2 from stopping the nested-table-expression when it gets to "id = 50". If this were to occur, the last query above would get a next value of 6, and a previous value of 5.

Multi-table Usage

Imagine that one wanted to maintain a unique sequence of values over multiple tables. One can do this by creating a before insert trigger on each table that replaces whatever value the user provides with the current one from a common sequence. Below is an example:

```

CREATE SEQUENCE cust#
  START WITH 1
  INCREMENT BY 1
  NO MAXVALUE
  NO CYCLE
  ORDER;

CREATE TABLE us_customer
(cust#          INTEGER          NOT NULL
 ,cname        CHAR(10)         NOT NULL
 ,frst_sale    DATE              NOT NULL
 ,#sales       INTEGER          NOT NULL
 ,PRIMARY KEY  (cust#));

CREATE TRIGGER us_cust_ins
NO CASCADE BEFORE INSERT ON us_customer
REFERENCING NEW AS nnn
FOR EACH ROW MODE DB2SQL
SET nnn.cust# = NEXTVAL FOR cust#;

CREATE TABLE intl_customer
(cust#          INTEGER          NOT NULL
 ,cname        CHAR(10)         NOT NULL
 ,frst_sale    DATE              NOT NULL
 ,#sales       INTEGER          NOT NULL
 ,PRIMARY KEY  (cust#));

CREATE TRIGGER intl_cust_ins
NO CASCADE BEFORE INSERT ON intl_customer
REFERENCING NEW AS nnn
FOR EACH ROW MODE DB2SQL
SET nnn.cust# = NEXTVAL FOR cust#;

```

Figure 747, Create tables that use a common sequence

If we now insert some rows into the above tables, we shall find that customer numbers are assigned in the correct order, thus:

```

SELECT  cust#
        ,cname
FROM    FINAL TABLE
(INsert INTO us_customer (cname, frst_sale, #sales)
VALUES ('FRED', '2002-10-22', 1)
        , ('JOHN', '2002-10-23', 1));

```

	ANSWERS
	=====
	CUST# CNAME

	1 FRED
	2 JOHN


```

SELECT  cust#
        ,cname
FROM    FINAL TABLE
(INsert INTO intl_customer (cname, frst_sale, #sales)
VALUES ('SUE', '2002-11-12', 2)
        , ('DEB', '2002-11-13', 2));

```

	CUST# CNAME

	3 SUE
	4 DEB

Figure 748, Insert into tables with common sequence

One of the advantages of a standalone sequence over a functionally similar identity column is that one can use a `PREVVAL` expression to get the most recent value assigned (to the user), even if the previous usage was during a multi-row insert. Thus, after doing the above inserts, we can run the following query:

```

WITH temp (prev) AS
(VALUES (PREVVAL FOR cust#))
SELECT *
FROM   temp;

```

	ANSWER
	=====
	PREV

	4

Figure 749, Get previous value - select

The following does the same as the above, but puts the result in a host variable:

```
VALUES PREVVAL FOR CUST# INTO :host-var
```

Figure 750, Get previous value - into host-variable

As with identity columns, the above result will not equal what is actually in the table(s) - if the most recent insert was subsequently rolled back.

Counting Deletes

In the next example, two sequences are created: One records the number of rows deleted from a table, while the other records the number of delete statements run against the same:

```
CREATE SEQUENCE delete_rows
  START WITH 1
  INCREMENT BY 1
  NO MAXVALUE
  NO CYCLE
  ORDER;

CREATE SEQUENCE delete_stmts
  START WITH 1
  INCREMENT BY 1
  NO MAXVALUE
  NO CYCLE
  ORDER;

CREATE TABLE customer
(cust#          INTEGER          NOT NULL
 ,cname        CHAR(10)         NOT NULL
 ,frst_sale    DATE              NOT NULL
 ,#sales       INTEGER          NOT NULL
 ,PRIMARY KEY  (cust#));

CREATE TRIGGER cust_del_rows
AFTER DELETE ON customer
FOR EACH ROW MODE DB2SQL
  WITH temp1 (n1) AS (VALUES(1))
  SELECT NEXTVAL FOR delete_rows
  FROM   temp1;

CREATE TRIGGER cust_del_stmts
AFTER DELETE ON customer
FOR EACH STATEMENT MODE DB2SQL
  WITH temp1 (n1) AS (VALUES(1))
  SELECT NEXTVAL FOR delete_stmts
  FROM   temp1;
```

Figure 751, Count deletes done to table

Be aware that the second trigger will be run, and thus will update the sequence, regardless of whether a row was found to delete or not.

Identity Columns vs. Sequences - a Comparison

First to compare the two types of sequences:

- Only one identity column is allowed per table, whereas a single table can have multiple sequences and/or multiple references to the same sequence.
- Identity column sequences cannot span multiple tables. Sequences can.
- Sequences require triggers to automatically maintain column values (e.g. during inserts) in tables. Identity columns do not.

- Sequences can be incremented during inserts, updates, deletes (via triggers), or selects, whereas identity columns only get incremented during inserts.
- Sequences can be incremented (via triggers) once per row, or once per statement. Identity columns are always updated per row inserted.
- Sequences can be dropped and created independent of any tables that they might be used to maintain values in. Identity columns are part of the table definition.
- Identity columns are supported by the load utility. Trigger induced sequences are not.

For both types of sequence, one can get the current value by embedding the DML statement inside a select (e.g. see figure 737). Alternatively, one can use the relevant expression to get the current status. These differ as follows:

- The `IDENTITY_VAL_LOCAL` function returns null if no inserts to tables with identity columns have been done by the current user. In an equivalent situation, the `PREVVAL` expression gets a nasty SQL error.
- The `IDENTITY_VAL_LOCAL` function ignores multi-row inserts (without telling you). In a similar situation, the `PREVVAL` expression returns the last value generated.
- One cannot tell to which table an `IDENTITY_VAL_LOCAL` function result refers to. This can be a problem in one insert invokes another insert (via a trigger), which puts a row in another table with its own identity column. By contrast, in the `PREVVAL` function one explicitly identifies the sequence to be read.
- There is no equivalent of the `NEXTVAL` expression for identity columns.

Roll Your Own

If one really, really, needs to have a sequence of values with no gaps, then one can do it using an insert trigger, but there are costs, in processing time, concurrency, and functionality. To illustrate, consider the following table:

```
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 752, 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 753, 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 754, Sample inserts, roll your own sequence#

Support Multi-row Inserts

The next design is more powerful in that it supports multi-row inserts, and also more than one table if desired. It requires that there be a central location that holds the current high-value. In the example below, this value will be in a row in a special control table. Every insert into the related data table will, via triggers, first update, and then query, the row in the control table.

Control Table

The following table has one row per sequence of values being maintained:

```
CREATE TABLE control_table
(table_name      CHAR(18)      NOT NULL
,table_nmbr     INTEGER        NOT NULL
,PRIMARY KEY (table_name));
```

Figure 755, Control Table, DDL

Now to populate the table with some initial sequence# values:

```
INSERT INTO control_table VALUES ('invoice_table', 0);
INSERT INTO control_table VALUES ('2nd_data_tble', 0);
INSERT INTO control_table VALUES ('3rd_data_tble', 0);
```

Figure 756, Control Table, sample inserts

Data Table

Our sample data table has two fields of interest:

- The UNQVAL column will be populated, using a trigger, with a GENERATE_UNIQUE function output value. This is done before the row is actually inserted. Once the insert has completed, we will no longer care about or refer to the contents of this field.
- The INVOICE# column will be populated, using triggers, during the insert process with a unique ascending value. However, for part of the time during the insert the field will have a null value, which is why it is defined as being both non-unique and allowing nulls.

```
CREATE TABLE invoice_table
(unqval          CHAR(13) FOR BIT DATA    NOT NULL
,invoice#       INTEGER
,sale_date      DATE                      NOT NULL
,customer_id    CHAR(20)                 NOT NULL
,product_id     INTEGER                   NOT NULL
,quantity       INTEGER                   NOT NULL
,price          DECIMAL(18,2)            NOT NULL
,PRIMARY KEY(unqval));
```

Figure 757, Sample Data Table, DDL

Two insert triggers are required: The first acts before the insert is done, giving each new row a unique UNQVAL value:

```
CREATE TRIGGER invoice1
NO CASCADE BEFORE INSERT ON invoice_table
REFERENCING NEW AS nnn
FOR EACH ROW MODE DB2SQL
SET nnn.unqval = GENERATE_UNIQUE()
,nnn.invoice# = NULL;
```

Figure 758, Before trigger

The second trigger acts after the row is inserted. It first increments the control table by one, then updates invoice# in the current row with the same value. The UNQVAL field is used to locate the row to be changed in the second update:

```
CREATE TRIGGER invoice2
AFTER INSERT ON invoice_table
REFERENCING NEW AS nnn
FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
UPDATE control_table
SET table_nmbr = table_nmbr + 1
WHERE table_name = 'invoice_table';
UPDATE invoice_table
SET invoice# =
(SELECT table_nmbr
FROM control_table
WHERE table_name = 'invoice_table')
WHERE unqval = nnn.unqval
AND invoice# IS NULL;
END
```

Figure 759, After trigger

NOTE: The above two actions must be in a single trigger. If they are in two triggers, mass inserts will not work correctly because the first trigger (i.e. update) would be run (for all rows), followed by the second trigger (for all rows). In the end, every row inserted by the mass-insert would end up with the same invoice# value.

A final update trigger is required to prevent updates to the invoice# column:


```

CREATE TRIGGER invoice3
NO CASCADE BEFORE UPDATE OF invoice# ON invoice_table
REFERENCING OLD AS ooo
           NEW AS nnn
FOR EACH ROW MODE DB2SQL
WHEN (ooo.invoice# <> nnn.invoice#)
      SIGNAL SQLSTATE '71001' ('no updates allowed - you twit');

```

Figure 760, Update trigger

Design Comments

Though the above design works, it has certain practical deficiencies:

- The single row in the control table is a point of contention, because only one user can update it at a time. One must therefore commit often (perhaps more often than one would like to) in order to free up the locks on this row. Therefore, by implication, this design puts one is at the mercy of programmers.
- The two extra updates add a considerable overhead to the cost of the insert.
- The invoice number values generated by AFTER trigger cannot be obtained by selecting from an insert statement (see page 67). In fact, selecting from the FINAL TABLE will result in a SQL error. One has to instead select from the NEW TABLE, which returns the new rows before the AFTER trigger was applied.

As with ordinary sequences, this design enables one to have multiple tables referring to a single row in the control table, and thus using a common sequence.

Temporary Tables

Introduction

How one defines a temporary table depends in part upon how often, and for how long, one intends to use it:

- Within a query, single use.
- Within a query, multiple uses.
- For multiple queries in one unit of work.
- For multiple queries, over multiple units of work, in one thread.

Single Use in Single Statement

If one intends to use a temporary table just once, it can be defined as a nested table expression. In the following example, we use a temporary table to sequence the matching rows in the STAFF table by descending salary. We then select the 2nd through 3rd rows:

```

SELECT  id
        ,salary
FROM    (SELECT  s.*
        ,ROW_NUMBER() OVER(ORDER BY salary DESC) AS sorder
        FROM    staff s
        WHERE   id < 200
        )AS xxx
WHERE   sorder BETWEEN 2 AND 3
ORDER BY id;

```

ANSWER	
=====	
ID	SALARY
---	-----
50	20659.80
140	21150.00

Figure 761, Nested Table Expression

NOTE: A fullselect in parenthesis followed by a correlation name (see above) is also called a nested table expression.

Here is another way to express the same:

```

WITH xxx (id, salary, sorder) AS
(SELECT  ID
        ,salary
        ,ROW_NUMBER() OVER(ORDER BY salary DESC) AS sorder
 FROM    staff
 WHERE   id < 200
 )
SELECT  id
        ,salary
FROM    xxx
WHERE   sorder BETWEEN 2 AND 3
ORDER BY id;

```

ANSWER	
=====	
ID	SALARY
---	-----
50	20659.80
140	21150.00

Figure 762, Common Table Expression

Multiple Use in Single Statement

Imagine that one wanted to get the percentage contribution of the salary in some set of rows in the STAFF table - compared to the total salary for the same. The only way to do this is to access the matching rows twice; Once to get the total salary (i.e. just one row), and then again to join the total salary value to each individual salary - to work out the percentage.

Selecting the same set of rows twice in a single query is generally unwise because repeating the predicates increases the likelihood of typos being made. In the next example, the desired rows are first placed in a temporary table. Then the sum salary is calculated and placed in another temporary table. Finally, the two temporary tables are joined to get the percentage:

```

WITH
rows_wanted AS
  (SELECT *
   FROM staff
   WHERE id < 100
        AND UCASE(name) LIKE '%T%'
  ),
sum_salary AS
  (SELECT SUM(salary) AS sum_sal
   FROM rows_wanted)
SELECT id
       ,name
       ,salary
       ,sum_sal
       ,INT((salary * 100) / sum_sal) AS pct
FROM   rows_wanted
       ,sum_salary
ORDER BY id;

```

```

ANSWER
=====
ID NAME      SALARY    SUM_SAL  PCT
-- ----
70 Rothman 16502.83 34504.58 47
90 Koonitz 18001.75 34504.58 52

```

Figure 763, Common Table Expression

Multiple Use in Multiple Statements

To refer to a temporary table in multiple SQL statements in the same thread, one has to define a declared global temporary table. An example follows:

```

DECLARE GLOBAL TEMPORARY TABLE session.fred
(dept          SMALLINT      NOT NULL
,avg_salary    DEC(7,2)     NOT NULL
,num_emps      SMALLINT     NOT NULL)
ON COMMIT PRESERVE ROWS;
COMMIT;

INSERT INTO session.fred
SELECT dept
       ,AVG(salary)
       ,COUNT(*)
FROM   staff
WHERE  id > 200
GROUP BY dept;
COMMIT;

```

```

ANSWER#1
=====
CNT
---
4

```

```

SELECT COUNT(*) AS cnt
FROM   session.fred;

DELETE FROM session.fred
WHERE  dept > 80;

SELECT *
FROM   session.fred;

```

```

ANSWER#2
=====
DEPT  AVG_SALARY  NUM_EMPS
----  -
10    20168.08     3
51    15161.43     3
66    17215.24     5

```

Figure 764, Declared Global Temporary Table

Unlike an ordinary table, a declared global temporary table is not defined in the DB2 catalogue. Nor is it sharable by other users. It only exists for the duration of the thread (or less) and can only be seen by the person who created it. For more information, see page 284.

Temporary Tables - in Statement

Three general syntaxes are used to define temporary tables in a query:

- Use a WITH phrase at the top of the query to define a common table expression.
- Define a fullselect in the FROM part of the query.
- Define a fullselect in the SELECT part of the query.

The following three queries, which are logically equivalent, illustrate the above syntax styles. Observe that the first two queries are explicitly defined as left outer joins, while the last one is implicitly a left outer join:

```

WITH staff_dept AS
(SELECT   dept           AS dept#
        ,MAX(salary) AS max_sal
  FROM   staff
 WHERE  dept < 50
 GROUP BY dept
)
SELECT   id
        ,dept
        ,salary
        ,max_sal
  FROM   staff
 LEFT OUTER JOIN
        staff_dept
 ON      dept = dept#
 WHERE  name LIKE 'S%'
 ORDER BY id;
ANSWER
=====
ID  DEPT  SALARY  MAX_SAL
---  ---  ---  ---
 10  20  18357.50  18357.50
 190 20  14252.75  18357.50
 200 42  11508.60  18352.80
 220 51  17654.50   -

```

Figure 765, Identical query (1 of 3) - using Common Table Expression

```

SELECT   id
        ,dept
        ,salary
        ,max_sal
  FROM   staff
 LEFT OUTER JOIN
        (SELECT   dept           AS dept#
        ,MAX(salary) AS max_sal
  FROM   staff
 WHERE  dept < 50
 GROUP BY dept
)AS STAFF_dept
 ON      dept = dept#
 WHERE  name LIKE 'S%'
 ORDER BY id;
ANSWER
=====
ID  DEPT  SALARY  MAX_SAL
---  ---  ---  ---
 10  20  18357.50  18357.50
 190 20  14252.75  18357.50
 200 42  11508.60  18352.80
 220 51  17654.50   -

```

Figure 766, Identical query (2 of 3) - using fullselect in FROM

```

SELECT   id
        ,dept
        ,salary
        ,(SELECT   MAX(salary)
  FROM   staff s2
 WHERE  s1.dept = s2.dept
        AND s2.dept < 50
 GROUP BY dept)
        AS max_sal
  FROM   staff s1
 WHERE  name LIKE 'S%'
 ORDER BY id;
ANSWER
=====
ID  DEPT  SALARY  MAX_SAL
---  ---  ---  ---
 10  20  18357.50  18357.50
 190 20  14252.75  18357.50
 200 42  11508.60  18352.80
 220 51  17654.50   -

```

Figure 767, Identical query (3 of 3) - using fullselect 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.

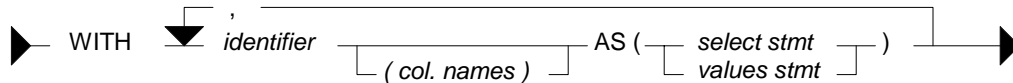


Figure 768, Common Table Expression Syntax

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 templ AS
  (SELECT MAX(name) AS max_name
   ,MAX(dept) AS max_dept
   FROM staff
  )
SELECT *
FROM templ;
```

ANSWER	
=====	
MAX_NAME	MAX_DEPT

Yamaguchi	84

Figure 769, 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 templ (max_name,max_dept) AS
  (SELECT MAX(name)
   ,MAX(dept)
   FROM staff
  )
SELECT *
FROM templ;
```

ANSWER	
=====	
MAX_NAME	MAX_DEPT

Yamaguchi	84

Figure 770, 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
temp1 AS
  (SELECT   dept
           ,AVG(salary) AS avg_sal
    FROM    staff
    GROUP BY dept),
temp2 AS
  (SELECT   MAX(avg_sal) AS max_avg
    FROM    temp1)
SELECT *
FROM    temp2;

```

ANSWER
=====
MAX_AVG

20865.8625

Figure 771, Query with two common table expressions

FYI, the exact same query can be written using nested table expressions thus:

```

SELECT *
FROM    (SELECT MAX(avg_sal) AS max_avg
        FROM    (SELECT dept
                ,AVG(salary) AS avg_sal
                FROM    staff
                GROUP BY dept
                )AS temp1
        )AS temp2;

```

ANSWER
=====
MAX_AVG

20865.8625

Figure 772, 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
  (SELECT   id
           ,name
           ,dept
           ,salary
    FROM    staff
    WHERE   id < 300
           AND dept <> 55
           AND name LIKE 'S%'
           AND dept NOT IN
             (SELECT deptnumb
              FROM    org
              WHERE   division = 'SOUTHERN'
                    OR location = 'HARTFORD')
    )
,temp2 AS
  (SELECT   dept
           ,MAX(salary) AS max_sal
    FROM    temp1
    GROUP BY dept
    )
SELECT   t1.id
        ,t1.dept
        ,t1.salary
        ,t2.max_sal
FROM    temp1 t1
        ,temp2 t2
WHERE   t1.dept = t2.dept
ORDER BY t1.id;

```

ANSWER			
=====			
ID	DEPT	SALARY	MAX_SAL

10	20	18357.50	18357.50
190	20	14252.75	18357.50
200	42	11508.60	11508.60
220	51	17654.50	17654.50

Figure 773, 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 774, 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 775, Equivalent insert (to above) without common table expression

Full-Select

A fullselect is an alternative way to define a temporary table. Instead of using a WITH clause at the top of the statement, the temporary table definition is embedded in the body of the SQL statement. Certain rules apply:

- When used in a select statement, a fullselect can either be generated in the FROM part of the query - where it will return a temporary table, or in the SELECT part of the query - where it will return a column of data.
- When the result of a fullselect is a temporary table (i.e. in FROM part of a query), the table must be provided with a correlation name.
- When the result of a fullselect is a column of data (i.e. in SELECT part of query), each reference to the temporary table must only return a single value.

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
        WHERE   dept = deptnumb
        GROUP BY division
          )AS xxx
GROUP BY division;
```

ANSWER			
DIVISION	DIV_DEPT	#DPTS	#EMPS
Corporate	20865.86	1	4
Eastern	15670.32	3	13
Midwest	15905.21	2	9
Western	16875.99	2	9

Figure 776, Nested column function usage

The next query illustrates how multiple fullselects can be nested inside each other:


```

SELECT id
FROM (SELECT *
      FROM (SELECT id, years, salary
            FROM (SELECT *
                  FROM staff
                  WHERE dept < 77
                  )AS t1
            WHERE id < 300
            )AS t2
      WHERE job LIKE 'C%'
      )AS t3
  WHERE salary < 18000
  )AS t4
WHERE years < 5;

```

ANSWER
=====
ID

170
180
230

Figure 777, Nested fullselects

A very common usage of a fullselect is to join a derived table to a real table. In the following example, the average salary for each department is joined to the individual staff row:

```

SELECT  a.id
        ,a.dept
        ,a.salary
        ,DEC(b.avgсал,7,2) AS avg_dept
FROM    staff a
LEFT OUTER JOIN
        (SELECT  dept AS dept
              ,AVG(salary) AS avgсал
        FROM    staff
        GROUP BY dept
        HAVING  AVG(salary) > 16000
        )AS b
ON      a.dept = b.dept
WHERE   a.id < 40
ORDER BY a.id;

```

ANSWER
=====
ID DEPT SALARY AVG_DEPT
--- ---- -
10 20 18357.50 16071.52
20 20 18171.25 16071.52
30 38 17506.75 -

Figure 778, Join fullselect to real table

Table Function Usage

If the fullselect query has a reference to a row in a table that is outside of the fullselect, then it needs to be written as a TABLE function call. In the next example, the preceding "A" table is referenced in the fullselect, and so the TABLE function call is required:

```

SELECT  a.id
        ,a.dept
        ,a.salary
        ,b.deptsal
FROM    staff a
        ,TABLE
        (SELECT  b.dept
              ,SUM(b.salary) AS deptsal
        FROM    staff b
        WHERE   b.dept = a.dept
        GROUP BY b.dept
        )AS b
WHERE   a.id < 40
ORDER BY a.id;

```

ANSWER
=====
ID DEPT SALARY DEPTSAL
--- ---- -
10 20 18357.50 64286.10
20 20 18171.25 64286.10
30 38 17506.75 77285.55

Figure 779, Fullselect with external table reference

Below is the same query written without the reference to the "A" table in the fullselect, and thus without a TABLE function call:

```

SELECT      a.id
            ,a.dept
            ,a.salary
            ,b.deptsal
FROM        staff a
            ,(SELECT  b.dept
                   ,SUM(b.salary) AS deptsal
                   FROM    staff b
                   GROUP BY b.dept
                  )AS b
WHERE       a.id < 40
            AND    b.dept = a.dept
ORDER BY   a.id;

```

ANSWER			
ID	DEPT	SALARY	DEPTSAL
10	20	18357.50	64286.10
20	20	18171.25	64286.10
30	38	17506.75	77285.55

Figure 780, Fullselect without external table reference

Any externally referenced table in a fullselect must be defined in the query syntax (starting at the first FROM statement) before the fullselect. Thus, in the first example above, if the "A" table had been listed after the "B" table, then the query would have been invalid.

Full-Select in SELECT Phrase

A fullselect that returns a single column and row can be used in the SELECT part of a query:

```

SELECT      id
            ,salary
            ,(SELECT MAX(salary)
              FROM    staff
              ) AS maxsal
FROM        staff a
WHERE       id < 60
ORDER BY   id;

```

ANSWER		
ID	SALARY	MAXSAL
10	18357.50	22959.20
20	18171.25	22959.20
30	17506.75	22959.20
40	18006.00	22959.20
50	20659.80	22959.20

Figure 781, Use an uncorrelated Full-Select in a SELECT list

A fullselect in the SELECT part of a statement must return only a single row, but it need not always be the same row. In the following example, the ID and SALARY of each employee is obtained - along with the max SALARY for the employee's department.

```

SELECT      id
            ,salary
            ,(SELECT MAX(salary)
              FROM    staff b
              WHERE   a.dept = b.dept
              ) AS maxsal
FROM        staff a
WHERE       id < 60
ORDER BY   id;

```

ANSWER		
ID	SALARY	MAXSAL
10	18357.50	18357.50
20	18171.25	18357.50
30	17506.75	18006.00
40	18006.00	18006.00
50	20659.80	20659.80

Figure 782, Use a correlated Full-Select in a SELECT list

```

SELECT      id
            ,dept
            ,salary
            ,(SELECT MAX(salary)
              FROM    staff b
              WHERE   b.dept = a.dept)
            ,(SELECT MAX(salary)
              FROM    staff)
FROM        staff a
WHERE       id < 60
ORDER BY   id;

```

ANSWER				
ID	DEPT	SALARY	4	5
10	20	18357.50	18357.50	22959.20
20	20	18171.25	18357.50	22959.20
30	38	17506.75	18006.00	22959.20
40	38	18006.00	18006.00	22959.20
50	15	20659.80	20659.80	22959.20

Figure 783, Use correlated and uncorrelated Full-Selects in a SELECT list

INSERT Usage

The following query uses both an uncorrelated and correlated fullselect 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 784, Fullselect in INSERT

UPDATE Usage

The following example uses an uncorrelated fullselect to assign a set of workers the average salary in the company - plus two thousand dollars.

```

UPDATE staff a
SET salary =
  (SELECT AVG(salary)+ 2000
   FROM staff)
WHERE id < 60;

```

ANSWER:		SALARY	
ID	DEPT	BEFORE	AFTER
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 785, Use uncorrelated Full-Select to give workers company AVG salary (+\$2000)

The next statement uses a correlated fullselect to assign a set of workers the average salary for their department - plus two thousand dollars. Observe that when there is more than one worker in the same department, that they all get the same new salary. This is because the fullselect is resolved before the first update was done, not after each.

```

UPDATE staff a
SET salary =
  (SELECT AVG(salary) + 2000
   FROM staff b
   WHERE a.dept = b.dept )
WHERE id < 60;

```

ANSWER:		SALARY	
ID	DEPT	BEFORE	AFTER
10	20	18357.50	18071.52
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 786, Use correlated Full-Select to give workers department AVG salary (+\$2000)

NOTE: A fullselect is always resolved just once. If it is queried using a correlated expression, then the data returned each time may differ, but the table remains unchanged.

The next update is the same as the prior, except that two fields are changed:

```

UPDATE staff a
SET (salary,years) =
  (SELECT AVG(salary) + 2000
   ,MAX(years)
   FROM staff b
   WHERE a.dept = b.dept )
WHERE id < 60;

```

Figure 787, Update two fields by referencing Full-Select

Declared Global Temporary Tables

If we want to temporarily retain some rows for processing by subsequent SQL statements, we can use a Declared Global Temporary Table. A temporary table only exists until the thread is terminated (or sooner). It is not defined in the DB2 catalogue, and neither its definition nor its contents are visible to other users. Multiple users can declare the same temporary table at the same time. Each will be independently working with their own copy.

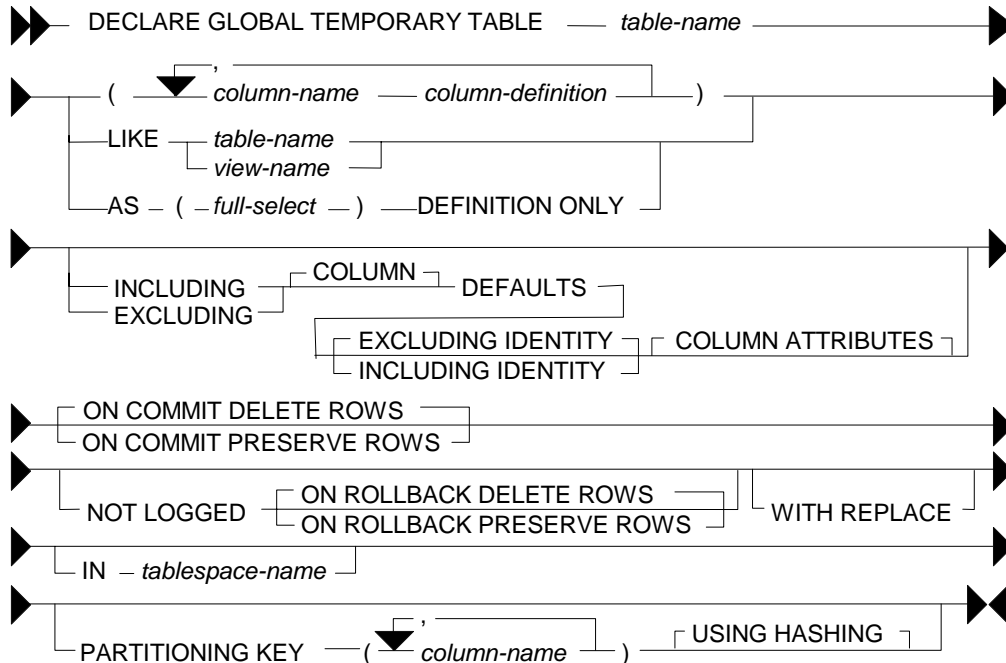


Figure 788, Declared Global Temporary Table syntax

Usage Notes

For a complete description of this feature, see the SQL reference. Below are some key points:

- The temporary table name can be any valid DB2 table name. The table qualifier, if provided, must be `SESSION`. If the qualifier is not provided, it is assumed to be `SESSION`.
- If the temporary table has been previously defined in this session, the `WITH REPLACE` clause can be used to override it. Alternatively, one can `DROP` the prior instance.
- An index can be defined on a global temporary table. The qualifier (i.e. `SESSION`) must be explicitly provided.
- Any column type can be used in the table, except for: `BLOB`, `CLOB`, `DBCLOB`, `LONG VARCHAR`, `LONG VARGRAPHIC`, `DATALINK`, reference, and structured data types.
- One can choose to preserve or delete (the default) the rows in the table when a commit occurs. Deleting the rows does not drop the table.
- Standard identity column definitions can be used if desired.
- Changes are not logged.

Sample SQL

Below is an example of declaring a global temporary table by listing the columns:

```
DECLARE GLOBAL TEMPORARY TABLE session.fred
(dept          SMALLINT      NOT NULL
,avg_salary   DEC(7,2)      NOT NULL
,num_emps     SMALLINT      NOT NULL)
ON COMMIT DELETE ROWS;
```

Figure 789, Declare Global Temporary Table - define columns

In the next example, the temporary table is defined to have exactly the same columns as the existing STAFF table:

```
DECLARE GLOBAL TEMPORARY TABLE session.fred
LIKE staff INCLUDING COLUMN DEFAULTS
WITH REPLACE
ON COMMIT PRESERVE ROWS;
```

Figure 790, Declare Global Temporary Table - like another table

In the next example, the temporary table is defined to have a set of columns that are returned by a particular select statement. The statement is not actually run at definition time, so any predicates provided are irrelevant:

```
DECLARE GLOBAL TEMPORARY TABLE session.fred AS
(SELECT   dept
        ,MAX(id)      AS max_id
        ,SUM(salary) AS sum_sal
FROM     staff
WHERE    name <> 'IDIOT'
GROUP BY dept)
DEFINITION ONLY
WITH REPLACE;
```

Figure 791, Declare Global Temporary Table - like query output

Indexes can be added to temporary tables in order to improve performance and/or to enforce uniqueness:

```
DECLARE GLOBAL TEMPORARY TABLE session.fred
LIKE staff INCLUDING COLUMN DEFAULTS
WITH REPLACE ON COMMIT DELETE ROWS;

CREATE UNIQUE INDEX session.fredx ON Session.fred (id);

INSERT INTO session.fred
SELECT *
FROM   staff
WHERE  id < 200;

SELECT COUNT(*)
FROM   session.fred;

COMMIT;

SELECT COUNT(*)
FROM   session.fred;
```

ANSWER
=====

19

ANSWER
=====

0

Figure 792, Temporary table with index

A temporary table has to be dropped to reuse the same name:

```

DECLARE GLOBAL TEMPORARY TABLE session.fred
(dept          SMALLINT    NOT NULL
,avg_salary   DEC(7,2)    NOT NULL
,num_emps     SMALLINT    NOT NULL)
ON COMMIT DELETE ROWS;

INSERT INTO session.fred
SELECT  dept
        ,AVG(salary)
        ,COUNT(*)
FROM    staff
GROUP BY dept;

SELECT  COUNT(*)
FROM    session.fred;

DROP TABLE session.fred;

DECLARE GLOBAL TEMPORARY TABLE session.fred
(dept          SMALLINT    NOT NULL)
ON COMMIT DELETE ROWS;

SELECT  COUNT(*)
FROM    session.fred;

```

ANSWER
=====
8

ANSWER
=====
0

Figure 793, Dropping a temporary table

Tablespace

Before a user can create a declared global temporary table, a `USER TEMPORARY` tablespace that they have access to, has to be created. A typical definition follows:

```

CREATE USER TEMPORARY TABLESPACE FRED
MANAGED BY DATABASE
USING (FILE 'C:\DB2\TEMPFRED\FRED1' 1000
       ,FILE 'C:\DB2\TEMPFRED\FRED2' 1000
       ,FILE 'C:\DB2\TEMPFRED\FRED3' 1000);

```

```
GRANT USE OF TABLESPACE FRED TO PUBLIC;
```

Figure 794, Create USER TEMPORARY tablespace

Do NOT use to Hold Output

In general, do not use a Declared Global Temporary Table to hold job output data, especially if the table is defined `ON COMMIT PRESERVE ROWS`. If the job fails halfway through, the contents of the temporary table will be lost. If, prior to the failure, the job had updated and then committed Production data, it may be impossible to recreate the lost output because the committed rows cannot be updated twice.

Recursive SQL

Recursive SQL enables one to efficiently resolve all manner of complex logical structures that can be really tough to work with using other techniques. On the down side, it is a little tricky to understand at first and it is occasionally expensive. In this chapter we shall first show how recursive SQL works and then illustrate some of the really cute things that one use it for.

Use Recursion To

- Create sample data.
- Select the first "n" rows.
- Generate a simple parser.
- Resolve a Bill of Materials hierarchy.
- Normalize and/or denormalize data structures.

When (Not) to Use Recursion

A good SQL statement is one that gets the correct answer, is easy to understand, and is efficient. Let us assume that a particular statement is correct. If the statement uses recursive SQL, it is never going to be categorized as easy to understand (though the reading gets much easier with experience). However, given the question being posed, it is possible that a recursive SQL statement is the simplest way to get the required answer.

Recursive SQL statements are neither inherently efficient nor inefficient. Because they often involve a join, it is very important that suitable indexes be provided. Given appropriate indexes, it is quite probable that a recursive SQL statement is the most efficient way to resolve a particular business problem. It all depends upon the nature of the question: If every row processed by the query is required in the answer set (e.g. Find all people who work for Bob), then a recursive statement is likely to very efficient. If only a few of the rows processed by the query are actually needed (e.g. Find all airline flights from Boston to Dallas, then show only the five fastest) then the cost of resolving a large data hierarchy (or network), most of which is immediately discarded, can be very prohibitive.

If one wants to get only a small subset of rows in a large data structure, it is very important that of the unwanted data is excluded as soon as possible in the processing sequence. Some of the queries illustrated in this chapter have some rather complicated code in them to do just this. Also, always be on the lookout for infinitely looping data structures.

Conclusion

Recursive SQL statements can be very efficient, if coded correctly, and if there are suitable indexes. When either of the above is not true, they can be very slow.

How Recursion Works

Below is a description of a very simple application. The table on the left contains a normalized representation of the hierarchical structure on the right. Each row in the table defines a relationship displayed in the hierarchy. The PKEY field identifies a parent key, the CKEY

field has related child keys, and the NUM field has the number of times the child occurs within the related parent.

HIERARCHY		
PKEY	CKEY	NUM
AAA	BBB	1
AAA	CCC	5
AAA	DDD	20
CCC	EEE	33
DDD	EEE	44
DDD	FFF	5
FFF	GGG	5

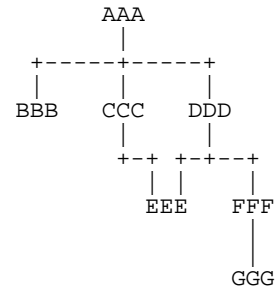


Figure 795, Sample Table description - Recursion

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
  (SELECT pkey, ckey
   FROM hierarchy
   WHERE pkey = 'AAA'
  UNION ALL
   SELECT C.pkey, C.ckey
   FROM hierarchy C
        ,parent P
   WHERE P.ckey = C.pkey
  )
SELECT pkey, ckey
FROM parent;
  
```

ANSWER			PROCESSING
=====			SEQUENCE
PKEY	CKEY		=====

AAA	BBB		< 1st pass
AAA	CCC		" "
AAA	DDD		" "
CCC	EEE		< 2nd pass
DDD	EEE		< 3rd pass
DDD	FFF		" "
FFF	GGG		< 4th pass

Figure 796, SQL that does Recursion

The above statement is best described by decomposing it into its individual components, and then following of sequence of events that occur:

- The WITH statement at the top defines a temporary table called PARENT.
- The upper part of the UNION ALL is only invoked once. It does an initial population of the PARENT table with the three rows that have an immediate parent key of AAA .
- The lower part of the UNION ALL is run recursively until there are no more matches to the join. In the join, the current child value in the temporary PARENT table is joined to related parent values in the DATA table. Matching rows are placed at the front of the temporary PARENT table. This recursive processing will stop when all of the rows in the PARENT table have been joined to the DATA table.
- The SELECT phrase at the bottom of the statement sends the contents of the PARENT table back to the user's program.

Another way to look at the above process is to think of the temporary PARENT table as a stack of data. This stack is initially populated by the query in the top part of the UNION ALL. Next, a cursor starts from the bottom of the stack and goes up. Each row obtained by the cursor is joined to the DATA table. Any matching rows obtained from the join are added to the top of the stack (i.e. in front of the cursor). When the cursor reaches the top of the stack, the statement is done. The following diagram illustrates this process:

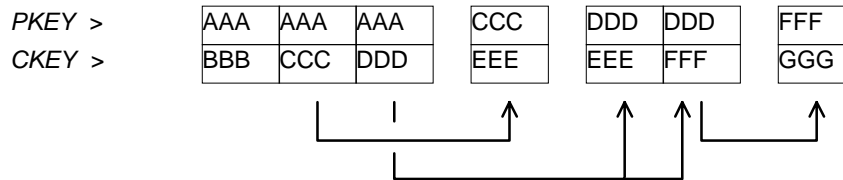


Figure 797, Recursive processing sequence

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.
- If done right, recursive SQL is often fairly efficient. When it involves a join similar to the example shown above, it is important to make sure that this join is efficient. To this end, suitable indexes should be provided.
- The output of a recursive SQL is a temporary table (usually). Therefore, all temporary table usage restrictions also apply to recursive SQL output. See the section titled "Common Table Expression" for details.
- The output of one recursive expression can be used as input to another recursive expression in the same SQL statement. This can be very handy if one has multiple logical hierarchies to traverse (e.g. First find all of the states in the USA, then find 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 298.

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 798, 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 289. 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.

<pre> WITH parent (ckey) AS (SELECT ckey FROM hierarchy WHERE pkey = 'AAA' UNION ALL SELECT C.ckey FROM hierarchy C ,parent P WHERE P.ckey = C.pkey) SELECT ckey FROM parent;</pre>	<pre> ANSWER ===== CKEY ----- BBB CCC DDD EEE EEE FFF GGG</pre>	<pre> HIERARCHY +-----+ PKEY CKEY NUM +-----+ AAA BBB 1 AAA CCC 5 AAA DDD 20 CCC EEE 33 DDD EEE 44 DDD FFF 5 FFF GGG 5 +-----+</pre>
--	---	---

Figure 799, 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 298 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
  (SELECT DISTINCT pkey
   FROM hierarchy
   WHERE pkey = 'AAA'
   UNION ALL
   SELECT C.ckey
   FROM hierarchy C
        ,parent P
   WHERE P.ckey = C.pkey
  )
SELECT ckey
FROM parent;

```

ANSWER	HIERARCHY		
=====	+-----+		
CKEY	PKEY	CKEY	NUM
----	----	----	----
AAA	AAA	BBB	1
BBB	AAA	CCC	5
CCC	AAA	DDD	20
DDD	CCC	EEE	33
EEE	DDD	EEE	44
EEE	DDD	FFF	5
FFF	FFF	GGG	5
GGG	+-----+		

Figure 800, 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
  (SELECT DISTINCT pkey
   FROM hierarchy
   WHERE pkey = 'AAA'
   UNION ALL
   SELECT C.ckey
   FROM hierarchy C
        ,parent P
   WHERE P.ckey = C.pkey
  )
SELECT DISTINCT ckey
FROM parent;

```

ANSWER	HIERARCHY		
=====	+-----+		
CKEY	PKEY	CKEY	NUM
----	----	----	----
AAA	AAA	BBB	1
BBB	AAA	CCC	5
CCC	AAA	DDD	20
DDD	CCC	EEE	33
EEE	DDD	EEE	44
FFF	DDD	FFF	5
GGG	FFF	GGG	5
	+-----+		

Figure 801, 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
  (SELECT DISTINCT pkey
   FROM hierarchy
   WHERE pkey = 'AAA'
   UNION ALL
   SELECT C.ckey
   FROM hierarchy C
        ,parent P
   WHERE P.ckey = C.pkey
  ),
distinct_parent (ckey) AS
  (SELECT DISTINCT ckey
   FROM parent
  )
SELECT ckey
FROM distinct_parent;

```

ANSWER	HIERARCHY		
=====	+-----+		
CKEY	PKEY	CKEY	NUM
----	----	----	----
AAA	AAA	BBB	1
BBB	AAA	CCC	5
CCC	AAA	DDD	20
DDD	CCC	EEE	33
EEE	DDD	EEE	44
FFF	DDD	FFF	5
GGG	FFF	GGG	5
	+-----+		

Figure 802, 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
  (SELECT DISTINCT pkey, 0
   FROM hierarchy
   WHERE pkey = 'AAA'
   UNION ALL
   SELECT C.ckey, P.lvl +1
   FROM hierarchy C
        ,parent P
   WHERE P.ckey = C.pkey
  )
SELECT ckey, lvl
FROM parent;
```

ANSWER		AAA		
=====				
CKEY LVL		+-----+-----+		

AAA 0		BBB	CCC	DDD
BBB 1				
CCC 1			+--+ +--+ +--+	
DDD 1				
EEE 2			EEE	FFF
EEE 2				
FFF 2				GGG
GGG 3				

Figure 803, 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 is 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
  (SELECT DISTINCT pkey, 0
   FROM hierarchy
   WHERE pkey = 'AAA'
   UNION ALL
   SELECT C.ckey, P.lvl +1
   FROM hierarchy C
        ,parent P
   WHERE P.ckey = C.pkey
  )
SELECT ckey, lvl
FROM parent
WHERE lvl < 3;
```

ANSWER		HIERARCHY		
=====		+-----+-----+		
CKEY LVL		PKEY	CKEY	NUM
----		----	----	----
AAA 0		AAA	BBB	1
BBB 1		AAA	CCC	5
CCC 1		AAA	DDD	20
DDD 1		CCC	EEE	33
EEE 2		DDD	EEE	44
EEE 2		DDD	FFF	5
FFF 2		FFF	GGG	5
		+-----+-----+		

Figure 804, 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
  (SELECT DISTINCT pkey, 0
   FROM hierarchy
   WHERE pkey = 'AAA'
   UNION ALL
   SELECT C.ckey, P.lvl +1
   FROM hierarchy C
        ,parent P
   WHERE P.ckey = C.pkey
        AND P.lvl+1 < 3
  )
SELECT ckey, lvl
FROM parent;
```

ANSWER		AAA		
=====				
CKEY LVL		+-----+-----+		

AAA 0		BBB	CCC	DDD
BBB 1				
CCC 1			+--+ +--+ +--+	
DDD 1				
EEE 2			EEE	FFF
EEE 2				
FFF 2				GGG

Figure 805, Select rows where LEVEL < 3

The only difference between this statement and the one before is that the level check is now done in the recursive part of the statement. This new level-check predicate has a dual function: It gives us the answer that we want, and it stops the SQL from running forever if the database happens to contain an infinite loop (e.g. DDD was also a parent of AAA).

One problem with this general statement design is that it can not be used to list only that data which pertains to a certain lower level (e.g. display only level 3 data). To answer this kind of question efficiently we can combine the above two queries, having appropriate predicates in both places (see next).

Select Explicit Level

Get a list of all the children of AAA that are exactly two levels below AAA.

```
WITH parent (ckey, lvl) AS
  (SELECT DISTINCT pkey, 0
   FROM hierarchy
   WHERE pkey = 'AAA'
   UNION ALL
   SELECT C.ckey, P.lvl +1
   FROM hierarchy C
        ,parent P
   WHERE P.ckey = C.pkey
        AND P.lvl+1 < 3
  )
SELECT ckey, lvl
FROM parent
WHERE lvl = 2;
```

ANSWER		HIERARCHY		
=====		+-----+-----+		
CKEY	LVL	PKEY	CKEY	NUM
-----		-----		
EEE	2	AAA	BBB	1
EEE	2	AAA	CCC	5
FFF	2	AAA	DDD	20
		CCC	EEE	33
		DDD	EEE	44
		DDD	FFF	5
		FFF	GGG	5
		+-----+-----+		

Figure 806, Select rows where LEVEL = 2

In the recursive part of the above statement all of the levels up to and including that which is required are obtained. All undesired lower levels are then removed in the final select.

Trace a Path - Use Multiple Recursions

Multiple recursive joins can be included in a single query. The joins can run independently, or the output from one recursive join can be used as input to a subsequent. Such code enables one to do the following:

- Expand multiple hierarchies in a single query. For example, one might first get a list of all departments (direct and indirect) in a particular organization, and then use the department list as a seed to find all employees (direct and indirect) in each department.
- Go down, and then up, a given hierarchy in a single query. For example, one might want to find all of the children of AAA, and then all of the parents. The combined result is the list of objects that AAA is related to via a direct parent-child path.
- Go down the same hierarchy twice, and then combine the results to find the matches, or the non-matches. This type of query might be used to, for example, see if two companies own shares in the same subsidiary.

The next example recursively searches the HIERARCHY table for all values that are either a child or a parent (direct or indirect) of the object DDD. The first part of the query gets the list of children, the second part gets the list of parents (but never the value DDD itself), and then the results are combined.

```

WITH children (kkey, lvl) AS
  (SELECT ckey, 1
   FROM hierarchy
   WHERE pkey = 'DDD'
  UNION ALL
   SELECT H.ckey, C.lvl + 1
   FROM hierarchy H
        ,children C
   WHERE H.pkey = C.kkey
  )
,parents (kkey, lvl) AS
  (SELECT pkey, -1
   FROM hierarchy
   WHERE ckey = 'DDD'
  UNION ALL
   SELECT H.pkey, P.lvl - 1
   FROM hierarchy H
        ,parents P
   WHERE H.ckey = P.kkey
  )
SELECT kkey ,lvl
FROM children
UNION ALL
SELECT kkey ,lvl
FROM parents;

```

```

ANSWER
=====
KKEY LVL
-----
AAA   -1
EEE    1
FFF    1
GGG    2

```

Figure 807, Find all children and parents of DDD

Extraneous Warning Message

Some recursive SQL statements generate the following warning when the DB2 parser has reason to suspect that the statement may run forever:

```
SQL0347W The recursive common table expression "GRAEME.TEMP1" may contain an
infinite loop. SQLSTATE=01605
```

The text that accompanies this message provides detailed instructions on how to code recursive SQL so as to avoid getting into an infinite loop. The trouble is that even if you do exactly as told you may still get the silly message. To illustrate, the following two SQL statements are almost identical. Yet the first gets a warning and the second does not:

```

WITH temp1 (n1) AS
  (SELECT id
   FROM staff
   WHERE id = 10
  UNION ALL
   SELECT n1 +10
   FROM temp1
   WHERE n1 < 50
  )
SELECT *
FROM temp1;

```

```

ANSWER
=====
N1
--
warn
10
20
30
40
50

```

Figure 808, Recursion - with warning message

```

WITH temp1 (n1) AS
  (SELECT INT(id)
   FROM staff
   WHERE id = 10
  UNION ALL
   SELECT n1 +10
   FROM temp1
   WHERE n1 < 50
  )
SELECT *
FROM temp1;

```

```

ANSWER
=====
N1
--
10
20
30
40
50

```

Figure 809, 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.

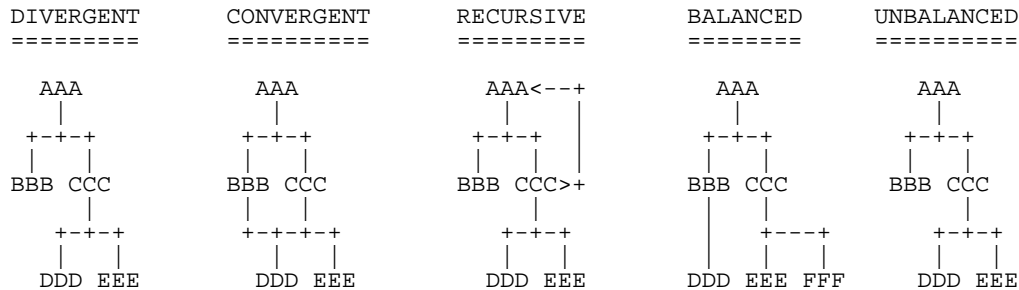


Figure 810, Hierarchy Flavours

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

Figure 811, Divergent Hierarchy - Table and Layout

Some database designers like to make the arbitrary judgment that every object has a parent, and in those cases where there is no "real" parent, the object considered to be a parent of itself. In the above table, this would mean that AAA would be defined as a parent of AAA. Please appreciate that this judgment call does not affect the objects that the database represents, but it can have a dramatic impact on SQL usage and performance.

Prior to the introduction of recursive SQL, defining top level objects as being self-parenting was sometimes a good idea because it enabled one to resolve a hierarchy using a simple join without unions. This same process is now best done with recursive SQL. Furthermore, if objects in the database are defined as self-parenting, the recursive SQL will get into an infinite loop unless extra predicates are provided.

Convergent Hierarchy

NUMBER OF TABLES: A convergent hierarchy has many-to-many relationships that require two tables for normalized data storage. The other hierarchy types require but a single table.

In this flavour of hierarchy, each object can have none, one, or more than one, parent and/or dependent child objects. Convergent hierarchies are often much more difficult to work with than similar divergent hierarchies. Logical entities, or man-made objects, (e.g. Company Divisions) often have this type of hierarchy.

Two tables are required in order to represent this kind of hierarchy in a fully normalized form. One table describes the object, and the other describes the relationships between the objects.

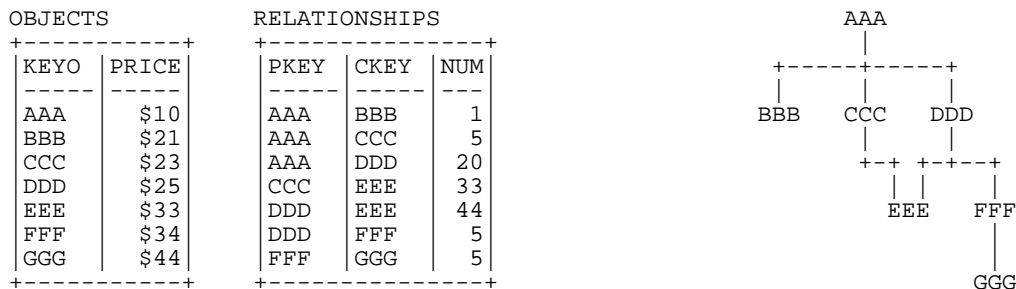


Figure 812, Convergent Hierarchy - Tables and Layout

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 298 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.

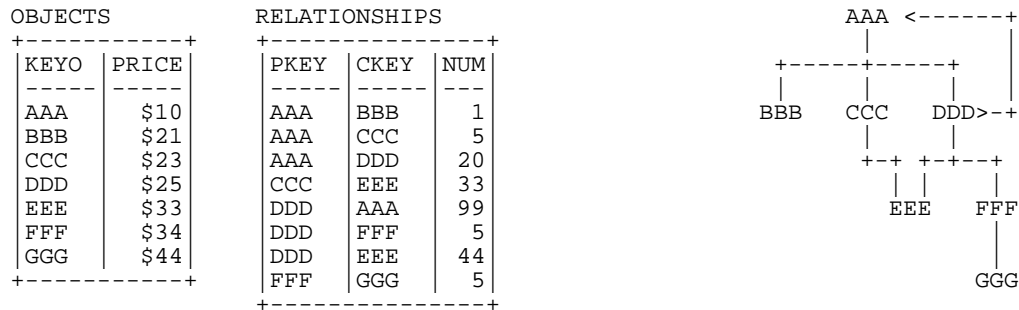


Figure 813, Recursive Hierarchy - Tables and Layout

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 298 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.

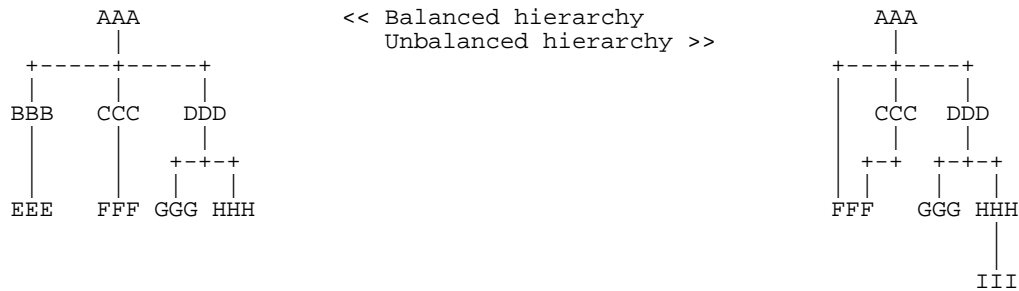


Figure 814, Balanced and Unbalanced Hierarchies

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-assemblies (e.g. a bolt connecting to engine to the wing) must not be counted twice, missed out, nor assigned to the wrong sub-grouping. As always, such questions are essentially easy to answer, the trick is to get the right answer.

Halting Recursive Processing

One occasionally encounters recursive hierarchical data structures (i.e. where the parent item points to the child, which then points back to the parent). This section describes how to write recursive SQL statements that can process such structures without running forever. There are three general techniques that one can use:

- Stop processing after reaching a certain number of levels.
- Keep a record of where you have been, and if you ever come back, either fail or in some other way stop recursive processing.
- Keep a record of where you have been, and if you ever come back, simply ignore that row and keep on resolving the rest of hierarchy.

Sample Table DDL & DML

The following table is a normalized representation of the recursive hierarchy on the right. Note that AAA and DDD are both a parent and a child of each other.

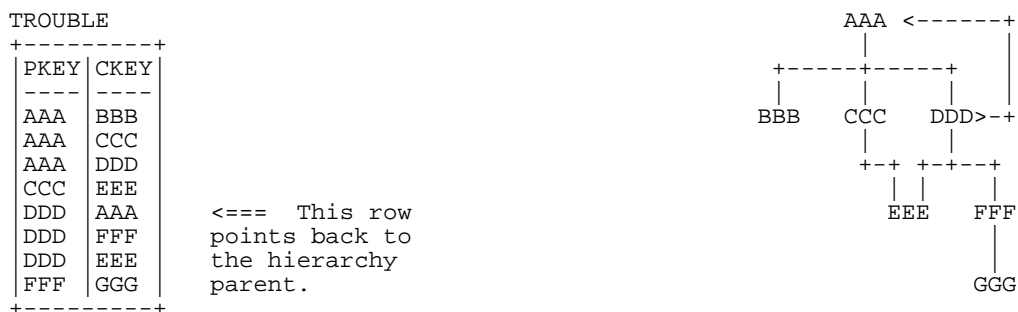


Figure 815, Recursive Hierarchy - Sample Table and Layout

Below is the DDL and DML that was used to create the above table.

```

CREATE TABLE trouble
(pkey      CHAR(03)      NOT NULL
,ckey      CHAR(03)      NOT NULL);

CREATE UNIQUE INDEX tble_x1 ON trouble (pkey, ckey);
CREATE UNIQUE INDEX tble_x2 ON trouble (ckey, pkey);

INSERT INTO trouble VALUES
('AAA','BBB'),
('AAA','CCC'),
('AAA','DDD'),
('CCC','EEE'),
('DDD','AAA'),
('DDD','EEE'),
('DDD','FFF'),
('FFF','GGG');

```

Figure 816, Sample Table DDL - Recursive Hierarchy

Other Loop Types

In the above table, the beginning object (i.e. AAA) is part of the data loop. This type of loop can be detected using simpler SQL than what is given here. But a loop that does not include the beginning object (e.g. AAA points to BBB, which points to CCC, which points back to BBB) requires the somewhat complicated SQL that is used in this section.

Stop After "n" Levels

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

<pre> WITH parent (pkey, ckey, lvl) AS (SELECT DISTINCT pkey ,pkey ,0 FROM trouble WHERE pkey = 'AAA' UNION ALL SELECT C.pkey ,C.ckey ,P.lvl + 1 FROM trouble C ,parent P WHERE P.ckey = C.pkey AND P.lvl + 1 < 4) SELECT * FROM parent; </pre>	<pre> ANSWER ===== PKEY CKEY LVL ---- AAA AAA 0 AAA BBB 1 AAA CCC 1 AAA DDD 1 CCC EEE 2 DDD AAA 2 DDD EEE 2 DDD FFF 2 AAA BBB 3 AAA CCC 3 AAA DDD 3 FFF GGG 3 </pre>	<pre> TROUBLE +-----+ PKEY CKEY +-----+ AAA BBB AAA CCC AAA DDD CCC EEE DDD AAA DDD FFF DDD EEE FFF GGG +-----+ </pre>
---	--	---

Figure 817, Stop Recursive SQL after "n" levels

In order for the above statement to get the right answer, we need to know before beginning the maximum number of valid dependent levels (i.e. non-looping) there are in the hierarchy. This information is then incorporated into the recursive predicate (see: P.LVI + 1 < 4).

If the number of levels is not known, and we guess wrong, we may not find all the children of AAA. For example, if we had stopped at "2" in the above query, we would not have found the child GGG.

A more specific disadvantage of the above statement is that the list of children contains duplicates. These duplicates include those specific values that compose the infinite loop (i.e. AAA and DDD), and also any children of either of the above.

Stop When Loop Found

A far better way to stop recursive processing is to halt when, and only when, we determine that we have been to the target row previously. To do this, we need to maintain a record of where we have been, and then check this record against the current key value in each row joined to. DB2 does not come with an in-built function that can do this checking, so we shall define our own.

Define Function

Below is the definition code for a user-defined DB2 function that is very similar to the standard LOCATE function. It searches for one string in another, block by block. For example, if one was looking for the string "ABC", this function would search the first three bytes, then the next three bytes, and so on. If a match is found, the function returns the relevant block number, else zero.

```
CREATE FUNCTION LOCATE_BLOCK(searchstr VARCHAR(30000)
                           ,lookinstr VARCHAR(30000))
RETURNS INTEGER
BEGIN ATOMIC
  DECLARE lookinlen, searchlen INT;
  DECLARE locatevar, returnvar INT DEFAULT 0;
  DECLARE beginlook          INT DEFAULT 1;
  SET lookinlen = LENGTH(lookinstr);
  SET searchlen = LENGTH(searchstr);
  WHILE locatevar = 0 AND
        beginlook <= lookinlen DO
    SET locatevar = LOCATE(searchstr,SUBSTR(lookinstr
                                           ,beginlook
                                           ,searchlen));

    SET beginlook = beginlook + searchlen;
    SET returnvar = returnvar + 1;
  END WHILE;
  IF locatevar = 0 THEN
    SET returnvar = 0;
  END IF;
  RETURN returnvar;
END
```

Figure 818, LOCATE_BLOCK user defined function

Below is an example of the function in use. Observe that the function did not find the string "th" in the name "Smith" because the two characters did not start in an position that was some multiple of the length of the test string:

SELECT id	ANSWER
,name	=====
,LOCATE('th',name) AS l1	ID NAME L1 L2
,LOCATE_BLOCK('th',name) AS l2	--- ----- -- --
FROM staff	70 Rothman 3 2
WHERE LOCATE('th',name) > 1;	220 Smith 4 0

Figure 819, LOCATE_BLOCK function example

NOTE: The LOCATE_BLOCK function shown above is the minimalist version, without any error checking. If it were used in a Production environment, it would have checks for nulls, and for various invalid input values.

Use Function

Now all we need to do is build a string, as we do the recursion, that holds every key value that has previously been accessed. This can be done using simple concatenation:

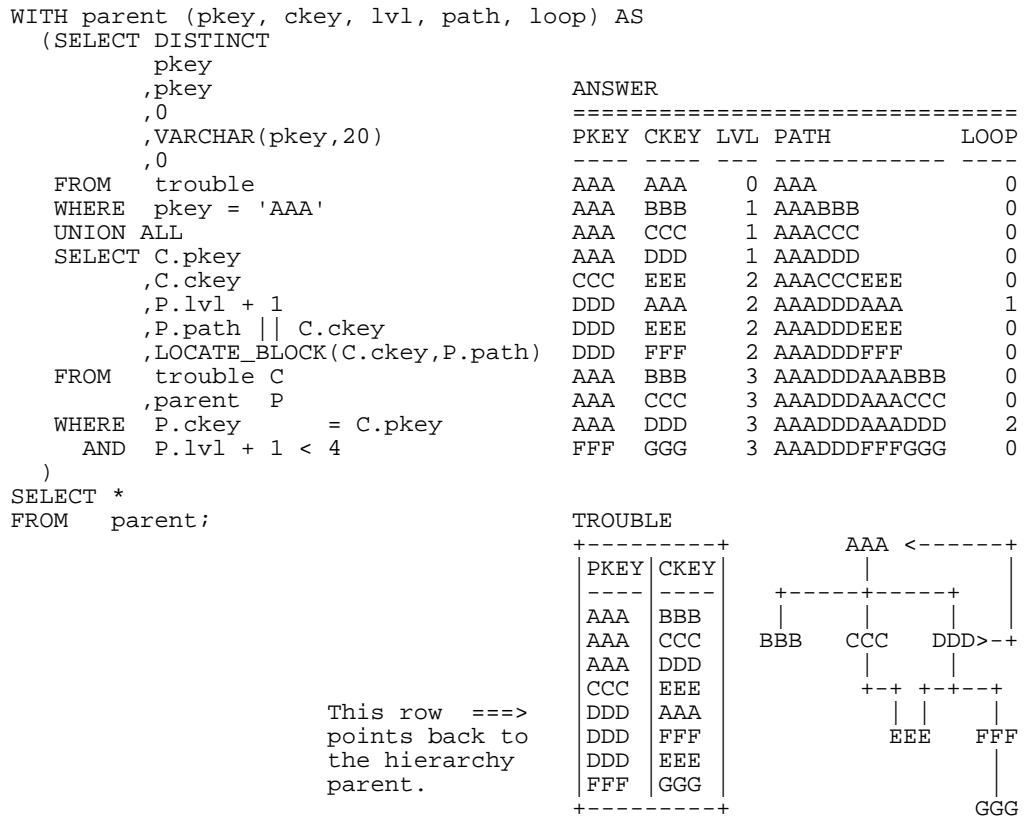


Figure 820, Show path, and rows in loop

Now we can get rid of the level check, and instead use the LOCATE_BLOCK function to avoid loops in the data:

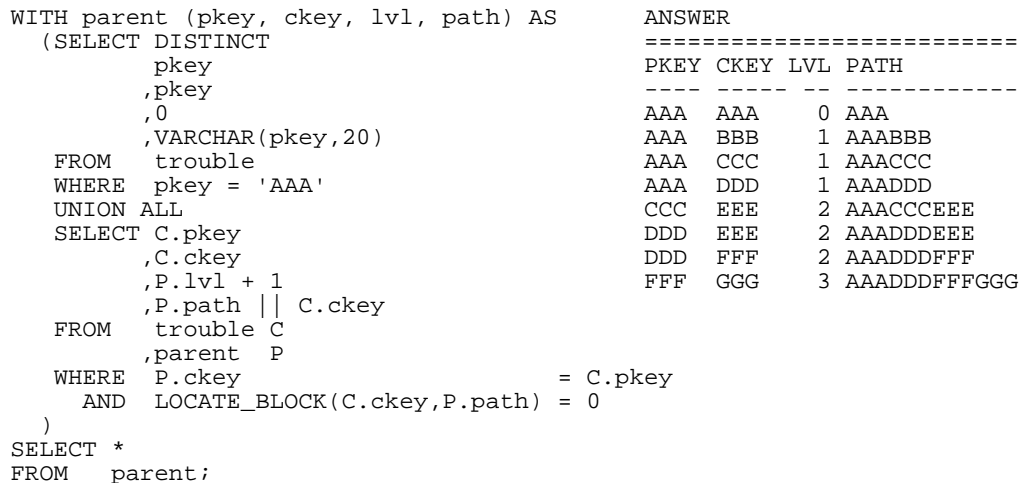


Figure 821, Use LOCATE_BLOCK function to stop recursion

The next query is the same as the previous, except that instead of excluding all loops from the answer-set, it marks them as such, and gets the first item, but goes no further;

```

WITH parent (pkey, ckey, lvl, path, loop) AS
  (SELECT DISTINCT
    pkey
    ,pkey
    ,0
    ,VARCHAR(pkey,20)
  FROM trouble
  WHERE pkey = 'AAA'
  UNION ALL
  SELECT C.pkey
    ,C.ckey
    ,P.lvl + 1
    ,P.path || C.ckey
    ,LOCATE_BLOCK(C.ckey,P.path)
  FROM trouble C
    ,parent P
  WHERE P.ckey = C.pkey
    AND P.loop = 0
  )
SELECT *
FROM parent;

```

ANSWER

```

=====
PKEY CKEY LVL PATH          LOOP
-----
AAA  AAA   0  AAA                0
AAA  BBB   1  AAABBB             0
AAA  CCC   1  AAACCC             0
AAA  DDD   1  AAADDD             0
CCC  EEE   2  AAACCCEEE         0
DDD  AAA   2  AAADDAAA           1
DDD  EEE   2  AAADDDEEE         0
DDD  FFF   2  AAADDDFFF         0
FFF  GGG   3  AAADDDFFFGGG      0

```

Figure 822, Use LOCATE_BLOCK function to stop recursion

The next query tosses in another predicate (in the final select) to only list those rows that point back to a previously processed parent:

```

WITH parent (pkey, ckey, lvl, path, loop) AS
  (SELECT DISTINCT
    pkey
    ,pkey
    ,0
    ,VARCHAR(pkey,20)
  FROM trouble
  WHERE pkey = 'AAA'
  UNION ALL
  SELECT C.pkey
    ,C.ckey
    ,P.lvl + 1
    ,P.path || C.ckey
    ,LOCATE_BLOCK(C.ckey,P.path)
  FROM trouble C
    ,parent P
  WHERE P.ckey = C.pkey
    AND P.loop = 0
  )
SELECT pkey
    ,ckey
FROM parent
WHERE loop > 0;

```

ANSWER

```

=====
PKEY CKEY
-----
DDD  AAA

```

TROUBLE

```

+-----+
| PKEY | CKEY |
|-----|-----|
| AAA  | BBB  |
| AAA  | CCC  |
| AAA  | DDD  |
| CCC  | EEE  |
| DDD  | AAA  |
| DDD  | FFF  |
| DDD  | EEE  |
| FFF  | GGG  |
+-----+

```

This row ==> points back to the hierarchy parent.

Figure 823, List rows that point back to a parent

To delete the offending rows from the table, all one has to do is insert the above values into a temporary table, then delete those rows in the TROUBLE table that match. However, before one does this, one has to decide which rows are the ones that should not be there.

In the above query, we started processing at AAA, and then said that any row that points back to AAA, or to some child of AAA, is causing a loop. We thus identified the row from DDD to AAA as being a problem. But if we had started at the value DDD, we would have said instead that the row from AAA to DDD was the problem. The point to remember here is that the row you decide to delete is a consequence of the row that you decided to define as your starting point.

```

DECLARE GLOBAL TEMPORARY TABLE SESSION.del_list
(pkey CHAR(03) NOT NULL
,ckey CHAR(03) NOT NULL)
ON COMMIT PRESERVE ROWS;

```

```

INSERT INTO SESSION.del_list
WITH parent (pkey, ckey, lvl, path, loop) AS
(SELECT DISTINCT

```

```

    pkey
    ,pkey
    ,0
    ,VARCHAR(pkey,20)
    ,0
FROM trouble
WHERE pkey = 'AAA'
UNION ALL
SELECT C.pkey
    ,C.ckey
    ,P.lvl + 1
    ,P.path || C.ckey
    ,LOCATE_BLOCK(C.ckey,P.path)
FROM trouble C
    ,parent P
WHERE P.ckey = C.pkey
    AND P.loop = 0
)
SELECT pkey
    ,ckey
FROM parent
WHERE loop > 0;

```

```

DELETE
FROM trouble
WHERE (pkey,ckey) IN
    (SELECT pkey, ckey
    FROM SESSION.del_list);

```

This row ==>
points back to
the hierarchy
parent.

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

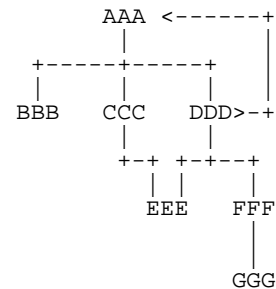


Figure 824, Delete rows that loop back to a parent

Working with Other Key Types

The LOCATE_BLOCK solution shown above works fine, as long as the key in question is a fixed length character field. If it isn't, it can be converted to one, depending on what it is:

- Cast VARCHAR columns as type CHAR.
- Convert other field types to character using the HEX function.

Keeping the Hierarchy Clean

Rather than go searching for loops, one can toss in a couple of triggers that will prevent the table from ever getting data loops in the first place. There will be one trigger for inserts, and another for updates. Both will have the same general logic:

- For each row inserted/updated, retain the new PKEY value.
- Recursively scan the existing rows, starting with the new CKEY value.
- Compare each existing CKEY value retrieved to the new PKEY value. If it matches, the changed row will cause a loop, so flag an error.
- If no match is found, allow the change.

Here is the insert trigger:

```

CREATE TRIGGER TBL_INS
NO CASCADE BEFORE INSERT ON trouble
REFERENCING NEW AS NNN
FOR EACH ROW MODE DB2SQL
  WITH temp (pkey, ckey) AS
    (VALUES (NNN.pkey
            ,NNN.ckey)
     UNION ALL
     SELECT TTT.pkey
            ,CASE
              WHEN TTT.ckey = TBL.pkey
              THEN RAISE_ERROR('70001','LOOP FOUND')
              ELSE TBL.ckey
            END
     FROM   trouble TBL
           ,temp   TTT
     WHERE  TTT.ckey = TBL.pkey
    )
  SELECT *
  FROM   temp;

```

This trigger
would reject
insertion of
this row.

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

Figure 825, INSERT trigger

Here is the update trigger:

```

CREATE TRIGGER TBL_UPD
NO CASCADE BEFORE UPDATE OF pkey, ckey ON trouble
REFERENCING NEW AS NNN
FOR EACH ROW MODE DB2SQL
  WITH temp (pkey, ckey) AS
    (VALUES (NNN.pkey
            ,NNN.ckey)
     UNION ALL
     SELECT TTT.pkey
            ,CASE
              WHEN TTT.ckey = TBL.pkey
              THEN RAISE_ERROR('70001','LOOP FOUND')
              ELSE TBL.ckey
            END
     FROM   trouble TBL
           ,temp   TTT
     WHERE  TTT.ckey = TBL.pkey
    )
  SELECT *
  FROM   temp;

```

Figure 826, UPDATE trigger

Given the above preexisting TROUBLE data (absent the DDD to AAA row), the following statements would be rejected by the above triggers:

```

INSERT INTO trouble VALUES('GGG','AAA');

UPDATE trouble SET ckey = 'AAA' WHERE pkey = 'FFF';
UPDATE trouble SET pkey = 'GGG' WHERE ckey = 'DDD';

```

Figure 827, Invalid DML statements

Observe that neither of the above triggers use the LOCATE_BLOCK function to find a loop. This is because these triggers are written assuming that the table is currently loop free. If this is not the case, they may run forever.

The LOCATE_BLOCK function enables one to check every row processed, to see if one has been to that row before. In the above triggers, only the start position is checked for loops. So if there was a loop that did not encompass the start position, the LOCATE_BLOCK check would find it, but the code used in the triggers would not.

Clean Hierarchies and Efficient Joins

Introduction

One of the more difficult problems in any relational database system involves joining across multiple hierarchical data structures. The task is doubly difficult when one or more of the hierarchies involved is a data structure that has to be resolved using recursive processing. In this section, we will describe how one can use a mixture of tables and triggers to answer this kind of query very efficiently.

A typical question might go as follows: Find all matching rows where the customer is in some geographic region, and the item sold is in some product category, and person who made the sale is in some company sub-structure. If each of these qualifications involves expanding a hierarchy of object relationships of indeterminate and/or nontrivial depth, then a simple join or standard data denormalization will not work.

In DB2, one can answer this kind of question by using recursion to expand each of the data hierarchies. Then the query would join (sans indexes) the various temporary tables created by the recursive code to whatever other data tables needed to be accessed. Unfortunately, the performance will probably be lousy.

Alternatively, one can often efficiently answer this general question using a set of suitably indexed summary tables that are an expanded representation of each data hierarchy. With these tables, the DB2 optimizer can much more efficiently join to other data tables, and so deliver suitable performance.

In this section, we will show how to make these summary tables and, because it is a prerequisite, also show how to ensure that the related base tables do not have recursive data structures. Two solutions will be described: One that is simple and efficient, but which stops updates to key values. And another that imposes fewer constraints, but which is a bit more complicated.

Limited Update Solution

Below on the left is a hierarchy of data items. This is a typical unbalanced, non-recursive data hierarchy. In the center is a normalized representation of this hierarchy. The only thing that is perhaps a little unusual here is that an item at the top of a hierarchy (e.g. AAA) is deemed to be a parent of itself. On the right is an exploded representation of the same hierarchy.

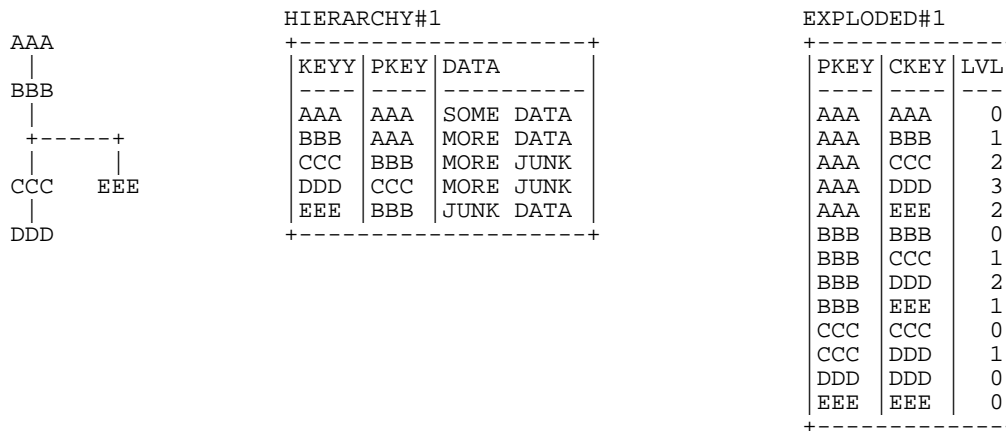


Figure 828, Data Hierarchy, with normalized and exploded representations

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 829, 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
,lv1 SMALLINT NOT NULL
,PRIMARY KEY(pkey,ckey));
```

Figure 830, 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 831, 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
AFTER INSERT ON hierarchy#1
REFERENCING NEW AS NNN
FOR EACH ROW MODE DB2SQL
INSERT
  INTO exploded#1
  WITH temp(pkey, ckey, lvl) AS
    (VALUES (NNN.keyy
            ,NNN.keyy
            ,0)
     UNION ALL
     SELECT  N.pkey
            ,NNN.keyy
            ,T.lvl +1
     FROM    temp          T
            ,hierarchy#1 N
     WHERE   N.keyy = T.pkey
            AND  N.keyy <> N.pkey
    )
  SELECT *
```

HIERARCHY#1			EXPLODED#1		
KEYY	PKEY	DATA	PKEY	CKEY	LVL
AAA	AAA	S...	AAA	AAA	0
BBB	AAA	M...	AAA	BBB	1
CCC	BBB	M...	AAA	CCC	2
DDD	CCC	M...	AAA	DDD	3
EEE	BBB	J...	AAA	EEE	2
			BBB	BBB	0
			BBB	CCC	1
			BBB	DDD	2
			BBB	EEE	1
			CCC	CCC	0
			CCC	DDD	1
			DDD	DDD	0
			EEE	EEE	0

```
FROM    temp;
```

Figure 832, Trigger to maintain exploded table after insert in hierarchy table

There is no update trigger because updates are not allowed to the hierarchy table.

Querying the Exploded Table

Once supplied with suitable indexes, the exploded table can be queried like any other table. It will always return the current state of the data in the related hierarchy table.

```
SELECT *
FROM    exploded#1
WHERE   pkey = :host-var
ORDER BY pkey
        ,ckey
        ,lvl;
```

Figure 833, 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 834, Hierarchy table that allows updates to PKEY

The previous hierarchy table came with a trigger that prevented all updates to the PKEY field. This table comes instead with a trigger than checks to see that such updates do not result in a recursive data structure. It starts out at the changed row, then works upwards through the chain of PKEY values. If it ever comes back to the original row, it flags an error:

```

CREATE TRIGGER HIR#2_UPD
NO CASCADE BEFORE UPDATE OF pkey ON hierarchy#2
REFERENCING NEW AS NNN
                OLD AS OOO
FOR EACH ROW MODE DB2SQL
WHEN (NNN.pkey <> OOO.pkey
      AND NNN.pkey <> NNN.keyy)
  WITH temp (keyy, pkey) AS
    (VALUES (NNN.keyy
            ,NNN.pkey)
     UNION ALL
     SELECT LP2.keyy
            ,CASE
              WHEN LP2.keyy = NNN.keyy
              THEN RAISE_ERROR('70001','LOOP FOUND')
              ELSE LP2.pkey
            END
     FROM   hierarchy#2 LP2
           ,temp      TMP
    WHERE  TMP.pkey = LP2.keyy
           AND TMP.keyy <> TMP.pkey
    )
SELECT *
FROM   temp;

```

HIERARCHY#2		
KEYY	PKEY	DATA
AAA	AAA	S...
BBB	AAA	M...
CCC	BBB	M...
DDD	CCC	M...
EEE	BBB	J...

Figure 835, 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 836, 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 837, 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
AFTER INSERT ON hierarchy#2
REFERENCING NEW AS NNN
FOR EACH ROW MODE DB2SQL
INSERT
  INTO exploded#2
  WITH temp(pkey, ckey, lvl) AS
    (SELECT NNN.keyy
      ,NNN.keyy
      ,0
    FROM hierarchy#2
    WHERE key = NNN.keyy
    UNION ALL
    SELECT N.pkey
      ,NNN.keyy
      ,T.lvl +1
    FROM temp T
      ,hierarchy#2 N
    WHERE N.key = T.pkey
      AND N.key <> N.pkey
    )
  SELECT *
```

HIERARCHY#2			EXPLODED#2		
KEYY	PKEY	DATA	PKEY	CKEY	LVL
AAA	AAA	S...	AAA	AAA	0
BBB	AAA	M...	AAA	BBB	1
CCC	BBB	M...	AAA	CCC	2
DDD	CCC	M...	AAA	DDD	3
EEE	BBB	J...	AAA	EEE	2
			BBB	BBB	0
			BBB	CCC	1
			BBB	DDD	2
			BBB	EEE	1
			CCC	CCC	0
			CCC	DDD	1
			DDD	DDD	0
			EEE	EEE	0

```
FROM temp;

```

Figure 838, 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 its 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 templ(ckey) AS
    (VALUES (NNN.keyy)
    UNION ALL
    SELECT N.keyy
    FROM templ T
      ,hierarchy#2 N
    WHERE N.pkey = T.ckey
      AND N.pkey <> N.keyy
    )

```

Figure 839, 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 840, 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 841, 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).

Triggers

A trigger initiates an action whenever a row, or set of rows, is changed. The change can be either an insert, update or delete.

NOTE. The *DB2 Application Development Guide: Programming Server Applications* is an excellent source of information on using triggers. The *SQL Reference* has all the basics.

Trigger Syntax

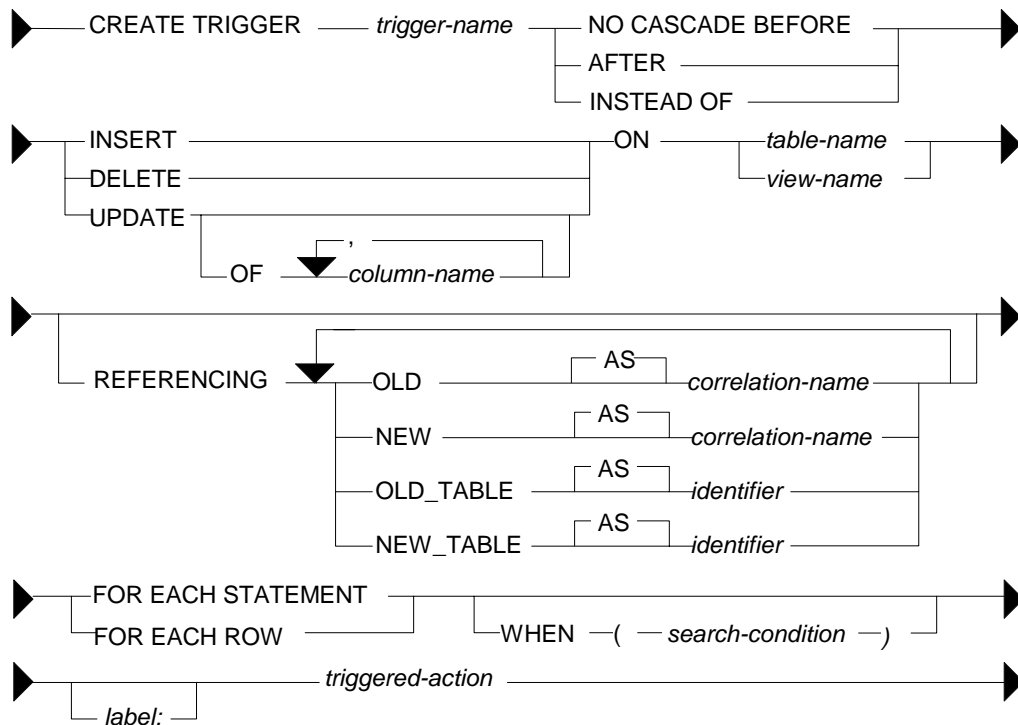


Figure 842, Create Trigger syntax

Usage Notes

Trigger Types

- A BEFORE trigger is run before the row is changed. It is typically used to change the values being entered (e.g. set a field to the current date), or to flag an error. It cannot be used to initiate changes in other tables.
- An AFTER trigger is run after the row is changed. It can do everything a before trigger can do, plus modify data in other tables or systems (e.g. it can insert a row into an audit table after an update).
- An INSTEAD OF trigger is used in a view to do something instead of the action that the user intended (e.g. do an insert instead of an update). There can be only one instead of trigger per possible DML type on a given view.

NOTE: See the chapter titled "Retaining a Record" on page 327 for a sample application that uses INSTEAD OF triggers to record all changes to the data in a set of tables.

Action Type

- Each trigger applies to a single kind of DML action (i.e. insert, update, or delete). With the exception of instead of triggers, there can be as many triggers per action and per table as desired. An update trigger can be limited to changes to certain columns.

Object Type

- A table can have both BEFORE and AFTER triggers. The former have to be defined FOR EACH ROW.
- A view can have INSTEAD OF triggers (up to three - one per DML type).

Referencing

In the body of the trigger the object being changed can be referenced using a set of optional correlation names:

- OLD refers to each individual row before the change (does not apply to an insert).
- NEW refers to each individual row after the change (does not apply to a delete).
- OLD_TABLE refers to the set of rows before the change (does not apply to an insert).
- NEW_TABLE refers to the set of rows after the change (does to apply to a delete).

Application Scope

- A trigger defined FOR EACH STATEMENT is invoked once per statement.
- A trigger defined FOR EACH ROW is invoked once per individual row changed.

NOTE: If one defines two FOR EACH ROW triggers, the first is applied for all rows before the second is run. To do two separate actions per row, one at a time, one has to define a single trigger that includes the two actions in a single compound SQL statement.

When Check

One can optionally include some predicates so that the body of the trigger is only invoked when certain conditions are true.

Trigger Usage

A trigger can be invoked whenever one of the following occurs:

- A row in a table is inserted, updated, or deleted.
- An (implied) row in a view is inserted, updated, or deleted.
- A referential integrity rule on a related table causes a cascading change (i.e. delete or set null) to the triggered table.
- A trigger on an unrelated table or view is invoked - and that trigger changes rows in the triggered table.

If no rows are changed, a trigger defined FOR EACH ROW is not run, while a trigger defined FOR EACH STATEMENT is still run. To prevent the latter from doing anything when this happens, add a suitable WHEN check.

Trigger Examples

This section uses a set of simple sample tables to illustrate general trigger usage.

Sample Tables

```
CREATE TABLE cust_balance
(cust#          INTEGER          NOT NULL
GENERATED ALWAYS AS IDENTITY
,status        CHAR(2)          NOT NULL
,balance       DECIMAL(18,2)    NOT NULL
,num_trans     INTEGER          NOT NULL
,cur_ts        TIMESTAMP        NOT NULL
,PRIMARY KEY  (cust#));
```

```
CREATE TABLE cust_history
(cust#          INTEGER          NOT NULL
,trans#        INTEGER          NOT NULL
,balance       DECIMAL(18,2)    NOT NULL
,bgn_ts        TIMESTAMP        NOT NULL
,end_ts        TIMESTAMP        NOT NULL
,PRIMARY KEY  (cust#, bgn_ts));
```

```
CREATE TABLE cust_trans
(min_cust#     INTEGER
,max_cust#     INTEGER
,rows_tot      INTEGER          NOT NULL
,change_val    DECIMAL(18,2)
,change_type   CHAR(1)          NOT NULL
,cur_ts        TIMESTAMP        NOT NULL
,PRIMARY KEY  (cur_ts));
```

← Every state of a row in the balance table will be recorded in the history table.

← Every valid change to the balance table will be recorded in the transaction table.

Figure 843, Sample Tables

Before Row Triggers - Set Values

The first trigger below overrides whatever the user enters during the insert, and before the row is inserted, sets both the cur-ts and number-of-trans columns to their correct values:

```
CREATE TRIGGER cust_bal_ins1
NO CASCADE BEFORE INSERT
ON cust_balance
REFERENCING NEW AS nnn
FOR EACH ROW
MODE DB2SQL
SET nnn.cur_ts      = CURRENT TIMESTAMP
, nnn.num_trans = 1;
```

Figure 844, Before insert trigger - set values

The following trigger does the same before an update:

```
CREATE TRIGGER cust_bal_upd1
NO CASCADE BEFORE UPDATE
ON cust_balance
REFERENCING NEW AS nnn
OLD AS ooo
FOR EACH ROW
MODE DB2SQL
SET nnn.cur_ts      = CURRENT TIMESTAMP
, nnn.num_trans = ooo.num_trans + 1;
```

Figure 845, Before update trigger - set values

Before Row Trigger - Signal Error

The next trigger will flag an error (and thus fail the update) if the customer balance is reduced by too large a value:

```
CREATE TRIGGER cust_bal_upd2
NO CASCADE BEFORE UPDATE OF balance
ON cust_balance
REFERENCING NEW AS nnn
                OLD AS ooo
FOR EACH ROW
MODE DB2SQL
WHEN (ooo.balance - nnn.balance > 1000)
    SIGNAL SQLSTATE VALUE '71001'
    SET MESSAGE_TEXT = 'Cannot withdraw > 1000';
```

Figure 846, Before Trigger - flag error

After Row Triggers - Record Data States

The three triggers in this section record the state of the data in the customer table. The first is invoked after each insert. It records the new data in the customer-history table:

```
CREATE TRIGGER cust_his_ins1
AFTER INSERT
ON cust_balance
REFERENCING NEW AS nnn
FOR EACH ROW
MODE DB2SQL
    INSERT INTO cust_history VALUES
        (nnn.cust#
        ,nnn.num_trans
        ,nnn.balance
        ,nnn.cur_ts
        , '9999-12-31-24.00.00');
```

Figure 847, After Trigger - record insert

The next trigger is invoked after every update of a row in the customer table. It first runs an update (of the old history row), and then does an insert. Because this trigger uses a compound SQL statement, it cannot use the semi-colon as the statement delimiter:

```
CREATE TRIGGER cust_his_upd1
AFTER UPDATE
ON cust_balance
REFERENCING OLD AS ooo
                NEW AS nnn
FOR EACH ROW
MODE DB2SQL
BEGIN ATOMIC
    UPDATE cust_history
    SET    end_ts = CURRENT_TIMESTAMP
    WHERE cust# = ooo.cust#
        AND bgn_ts = ooo.cur_ts;
    INSERT INTO cust_history VALUES
        (nnn.cust#
        ,nnn.num_trans
        ,nnn.balance
        ,nnn.cur_ts
        , '9999-12-31-24.00.00');
END
```

Figure 848, After Trigger - record update

Notes

- The above trigger relies on the fact that the customer-number cannot change (note: it is generated always) to link the two rows in the history table together. In other words, the old row will always have the same customer-number as the new row.
- The above also trigger relies on the presence of the cust_bal_upd1 before trigger (see page 313) to set the nnn.cur_ts value to the current timestamp.

The final trigger records a delete by doing an update to the history table:

```
CREATE TRIGGER cust_his_dell
AFTER DELETE
ON cust_balance
REFERENCING OLD AS ooo
FOR EACH ROW
MODE DB2SQL
  UPDATE cust_history
  SET   end_ts = CURRENT_TIMESTAMP
  WHERE cust# = ooo.cust#
  AND  bgn_ts = ooo.cur_ts;
```

Figure 849, After Trigger - record delete

After Statement Triggers - Record Changes

The following three triggers record every type of change (i.e. insert, update, or delete) to any row, or set of rows (including an empty set) in the customer table. They all run an insert that records the type and number of rows changed:

```
CREATE TRIGGER trans_his_ins1
AFTER INSERT
ON cust_balance
REFERENCING NEW_TABLE AS newtab
FOR EACH STATEMENT
MODE DB2SQL
  INSERT INTO cust_trans
  SELECT  MIN(cust#)
         ,MAX(cust#)
         ,COUNT(*)
         ,SUM(balance)
         ,'I'
         ,CURRENT_TIMESTAMP
  FROM    newtab;
```

Figure 850, After Trigger - record insert

```
CREATE TRIGGER trans_his_upd1
AFTER UPDATE
ON cust_balance
REFERENCING OLD_TABLE AS oldtab
          NEW_TABLE AS newtab
FOR EACH STATEMENT
MODE DB2SQL
  INSERT INTO cust_trans
  SELECT  MIN(nt.cust#)
         ,MAX(nt.cust#)
         ,COUNT(*)
         ,SUM(nt.balance - ot.balance)
         ,'U'
         ,CURRENT_TIMESTAMP
  FROM    oldtab ot
         ,newtab nt
  WHERE   ot.cust# = nt.cust#;
```

Figure 851, After Trigger - record update

```

CREATE TRIGGER trans_his_dell
AFTER DELETE
ON cust_balance
REFERENCING OLD_TABLE AS oldtab
FOR EACH STATEMENT
MODE DB2SQL
  INSERT INTO cust_trans
  SELECT  MIN(cust#)
         ,MAX(cust#)
         ,COUNT(*)
         ,SUM(balance)
         ,'D'
         ,CURRENT_TIMESTAMP
  FROM    oldtab;

```

Figure 852, After Trigger - record delete

Notes

- If the DML statement changes no rows, the OLD or NEW table referenced by the trigger will be empty, but still exist, and a SELECT COUNT(*) on the (empty) table will return a zero, which will then be inserted.
- Any DML statements that failed (e.g. stopped by the before trigger), or that were subsequently rolled back, will not be recorded in the transaction table.

Examples of Usage

The following DML statements were run against the customer table:

```

INSERT INTO cust_balance (status, balance) VALUES ('C',123.45);
INSERT INTO cust_balance (status, balance) VALUES ('C',000.00);
INSERT INTO cust_balance (status, balance) VALUES ('D', -1.00);

UPDATE cust_balance
SET   balance = balance + 123
WHERE cust#  <= 2;

UPDATE cust_balance
SET   balance = balance * -1
WHERE cust#  = -1;

UPDATE cust_balance
SET   balance = balance - 123
WHERE cust#  = 1;

DELETE
FROM  cust_balance
WHERE cust# = 3;

```

Figure 853, Sample DML statements

Tables After DML

At the end of the above, the three tables had the following data:

CUST#	STATUS	BALANCE	NUM_TRANS	CUR_TS
1	C	123.45	3	2005-05-31-19.58.46.096000
2	C	123.00	2	2005-05-31-19.58.46.034000

Figure 854, Customer-balance table rows

CUST#	TRANS#	BALANCE	BGN_TS	END_TS
1	1	123.45	2005-05-31-19.58.45.971000	2005-05-31-19.58.46.034000
1	2	246.45	2005-05-31-19.58.46.034000	2005-05-31-19.58.46.096000
1	3	123.45	2005-05-31-19.58.46.096000	9999-12-31-24.00.00.000000
2	1	0.00	2005-05-31-19.58.45.987000	2005-05-31-19.58.46.034000
2	2	123.00	2005-05-31-19.58.46.034000	9999-12-31-24.00.00.000000
3	1	-1.00	2005-05-31-19.58.46.003000	2005-05-31-19.58.46.096003

Figure 855, Customer-history table rows

MIN_CUST#	MAX_CUST#	ROWS	CHANGE_VAL	CHANGE_TYPE	CUR_TS
1	1	1	123.45	I	2005-05-31-19.58.45.971000
2	2	1	0.00	I	2005-05-31-19.58.45.987000
3	3	1	-1.00	I	2005-05-31-19.58.46.003000
1	2	2	246.00	U	2005-05-31-19.58.46.034000
-	-	0	-	U	2005-05-31-19.58.46.065000
1	1	1	-123.00	U	2005-05-31-19.58.46.096000
3	3	1	1.00	D	2005-05-31-19.58.46.096003

Figure 856, Customer-transaction table rows

Protecting Your Data

There is no use having a database if the data in it is not valid. This chapter introduces some of the tools that exist in DB2 to enable one to ensure the validity of the data in your application.

Issues Covered

- Enforcing field uniqueness.
- Enforcing field value ranges.
- Generating key values.
- Maintaining summary columns.
- Enforcing relationships between and within tables.

Issues Not Covered

- Data access authorization.
- Recovery and backup.

Sample Application

Consider the following two tables, which make up a very simple application:

```
CREATE TABLE customer_balance
(cust_id          INTEGER
 ,cust_name      VARCHAR(20)
 ,cust_sex       CHAR(1)
 ,num_sales      SMALLINT
 ,total_sales    DECIMAL(12,2)
 ,master_cust_id INTEGER
 ,cust_insert_ts TIMESTAMP
 ,cust_update_ts TIMESTAMP);

CREATE TABLE us_sales
(invoice#        INTEGER
 ,cust_id        INTEGER
 ,sale_value     DECIMAL(18,2)
 ,sale_insert_ts TIMESTAMP
 ,sale_update_ts TIMESTAMP);
```

Figure 857, Sample Views used in Join Examples

Customer Balance Table

We want DB2 to enforce the following business rules:

- CUST_ID will be a unique positive integer value, always ascending, never reused, and automatically generated by DB2. This field cannot be updated by a user.
- CUST_NAME has the customer name. It can be anything, but not blank.
- CUST_SEX must be either "M" or "F".
- NUM_SALES will have a count of the sales (for the customer), as recorded in the related US-sales table. The value will be automatically maintained by DB2. It cannot be updated directly by a user.

- **TOTAL_SALES** will have the sum sales (in US dollars) for the customer. The value will be automatically updated by DB2. It cannot be updated directly by a user.
- **MASTER_CUST_ID** will have, if there exists, the customer-ID of the customer that this customer is a dependent of. If there is no master customer, the value is null. If the master customer is deleted, this row will also be deleted (if possible).
- **CUST_INSERT_TS** has the timestamp when the row was inserted. The value is automatically generated by DB2. Any attempt to change will induce an error.
- **CUST_UPDATE_TS** has the timestamp when the row was last updated by a user (note: not by a trigger as a result of a change to the US-sales table). The value is automatically generated by DB2. Any attempt to change will induce an error.
- A row can only be deleted when there are no corresponding rows in the US-sales table (i.e. for the same customer).

US Sales Table

We want DB2 to enforce the following business rules:

- **INVOICE#**: will be a unique ascending integer value. The uniqueness will apply to the US-sales table, plus any international sales tables (i.e. to more than one table).
- **CUST_ID** is the customer ID, as recorded in the customer-balance table. No row can be inserted into the US-sales table except that there is a corresponding row in the customer-balance table. Once inserted, this value cannot be updated.
- **SALE_VALUE** is the value of the sale, in US dollars. When a row is inserted, this value is added to the related total-sales value in the customer-balance table. If the value is subsequently updated, the total-sales value is maintained in sync.
- **SALE_INSERT_TS** has the timestamp when the row was inserted. The value is automatically generated by DB2. Any attempt to change will induce an error.
- **SALE_UPDATE_TS** has the timestamp when the row was last updated. The value is automatically generated by DB2. Any attempt to change will induce an error.
- Deleting a row from the US-sales table has no impact on the customer-balance table (i.e. the total-sales is not decremented). But a row can only be deleted from the latter when there are no more related rows in the US-sales table.

Enforcement Tools

To enforce the above business rules, we are going to have to use:

- Unique indexes.
- Secondary non-unique indexes (needed for performance).
- Primary and foreign key definitions.
- User-defined distinct data types.
- Nulls-allowed and not-null columns.
- Column value constraint rules.
- Before and after triggers.

Distinct Data Types

Two of the fields are to contain US dollars, the implication being the data in these columns should not be combined with columns that contain Euros, or Japanese Yen, or my shoe size. To this end, we will define a distinct data type for US dollars:

```
CREATE DISTINCT TYPE us_dollars AS decimal(18,2) WITH COMPARISONS;
```

Figure 858, Create US-dollars data type

See page 27 for a more detailed discussion of this topic.

Customer-Balance Table

Now that we have defined the data type, we can create our first table:

```
CREATE TABLE customer_balance
(cust_id          INTEGER          NOT NULL
                  GENERATED ALWAYS AS IDENTITY
                  (START WITH 1
                   , INCREMENT BY 1
                   , NO CYCLE
                   , NO CACHE)
,cust_name       VARCHAR(20)      NOT NULL
,cust_sex        CHAR(1)         NOT NULL
,num_sales       SMALLINT        NOT NULL
,total_sales     us_dollars      NOT NULL
,master_cust_id  INTEGER
,cust_insert_ts  TIMESTAMP       NOT NULL
,cust_update_ts  TIMESTAMP       NOT NULL
,PRIMARY KEY    (cust_id)
,CONSTRAINT c1  CHECK (cust_name <> '')
,CONSTRAINT c2  CHECK (cust_sex   = 'F'
                      OR cust_sex = 'M')
,CONSTRAINT c3  FOREIGN KEY (master_cust_id)
                  REFERENCES customer_balance (cust_id)
                  ON DELETE CASCADE);
```

Figure 859, Customer-Balance table DDL

The following business rules are enforced above:

- The customer-ID is defined as an identity column (see page 257), which means that the value is automatically generated by DB2 using the rules given. The field cannot be updated by the user.
- The customer-ID is defined as the primary key, which automatically generates a unique index on the field, and also enables us to reference the field using a referential integrity rule. Being a primary key prevents updates, but we had already prevented them because the field is an identity column.
- The total-sales column is uses the type us-dollars.
- Constraints C1 and C2 enforce two data validation rules.
- Constraint C3 relates the current row to a master customer, if one exists. Furthermore, if the master customer is deleted, this row is also deleted.
- All of the columns, except for the master-customer-id, are defined as NOT NULL, which means that a value must be provided.

We still have several more business rules to enforce - relating to automatically updating fields and/or preventing user updates. These will be enforced using triggers.

US-Sales Table

Now for the related US-sales table:

```
CREATE TABLE us_sales
(invoice#          INTEGER          NOT NULL
 ,cust_id          INTEGER          NOT NULL
 ,sale_value       us_dollars       NOT NULL
 ,sale_insert_ts   TIMESTAMP        NOT NULL
 ,sale_update_ts   TIMESTAMP        NOT NULL
 ,PRIMARY KEY      (invoice#)
 ,CONSTRAINT u1 CHECK (sale_value > us_dollars(0))
 ,CONSTRAINT u2 FOREIGN KEY (cust_id)
                REFERENCES customer_balance
                ON DELETE RESTRICT);

CREATE INDEX us_sales_cust ON us_sales (cust_id);
```

Figure 860, US-Sales table DDL

The following business rules are enforced above:

- The invoice# is defined as the primary key, which automatically generates a unique index on the field, and also prevents updates.
- The sale-value uses the type us-dollars.
- Constraint U1 checks that the sale-value is always greater than zero.
- Constraint U2 checks that the customer-ID exists in the customer-balance table, and also prevents rows from being deleted from the latter if their exists a related row in this table.
- All of the columns are defined as NOT NULL, so a value must be provided for each.
- A secondary non-unique index is defined on customer-ID, so that deletes to the customer-balance table (which require checking this table for related customer-ID rows) are as efficient as possible.

Triggers

Triggers can sometimes be quite complex little programs. If coded incorrectly, they can do an amazing amount of damage. As such, it pays to learn quite a lot before using them. Below are some very brief notes, but please refer to the official DB2 documentation for a more detailed description. See also page 311 for a brief chapter on triggers.

Individual triggers are defined on a table, and for a particular type of DML statement:

- Insert.
- Update.
- Delete.

A trigger can be invoked once per:

- Row changed.
- Statement run.

A trigger can be invoked:

- Before the change is made.
- After the change is made.

Before triggers change input values before they are entered into the table and/or flag an error. After triggers do things after the row is changed. They may make more changes (to the target table, or to other tables), induce an error, or invoke an external program. SQL statements that select the changes made by DML (see page 67) cannot see the changes made by an after trigger if those changes impact the rows just changed.

The action of one "after" trigger can invoke other triggers, which may then invoke other triggers, and so on. Before triggers cannot do this because they can only act upon the input values of the DML statement that invoked them.

When there are multiple triggers for a single table/action, each trigger is run for all rows before the next trigger is invoked - even if defined "for each row". Triggers are invoked in the order that they were created.

Customer-Balance - Insert Trigger

For each row inserted into the Customer-Balance table we need to do the following:

- Set the num-sales to zero.
- Set the total-sales to zero.
- Set the update-timestamp to the current timestamp.
- Set the insert-timestamp to the current timestamp.

All of this can be done using a simple before trigger:

```
CREATE TRIGGER cust_balance_ins1
NO CASCADE BEFORE INSERT
ON customer_balance
REFERENCING NEW AS nnn
FOR EACH ROW
MODE DB2SQL
SET nnn.num_sales      = 0
   ,nnn.total_sales    = 0
   ,nnn.cust_insert_ts = CURRENT TIMESTAMP
   ,nnn.cust_update_ts = CURRENT TIMESTAMP;
```

Figure 861, Set values during insert

Customer-Balance - Update Triggers

For each row updated in the Customer-Balance table we need to do:

- Set the update-timestamp to the current timestamp.
- Prevent updates to the insert-timestamp, or sales fields.

We can use the following trigger to maintain the update-timestamp:

```
CREATE TRIGGER cust_balance_upd1
NO CASCADE BEFORE UPDATE OF cust_update_ts
ON customer_balance
REFERENCING NEW AS nnn
FOR EACH ROW
MODE DB2SQL
SET nnn.cust_update_ts = CURRENT TIMESTAMP;
```

Figure 862, Set update-timestamp during update

We can prevent updates to the insert-timestamp with the following trigger:

```

CREATE TRIGGER cust_balance_upd2
NO CASCADE BEFORE UPDATE OF cust_insert_ts
ON customer_balance
FOR EACH ROW
MODE DB2SQL
SIGNAL SQLSTATE VALUE '71001'
      SET MESSAGE_TEXT = 'Cannot update CUST insert-ts';

```

Figure 863, Prevent update of insert-timestamp

We don't want users to update the two sales counters directly. But the two fields do have to be updated (by a trigger) whenever there is a change to the us-sales table. The solution is to have a trigger that prevents updates if there is no corresponding row in the us-sales table where the update-timestamp is the current timestamp:

```

CREATE TRIGGER cust_balance_upd3
NO CASCADE BEFORE UPDATE OF num_sales, total_sales
ON customer_balance
REFERENCING NEW AS nnn
FOR EACH ROW
MODE DB2SQL
WHEN (CURRENT_TIMESTAMP NOT IN
      (SELECT sss.sale_update_ts
       FROM   us_sales sss
       WHERE  nnn.cust_id = sss.cust_id))
SIGNAL SQLSTATE VALUE '71001'
      SET MESSAGE_TEXT = 'Feilds only updated via US-Sales';

```

Figure 864, Prevent update of sales fields

US-Sales - Insert Triggers

For each row inserted into the US-sales table we need to do the following:

- Determine the invoice-number, which is unique over multiple tables.
- Set the update-timestamp to the current timestamp.
- Set the insert-timestamp to the current timestamp.
- Add the sale-value to the existing total-sales in the customer-balance table.
- Increment the num-sales counter in the customer-balance table.

The invoice-number is supposed to be unique over several tables, so we cannot generate it using an identity column. Instead, we have to call the following external sequence:

```

CREATE SEQUENCE us_sales_seq
AS INTEGER
START WITH 1
INCREMENT BY 1
NO CYCLE
NO CACHE
ORDER;

```

Figure 865, Define sequence

Once we have the above, the following trigger will take of the first three items:

```

CREATE TRIGGER us_sales_ins1
NO CASCADE BEFORE INSERT
ON us_sales
REFERENCING NEW AS nnn
FOR EACH ROW
MODE DB2SQL
SET nnn.invoice#           = NEXTVAL FOR us_sales_seq
  ,nnn.sale_insert_ts     = CURRENT_TIMESTAMP
  ,nnn.sale_update_ts     = CURRENT_TIMESTAMP;

```

Figure 866, Insert trigger

We need to use an "after" trigger to maintain the two related values in the Customer-Balance table. This will invoke an update to change the target row:

```
CREATE TRIGGER sales_to_cust_ins1
AFTER INSERT
ON us_sales
REFERENCING NEW AS nnn
FOR EACH ROW
MODE DB2SQL
UPDATE customer_balance ccc
SET   ccc.num_sales      = ccc.num_sales + 1
      ,ccc.total_sales   = DECIMAL(ccc.total_sales) +
                          DECIMAL(nnn.sale_value)
WHERE ccc.cust_id       = nnn.cust_id;
```

Figure 867, Propagate change to Customer-Balance table

US-Sales - Update Triggers

For each row updated in the US-sales table we need to do the following:

- Set the update-timestamp to the current timestamp.
- Prevent the customer-ID or insert-timestamp from being updated.
- Propagate the change to the sale-value to the total-sales in the customer-balance table.

We can use the following trigger to maintain the update-timestamp:

```
CREATE TRIGGER us_sales_upd1
NO CASCADE BEFORE UPDATE OF sale_value
ON us_sales
REFERENCING NEW AS nnn
              OLD AS ooo
FOR EACH ROW
MODE DB2SQL
SET nnn.sale_update_ts = CURRENT_TIMESTAMP;
```

Figure 868, Maintain update-timestamp

The next trigger prevents updates to the Customer-ID and insert-timestamp:

```
CREATE TRIGGER us_sales_upd2
NO CASCADE BEFORE UPDATE OF cust_id, sale_insert_ts
ON us_sales
FOR EACH ROW
MODE DB2SQL
SIGNAL SQLSTATE VALUE '71001'
SET MESSAGE_TEXT = 'Can only update sale_value';
```

Figure 869, Prevent updates to selected columns

We need to use another "after" trigger to maintain sales values in the Customer-Balance table:

```
CREATE TRIGGER sales_to_cust_upd1
AFTER UPDATE OF sale_value
ON us_sales
REFERENCING NEW AS nnn
              OLD AS ooo
FOR EACH ROW
MODE DB2SQL
UPDATE customer_balance ccc
SET   ccc.total_sales = DECIMAL(ccc.total_sales) -
                        DECIMAL(ooo.sale_value) +
                        DECIMAL(nnn.sale_value)
WHERE ccc.cust_id     = nnn.cust_id;
```

Figure 870, Propagate change to Customer-Balance table

Conclusion

The above application will now have logically consistent data. There is, of course, nothing to prevent an authorized user from deleting all rows, but whatever rows are in the two tables will obey the business rules that we specified at the start.

Tools Used

- Primary key - to enforce uniqueness, prevent updates, enable referential integrity.
- Unique index - to enforce uniqueness.
- Non-unique index - for performance during referential integrity check.
- Sequence object - to automatically generate key values for multiple tables.
- Identity column - to automatically generate key values for 1 table.
- Not-null columns - to prevent use of null values.
- Column constraints - to enforce basic domain-range rules.
- Distinct types - to prevent one type of data from being combined with another type.
- Referential integrity - to enforce relationships between rows/tables, and to enable cascading deletes when needed.
- Before triggers - to prevent unwanted changes and set certain values.
- After triggers - to propagate valid changes.

Retaining a Record

This chapter will describe a rather complex table/view/trigger schema that will enable us to offer several features that are often asked for:

- Record every change to the data in an application (auditing).
- Show the state of the data, as it was, at any point in the past (historical analysis).
- Follow the sequence of changes to any item (e.g. customer) in the database.
- Do "what if" analysis by creating virtual copies of the real world, and then changing them as desired, without affecting the real-world data.

Some sample code to illustrate the above concepts will be described below. A more complete example is available from my website.

Schema Design

Recording Changes

Below is a very simple table that records relevant customer data:

```
CREATE TABLE customer
(cust#          INTEGER          NOT NULL
 ,cust_name     CHAR(10)
 ,cust_mgr      CHAR(10)
 ,PRIMARY KEY(cust#));
```

Figure 871, Customer table

One can insert, update, and delete rows in the above table. The latter two actions destroy data, and so are incompatible with using this table to see all (prior) states of the data.

One way to record all states of the above table is to create a related customer-history table, and then to use triggers to copy all changes in the main table to the history table. Below is one example of such a history table:

```
CREATE TABLE customer_his
(cust#          INTEGER          NOT NULL
 ,cust_name     CHAR(10)
 ,cust_mgr      CHAR(10)
 ,cur_ts        TIMESTAMP        NOT NULL
 ,cur_actn      CHAR(1)          NOT NULL
 ,cur_user      VARCHAR(10)      NOT NULL
 ,prv_cust#     INTEGER
 ,prv_ts        TIMESTAMP
 ,PRIMARY KEY(cust#,cur_ts));

CREATE UNIQUE INDEX customer_his_x1 ON customer_his
(cust#, prv_ts, cur_ts);
```

Figure 872, Customer-history table

NOTE: The secondary index shown above will make the following view processing, which looks for a row that replaces the current, much more efficient.

Table Design

The history table has the same fields as the original Customer table, plus the following:

- CUR-TS: The current timestamp of the change.
- CUR-ACTN: The type of change (i.e. insert, update, or delete).
- CUR-USER: The user who made the change (for auditing purposes).
- PRV-CUST#: The previous customer number. This field enables one follow the sequence of changes for a given customer. The value is null if the action is an insert.
- PRV-TS: The timestamp of the last time the row was changed (null for inserts).

Observe that this history table does not have an end-timestamp. Rather, each row points back to the one that it (optionally) replaces. One advantage of such a schema is that there can be a many-to-one relationship between any given row, and the row, or rows, that replace it. When we add versions into the mix, this will become important.

Triggers

Below is the relevant insert trigger. It replicates the new customer row in the history table, along with the new fields. Observe that the two "previous" fields are set to null:

```
CREATE TRIGGER customer_ins
AFTER
INSERT ON customer
REFERENCING NEW AS nnn
FOR EACH ROW
MODE DB2SQL
INSERT INTO customer_his VALUES
(nnn.cust#
, nnn.cust_name
, nnn.cust_mgr
, CURRENT TIMESTAMP
, 'I'
, USER
, NULL
, NULL);
```

Figure 873, Insert trigger

Below is the update trigger. Because the customer table does not have a record of when it was last changed, we have to get this value from the history table - using a sub-query to find the most recent row:

```
CREATE TRIGGER customer_upd
AFTER
UPDATE ON customer
REFERENCING NEW AS nnn
OLD AS ooo
FOR EACH ROW
MODE DB2SQL
INSERT INTO customer_his VALUES
(nnn.cust#
, nnn.cust_name
, nnn.cust_mgr
, CURRENT TIMESTAMP
, 'U'
, USER
, ooo.cust#
, (SELECT MAX(cur_ts)
FROM customer_his hhh
WHERE ooo.cust# = hhh.cust#));
```

Figure 874, Update trigger

Below is the delete trigger. It is similar to the update trigger, except that the action is different and we are under no obligation to copy over the old non-key-data columns - but we can if we wish:

```
CREATE TRIGGER customer_del
AFTER
DELETE ON customer
REFERENCING OLD AS ooo
FOR EACH ROW
MODE DB2SQL
  INSERT INTO customer_his VALUES
  (ooo.cust#
  ,NULL
  ,NULL
  ,CURRENT TIMESTAMP
  , 'D'
  ,USER
  ,ooo.cust#
  ,(SELECT MAX(cur_ts)
  FROM   customer_his hhh
  WHERE  ooo.cust# = hhh.cust#));
```

Figure 875, Delete trigger

Views

We are now going to define a view that will let the user query the customer-history table - as if it were the ordinary customer table, but to look at the data as it was at any point in the past. To enable us to hide all the nasty SQL that is required to do this, we are going to ask that the user first enter a row into a profile table that has two columns:

- The user's DB2 USER value.
- The point in time at which the user wants to see the customer data.

Here is the profile table definition:

```
CREATE TABLE profile
(user_id      VARCHAR(10)   NOT NULL
,bgn_ts      TIMESTAMP     NOT NULL DEFAULT '9999-12-31-24.00.00'
,PRIMARY KEY(user_id));
```

Figure 876, Profile table

Below is a view that displays the customer data, as it was at the point in time represented by the timestamp in the profile table. The view shows all customer-history rows, as long as:

- The action was not a delete.
- The current-timestamp is <= the profile timestamp.
- There does not exist any row that "replaces" the current row (and that row has a current timestamp that is <= to the profile timestamp).

Now for the code:

```

CREATE VIEW customer_vw AS
SELECT hhh.*
      ,ppp.bgn_ts
FROM   customer_his hhh
      ,profile      ppp
WHERE  ppp.user_id  = USER
      AND hhh.cur_ts <= ppp.bgn_ts
      AND hhh.cur_actn <> 'D'
      AND NOT EXISTS
      (SELECT *
       FROM customer_his nnn
       WHERE nnn.prv_cust# = hhh.cust#
             AND nnn.prv_ts = hhh.cur_ts
             AND nnn.cur_ts <= ppp.bgn_ts);

```

Figure 877, View of Customer history

The above sample schema shows just one table, but it can easily be extended to support every table in a very large application. One could even write some scripts to make the creation of the history tables, triggers, and views, all but automatic.

Limitations

The above schema has the following limitations:

- Every data table has to have a unique key.
- The cost of every insert, update, and delete, is essentially doubled.
- Data items that are updated very frequently (e.g. customer daily balance) may perform poorly when queried because many rows will have to be processed in order to find the one that has not been replaced.
- The view uses the USER special register, which may not be unique per actual user.

Multiple Versions of the World

The next design is similar to the previous, but we are also going to allow users to both see and change the world - as it was in the past, and as it is now, without affecting the real-world data. These extra features require a much more complex design:

- We cannot use a base table and a related history table, as we did above. Instead we have just the latter, and use both views and INSTEAD OF triggers to make the users think that they are really seeing and/or changing the former.
- We need a version table - to record when the data in each version (i.e. virtual copy of the real world) separates from the real world data.
- Data integrity features, like referential integrity rules, have to be hand-coded in triggers, rather than written using standard DB2 code.

Version Table

The following table has one row per version created:

```

CREATE TABLE version
(vrsn      INTEGER          NOT NULL
 ,vrsn_bgn_ts  TIMESTAMP     NOT NULL
 ,CONSTRAINT version1 CHECK(vrsn >= 0)
 ,CONSTRAINT version2 CHECK(vrsn < 1000000000)
 ,PRIMARY KEY(vrsn));

```

Figure 878, Version table

The following rules apply to the above:

- Each version has a unique number. Up to one billion can be created.
- Each version must have a begin-timestamp, which records at what point in time it separates from the real world. This value must be \leq the current time.
- Rows cannot be updated or deleted in this table - only inserted. This rule is necessary to ensure that we can always trace all changes - in every version.
- The real-world is deemed to have a version number of zero, and a begin-timestamp value of high-values.

Profile Table

The following profile table has one row per user (i.e. USER special register) that reads from or changes the data tables. It records what version the user is currently using (note: the version timestamp data is maintained using triggers):

```
CREATE TABLE profile
(user_id      VARCHAR(10)  NOT NULL
, vrsn       INTEGER      NOT NULL
, vrsn_bgn_ts  TIMESTAMP   NOT NULL
, CONSTRAINT profile1 FOREIGN KEY(vrsn)
                    REFERENCES version(vrsn)
                    ON DELETE RESTRICT
, PRIMARY KEY(user_id));
```

Figure 879, Profile table

Customer (data) Table

Below is a typical data table. This one holds customer data:

```
CREATE TABLE customer_his
(cust#       INTEGER      NOT NULL
, cust_name  CHAR(10)     NOT NULL
, cust_mgr   CHAR(10)
, cur_ts     TIMESTAMP    NOT NULL
, cur_vrsn   INTEGER      NOT NULL
, cur_actn   CHAR(1)      NOT NULL
, cur_user   VARCHAR(10)  NOT NULL
, prv_cust#  INTEGER
, prv_ts     TIMESTAMP
, prv_vrsn   INTEGER
, CONSTRAINT customer1 FOREIGN KEY(cur_vrsn)
                    REFERENCES version(vrsn)
                    ON DELETE RESTRICT
, CONSTRAINT customer2 CHECK(cur_actn IN ('I', 'U', 'D'))
, PRIMARY KEY(cust#, cur_vrsn, cur_ts));

CREATE INDEX customer_x2 ON customer_his
(prv_cust#
, prv_ts
, prv_vrsn);
```

Figure 880, Customer table

Note the following:

- The first three fields are the only ones that the user will see.
- The users will never update this table directly. They will make changes to a view of the table, which will then invoke INSTEAD OF triggers.
- The foreign key check (on version) can be removed - if it is forbidden to ever delete any version. This check stops the removal of versions that have changed data.

- The constraint on CUR_ACTN is just a double-check - to make sure that the triggers that will maintain this table do not have an error. It can be removed, if desired.
- The secondary index will make the following view more efficient.

The above table has the following hidden fields:

- CUR-TS: The current timestamp of the change.
- CUR-VRSN: The version in which change occurred. Zero implies reality.
- CUR-ACTN: The type of change (i.e. insert, update, or delete).
- CUR-USER: The user who made the change (for auditing purposes).
- PRV-CUST#: The previous customer number. This field enables one follow the sequence of changes for a given customer. The value is null if the action is an insert.
- PRV-TS: The timestamp of the last time the row was changed (null for inserts).
- PRV-VRNS: The version of the row being replaced (null for inserts).

Views

The following view displays the current state of the data in the above customer table - based on the version that the user is currently using:

```
CREATE VIEW customer_vw AS
SELECT *
FROM   customer_his hhh
      ,profile      ppp
WHERE  ppp.user_id  = USER
      AND hhh.cur_actn <> 'D'
      AND (( ppp.vrsn = 0
      AND hhh.cur_vrsn = 0 )
      OR ( ppp.vrsn > 0
      AND hhh.cur_vrsn = 0
      AND hhh.cur_ts < ppp.vrsn_bgn_ts )
      OR ( ppp.vrsn > 0
      AND hhh.cur_vrsn = ppp.vrsn ) )
      AND NOT EXISTS
      (SELECT *
      FROM   customer_his nnn
      WHERE  nnn.prv_cust# = hhh.cust#
      AND    nnn.prv_ts    = hhh.cur_ts
      AND    nnn.prv_vrsn  = hhh.cur_vrsn
      AND    (( ppp.vrsn = 0
      AND    nnn.cur_vrsn = 0 )
      OR ( ppp.vrsn > 0
      AND    nnn.cur_vrsn = 0
      AND    nnn.cur_ts < ppp.vrsn_bgn_ts )
      OR ( ppp.vrsn > 0
      AND    nnn.cur_vrsn = ppp.vrsn ) ) );
```

Figure 881, Customer view - 1 of 2

The above view shows all customer rows, as long as:

- The action was not a delete.
- The version is either zero (i.e. reality), or the user's current version.
- If the version is reality, then the current timestamp is < the version begin-timestamp (as duplicated in the profile table).

- There does not exist any row that "replaces" the current row (and that row has a current timestamp that is <= to the profile (version) timestamp).

To make things easier for the users, we will create another view that sits on top of the above view. This one only shows the business fields:

```
CREATE VIEW customer AS
SELECT  cust#
        ,cust_name
        ,cust_mgr
FROM    customer_vw;
```

Figure 882, Customer view - 2 of 2

All inserts, updates, and deletes, are done against the above view, which then propagates down to the first view, whereupon they are trapped by INSTEAD OF triggers. The changes are then applied (via the triggers) to the underlying tables.

Insert Trigger

The following INSTEAD OF trigger traps all inserts to the first view above, and then applies the insert to the underlying table - with suitable modifications:

```
CREATE TRIGGER customer_ins
INSTEAD OF
INSERT ON customer_vw
REFERENCING NEW AS nnn
FOR EACH ROW
MODE DB2SQL
  INSERT INTO customer_his VALUES
  (nnn.cust#
  ,nnn.cust_name
  ,nnn.cust_mgr
  ,CURRENT TIMESTAMP
  ,(SELECT vrsn
    FROM profile
    WHERE user_id = USER)
  ,CASE
    WHEN 0 < (SELECT COUNT(*)
              FROM customer
              WHERE cust# = nnn.cust#)
    THEN RAISE_ERROR('71001','ERROR: Duplicate cust#')
    ELSE 'I'
  END
  ,USER
  ,NULL
  ,NULL
  ,NULL);
```

Figure 883, Insert trigger

Observe the following:

- The basic customer data is passed straight through.
- The current timestamp is obtained from DB2.
- The current version is obtained from the user's profile-table row.
- A check is done to see if the customer number is unique. One cannot use indexes to enforce such rules in this schema, so one has to code accordingly.
- The previous fields are all set to null.

Update Trigger

The following INSTEAD OF trigger traps all updates to the first view above, and turns them into an insert to the underlying table - with suitable modifications:

```
CREATE TRIGGER customer_upd
INSTEAD OF
UPDATE ON customer_vw
REFERENCING NEW AS nnn
                OLD AS ooo
FOR EACH ROW
MODE DB2SQL
INSERT INTO customer_his VALUES
(nnn.cust#
, nnn.cust_name
, nnn.cust_mgr
, CURRENT_TIMESTAMP
, ooo.vrsn
, CASE
    WHEN nnn.cust# <> ooo.cust#
    THEN RAISE_ERROR('72001', 'ERROR: Cannot change cust#')
    ELSE 'U'
END
, ooo.user_id
, ooo.cust#
, ooo.cur_ts
, ooo.cur_vrsn);
```

Figure 884, Update trigger

In this particular trigger, updates to the customer number (i.e. business key column) are not allowed. This rule is not necessary, it simply illustrates how one would write such code if one so desired.

Delete Trigger

The following INSTEAD OF trigger traps all deletes to the first view above, and turns them into an insert to the underlying table - with suitable modifications:

```
CREATE TRIGGER customer_del
INSTEAD OF
DELETE ON customer_vw
REFERENCING OLD AS ooo
FOR EACH ROW
MODE DB2SQL
INSERT INTO customer_his VALUES
(ooo.cust#
, ooo.cust_name
, ooo.cust_mgr
, CURRENT_TIMESTAMP
, ooo.vrsn
, 'D'
, ooo.user_id
, ooo.cust#
, ooo.cur_ts
, ooo.cur_vrsn);
```

Figure 885, Delete trigger

Summary

The only thing that the user need see in the above schema in the simplified (second) view that lists the business data columns. They would insert, update, and delete the rows in this view as if they were working on a real table. Under the covers, the relevant INSTEAD OF trigger would convert whatever they did into a suitable insert to the underlying table.

This schema supports the following:

- To do "what if" analysis, all one need do is insert a new row into the version table - with a begin timestamp that is the current time. This insert creates a virtual copy of every table in the application, which one can then update as desired.
- To do historical analysis, one simply creates a version with a begin-timestamp that is at some point in the past. Up to one billion versions are currently supported.
- To switch between versions, all one need do is update one's row in the profile table.
- One can use recursive SQL (not shown here) to follow the sequence of changes to any particular item, in any particular version.

This schema has the following limitations:

- Every data table has to have a unique (business) key.
- Data items that are updated very frequently (e.g. customer daily balance) may perform poorly when queried because many rows will have to be processed in order to find the one that has not been replaced.
- The views use the USER special register, which may not be unique per actual user.
- Data integrity features, like referential integrity rules, cascading deletes, and unique key checks, have to be hand-coded in the INSTEAD OF triggers.
- Getting the triggers right is quite hard. If the target application has many tables, it might be worthwhile to first create a suitable data-dictionary, and then write a script that generates as much of the code as possible.

Sample Code

See my website for more detailed sample code using the above schema.

Using SQL to Make SQL

This chapter describes how to use SQL to make SQL. For example, one might want to make DDL statements to create views on a set of tables.

Export Command

The following query will generate a set of queries that will count the rows in each of the selected DB2 catalogue views:

```

SELECT  'SELECT COUNT(*) FROM ' CONCAT
        RTRIM(tabschema)         CONCAT
        '.'                       CONCAT
        tablename                 CONCAT
        ';'
FROM    syscat.tables
WHERE   tabschema = 'SYSCAT'
AND     tablename LIKE 'N%'
ORDER  BY tabschema
        ,tablename;

```

ANSWER

```

=====
SELECT COUNT(*) FROM SYSCAT.NAMEMAPPINGS;
SELECT COUNT(*) FROM SYSCAT.NODEGROUPDEF;
SELECT COUNT(*) FROM SYSCAT.NODEGROUPS;

```

Figure 886, Generate SQL to count rows

If we wrap the above inside an EXPORT statement, and define no character delimiter, we will be able to create a file with the above answer - and nothing else. This could in turn be run as if were some SQL statement that we had written:

```

EXPORT TO C:\FRED.TXT OF DEL
MODIFIED BY NOCHARDEL
SELECT  'SELECT COUNT(*) FROM ' CONCAT
        RTRIM(tabschema)         CONCAT
        '.'                       CONCAT
        tablename                 CONCAT
        ';'
FROM    syscat.tables
WHERE   tabschema = 'SYSCAT'
AND     tablename LIKE 'N%'
ORDER  BY tabschema
        ,tablename;

```

Figure 887, Export generated SQL statements

Export Command Notes

The key EXPORT options used above are:

- The file name is "C\FRED.TXT".
- The data is sent to a delimited (i.e. DEL) file.
- The delimited output file uses no character delimiter (i.e. NOCHARDEL).

The remainder of this chapter will assume that we are using the EXPORT command, and will describe various ways to generate more elaborate SQL statements.

SQL to Make SQL

The next query is the same as the prior two, except that we have added the table name to each row of output:

```

SELECT  'SELECT  '''          CONCAT
         tabname          CONCAT
         ''' , COUNT(*) FROM '  CONCAT
         RTRIM(tabschema)    CONCAT
         '.'                CONCAT
         tabname            CONCAT
         ';'
FROM    syscat.tables
WHERE   tabschema = 'SYSCAT'
AND     tabname   LIKE 'N%'
ORDER  BY tabschema
        ,tabname;

```

ANSWER

```

=====
SELECT  'NAMEMAPPINGS', COUNT(*) FROM SYSCAT.NAMEMAPPINGS;
SELECT  'NODEGROUPDEF', COUNT(*) FROM SYSCAT.NODEGROUPDEF;
SELECT  'NODEGROUPS', COUNT(*) FROM SYSCAT.NODEGROUPS;

```

Figure 888, Generate SQL to count rows

We can make more readable output by joining the result set to four numbered rows, and then breaking the generated query down into four lines:

```

WITH templ (num) AS
  (VALUES (1),(2),(3),(4))
SELECT  CASE num
        WHEN 1 THEN 'SELECT '''
                | tabname
                | ''' AS tname'
        WHEN 2 THEN ' ,COUNT(*)'
                | ' AS #rows'
        WHEN 3 THEN 'FROM '
                | RTRIM(tabschema)
                | '.'
                | tabname
                | ';'
        WHEN 4 THEN ''
        END
FROM    syscat.tables
,templ
WHERE   tabschema = 'SYSCAT'
AND     tabname   LIKE 'N%'
ORDER  BY tabschema
        ,tabname
        ,num;

```

ANSWER

```

=====
SELECT 'NAMEMAPPINGS' AS tname
 ,COUNT(*) AS #rows
FROM   SYSCAT.NAMEMAPPINGS;

SELECT 'NODEGROUPDEF' AS tname
 ,COUNT(*) AS #rows
FROM   SYSCAT.NODEGROUPDEF;

SELECT 'NODEGROUPS' AS tname
 ,COUNT(*) AS #rows
FROM   SYSCAT.NODEGROUPS;

```

Figure 889, Generate SQL to count rows

So far we have generated separate SQL statements for each table that matches. But imagine that instead we wanted to create a single statement that processed all tables. For example, we might want to know the sum of the rows in all of the matching tables. There are two ways to do this, but neither of them are very good:

- We can generate a single large query that touches all of the matching tables. A query can be up to 2MB long, so we could reliably use this technique as long as we had less than about 5,000 tables to process.
- We can declare a global temporary table, then generate insert statements (one per matching table) that insert a count of the rows in the table. After running the inserts, we can sum the counts in the temporary table.

The next example generates a single query that counts all of the rows in the matching tables:

```

WITH temp1 (num) AS
  (VALUES (1),(2),(3),(4))
SELECT  CASE num
        WHEN 1 THEN 'SELECT SUM(C1)'
        when 2 then 'FROM ('
        WHEN 3 THEN '  SELECT COUNT(*) AS C1 FROM '   CONCAT
                    RTRIM(tabschema)                CONCAT
                    '.'                                CONCAT
                    tabname                          CONCAT
                    CASE dd
                    WHEN 1 THEN ''
                    ELSE ' UNION ALL'
                    END
        WHEN 4 THEN ') AS xxx;'
      END
FROM    (SELECT  tab.*
        ,ROW_NUMBER() OVER(ORDER BY tabschema ASC
        ,tabname ASC) AS aa
        ,ROW_NUMBER() OVER(ORDER BY tabschema DESC
        ,tabname DESC) AS dd
        FROM    syscat.tables tab
        WHERE   tabschema = 'SYSCAT'
        AND     tabname  LIKE 'N%'
      )AS xxx
,temp1
WHERE   (num <= 2 AND aa = 1)
      OR (num = 3)
      OR (num = 4 AND dd = 1)
ORDER BY tabschema ASC
        ,tabname  ASC
        ,num      ASC;

```

ANSWER

```

=====
SELECT SUM(C1)
FROM (
  SELECT COUNT(*) AS C1 FROM SYSCAT.NAMEMAPPINGS UNION ALL
  SELECT COUNT(*) AS C1 FROM SYSCAT.NODEGROUPDEF UNION ALL
  SELECT COUNT(*) AS C1 FROM SYSCAT.NODEGROUPS
) AS xxx;

```

Figure 890, Generate SQL to count rows (all tables)

The above query works as follows:

- A temporary table (i.e. temp1) is generated with one column and four rows.
- A nested table expression (i.e. xxx) is created with the set of matching rows (tables).
- Within the nested table expression the ROW_NUMBER function is used to define two new columns. The first will have the value 1 for the first matching row, and the second will have the value 1 for the last matching row.
- The xxx and temp1 tables are joined. Two new rows (i.e. num <= 2) are added to the front, and one new row (i.e. num = 4) is added to the back.
- The first two new rows (i.e. num = 1 and 2) are used to make the first part of the generated query.
- The last new row (i.e. num = 4) is used to make the tail end of the generated query.
- All other rows (i.e. num = 3) are used to create the core of the generated query.

In the above query no SQL is generated if no rows (tables) match. Alternatively, we might want to generate a query that returns zero if no rows match.

Running SQL Within SQL

This chapter describes how to generate and run SQL statements within SQL statements.

Introduction

Consider the following query:

```
SELECT    empno
         ,lastname
         ,workdept
         ,salary
FROM      employee
WHERE     empno = '000250';
```

Figure 891, Sample query

The above query exhibits all the usual strengths and weaknesses of the SQL language. It is easy to understand, simple to write, and assuming suitable indexes, efficient to run. But the query is annoyingly rigid in the sense that the both the internal query logic (i.e. which rows to fetch from what tables), and the set of columns to be returned, are fixed in the query syntax.

Reasonably intelligent programs accessing suitably well-structured data might want to run queries like the following:

```
SELECT    all-columns
FROM      all-relevant-tables
WHERE     all-predicates-are-true
```

Figure 892, Sample pseudo-query

It would of course be possible to compose the required query in the program and then run it. But there are some situations where it would be nice if we could also generate and then run the above pseudo-query inside the SQL language itself. This can be done, if there are two simple enhancements to the language:

- The ability to generate and run SQL within SQL.
- A way to make the query output-column-independent.

Generate SQL within SQL

To test the first concept above I wrote some very simple user-defined scalar functions (see pages: 344 and 348) that enable one to generate and run SQL within SQL. In these functions the first row/column value fetched is returned. To illustrate, consider the following pseudo-query:

```
SELECT    COUNT(*)
FROM      all-relevant-tables
WHERE     empno = '000250';
```

Figure 893, Sample pseudo-query

In the above pseudo-query we want to count all matching rows in all matching tables where the EMPNO is a given value. If we use the DB2 catalogue tables as a source dictionary, and we call a user-defined scalar function that can run SQL within SQL (see page: 348 for the function definition), we can write the following query:

```

SELECT  CHAR(tabname,15) AS tabname
        ,get_INTEGER(
          ' SELECT COUNT(*)' ||
          ' FROM ' || tabschema || '.' || tabname ||
          ' WHERE ' || colname || ' = '000250''
        ) AS num_rows
FROM    syscat.columns
WHERE   tabschema = USER
        AND colname = 'EMPNO'
        AND typename = 'CHARACTER'
ORDER BY tabname;

```

ANSWER	
TABNAME	NUM_ROWS

EMP_PHOTO	0
VEMP	1
VEMPDPT1	1
VEMPPROJACT	9
VSTAFAC2	9

Figure 894, Count matching rows in all matching tables

Make Query Column-Independent

The second issue to address was how to make the SQL language output-column-independent. This capability is needed in order to support the following type of pseudo-query:

```

SELECT  all-columns
FROM    all-relevant-tables
WHERE   empno = '000250';

```

Figure 895, Sample pseudo-query

The above query cannot be written in SQL because the set of columns to be returned can not be determined until the set of matching tables are identified. To get around this constraint, I wrote a very simple DB2 table function in Java (see page: 352) that accepts any valid query as input, runs it, and then returns all of the rows and columns fetched. But before returning anything, the function transposes each row/column instance into a single row – with a set of fixed columns returned that describe each row/column data instance (see page: 353).

The function is used below to run the above pseudo-query:

```

WITH temp1 AS
  (SELECT  tabname
         ,VARCHAR(
           ' SELECT *' ||
           ' FROM ' || tabschema || '.' || tabname ||
           ' WHERE ' || colname || ' = '000250''
         ) AS SQL_text
  FROM    syscat.columns
  WHERE   tabschema = USER
         AND colname = 'EMPNO'
         AND typename = 'CHARACTER'
  )
SELECT  CHAR(t1.tabname,10) AS tabname
        ,t2.row_number      AS row#
        ,t2.col_num        AS col#
        ,CHAR(t2.col_name,15) AS colname
        ,CHAR(t2.col_type,15) AS coltype
        ,CHAR(t2.col_value,20) AS colvalue
FROM    temp1 t1
        ,TABLE(tab_transpose(sql_text)) AS t2
ORDER BY t1.tabname
        ,t2.row_number
        ,t2.col_num;

```

Figure 896, Select all matching columns/rows in all matching tables

Below are the first three "rows" of the answer:

TABNAME	ROW#	COL#	COLNAME	COLTYPE	COLVALUE
EMPLOYEE	1	1	EMPNO	CHAR	000250
EMPLOYEE	1	2	FIRSTNME	VARCHAR	DANIEL
EMPLOYEE	1	3	MIDINIT	CHAR	S
EMPLOYEE	1	4	LASTNAME	VARCHAR	SMITH
EMPLOYEE	1	5	WORKDEPT	CHAR	D21
EMPLOYEE	1	6	PHONENO	CHAR	0961
EMPLOYEE	1	7	HIREDATE	DATE	1999-10-30
EMPLOYEE	1	8	JOB	CHAR	CLERK
EMPLOYEE	1	9	EDLEVEL	SMALLINT	15
EMPLOYEE	1	10	SEX	CHAR	M
EMPLOYEE	1	11	BIRTHDATE	DATE	1969-11-12
EMPLOYEE	1	12	SALARY	DECIMAL	49180.00
EMPLOYEE	1	13	BONUS	DECIMAL	400.00
EMPLOYEE	1	14	COMM	DECIMAL	1534.00
EMPPROJACT	1	1	EMPNO	CHAR	000250
EMPPROJACT	1	2	PROJNO	CHAR	AD3112
EMPPROJACT	1	3	ACTNO	SMALLINT	60
EMPPROJACT	1	4	EMPTIME	DECIMAL	1.00
EMPPROJACT	1	5	EMSTDATE	DATE	2002-01-01
EMPPROJACT	1	6	EMENDATE	DATE	2002-02-01
EMPPROJACT	2	1	EMPNO	CHAR	000250
EMPPROJACT	2	2	PROJNO	CHAR	AD3112
EMPPROJACT	2	3	ACTNO	SMALLINT	60
EMPPROJACT	2	4	EMPTIME	DECIMAL	0.50
EMPPROJACT	2	5	EMSTDATE	DATE	2002-02-01
EMPPROJACT	2	6	EMENDATE	DATE	2002-03-15

Figure 897, Transpose query output

Business Uses

At this point, I've got an interesting technical solution looking for a valid business problem. Some possible uses follow:

Frictionless Query

Imagine a relational database application where the table definitions are constantly changing. The programs using the data are able to adapt accordingly, in which case the intermediate SQL queries have to also be equally adaptable. The application could maintain a data dictionary that was updated in sync with the table changes. Each query would reference the dictionary at the start of its processing, and then build the main body of the query (i.e. that which obtains the desired application data) as needed.

I did some simple experiments using this concept. It worked, but I could see no overwhelming reason why one would use it, as opposed to building the query external to DB2, and then running it.

Adaptive Query

One could write a query where the internal query logic changed – depending on what data was encountered along the way. I tested this concept, and found that it works, but one still needs to define the general processing logic of the query somewhere. It was often easier to code a series of optional joins (in the query) to get the same result.

Meta-Data to Real-Data Join

A meta-data to real-data join can only be done using the SQL enhancements described above. Some examples of such a join include:

- List all tables containing a row where EMPID = '123'.
- List all rows (in any table) that duplicate a given row.

- Confirm that two "sets of tables" have identical data.
- Scan all plan-tables looking for specific access paths.
- Find the largest application table that has no index.

These types of query are relatively rare, but they certainly do exist, and they are legitimate business queries.

Meta Data Dictionaries

In the above examples the DB2 catalogue was used as the source of meta-data that describes the relationships between the tables accessed by the query. This works up to a point, but the DB2 catalogue is not really designed for this task. Thus it would probably be better to use a purpose-built meta-data dictionary. Whenever application tables were changed, the meta-data dictionary would be updated accordingly - or might in fact be the source of the change. SQL queries generated using the meta-data dictionary would automatically adjust as the table changes were implemented.

DB2 SQL Functions

This section describes how to join **meta-data** to **real data** in a single query. In other words, a query will begin by selecting a list of tables from the DB2 catalogue. It will then access each table in the list. Such a query cannot be written using ordinary SQL, because the set of tables to be accessed is not known to the statement. But it can be written if the query references a very simple user-defined scalar function and related stored procedure.

To illustrate, the following query will select a list of tables, and for each matching table get a count of the rows in the same:

```

SELECT  CHAR(tabschema,8) AS schema
        ,CHAR(tabname,20) AS tabname
        ,return_INTEGER
        ('SELECT COUNT(*) ' ||
        'FROM ' || tabschema || '.' || tabname
        )AS #rows
FROM    syscat.tables
WHERE   tabschema = 'SYSCAT'
        AND tabname LIKE 'RO%'
ORDER BY tabschema
        ,tabname
FOR FETCH ONLY
WITH UR;

```

ANSWER		
=====		
SCHEMA	TABNAME	#ROWS
-----	-----	-----
SYSCAT	ROUTINEAUTH	168
SYSCAT	ROUTINEDEP	41
SYSCAT	ROUTINEPARMS	2035
SYSCAT	ROUTINES	314

Figure 898, List tables, and count rows in same

Function and Stored Procedure Used

The above query calls a user-defined scalar function called `return_INTEGER` that accepts as input any valid single-column query and returns (you guessed it) an integer value that is the first row fetched by the query. The function is actually nothing more than a stub:


```

CREATE FUNCTION return_INTEGER (in_stmt VARCHAR(4000))
RETURNS INTEGER
LANGUAGE SQL
READS SQL DATA
NO EXTERNAL ACTION
BEGIN ATOMIC
    DECLARE out_val INTEGER;
    CALL    return_INTEGER(in_stmt,out_val);
    RETURN  out_val;
END

```

Figure 899, return_INTEGER function

The real work is done by a stored procedure that is called by the function:

```

CREATE PROCEDURE return_INTEGER (IN  in_stmt VARCHAR(4000)
                                ,OUT out_val INTEGER)

LANGUAGE SQL
READS SQL DATA
NO EXTERNAL ACTION
BEGIN
    DECLARE c1 CURSOR FOR s1;
    PREPARE s1 FROM in_stmt;
    OPEN    c1;
    FETCH   c1 INTO out_val;
    CLOSE   c1;
    RETURN;
END

```

Figure 900, return_INTEGER stored procedure

The combined function and stored-procedure logic goes as follow:

- Main query calls function - sends query text.
- Function calls stored-procedure - sends query text.
- Stored-procedure prepares, opens, fetches first row, and then closes query.
- Stored procedure returns result of first fetch back to the function
- Function returns the result back to the main query.

Different Data Types

One needs to have a function and related stored-procedure for each column type that can be returned. Below is a DECIMAL example:

```

CREATE PROCEDURE return_DECIMAL (IN  in_stmt VARCHAR(4000)
                                ,OUT out_val DECIMAL(31,6))

LANGUAGE SQL
READS SQL DATA
NO EXTERNAL ACTION
BEGIN
    DECLARE c1 CURSOR FOR s1;
    PREPARE s1 FROM in_stmt;
    OPEN    c1;
    FETCH   c1 INTO out_val;
    CLOSE   c1;
    RETURN;
END

```

Figure 901, return_DECIMAL function

```

CREATE FUNCTION return_DECIMAL (in_stmt VARCHAR(4000))
RETURNS DECIMAL(31,6)
LANGUAGE SQL
READS SQL DATA
NO EXTERNAL ACTION
BEGIN ATOMIC
    DECLARE out_val DECIMAL(31,6);
    CALL    return_DECIMAL(in_stmt,out_val);
    RETURN  out_val;
END

```

Figure 902, *return_DECIMAL* stored procedure

I have posted suitable examples for the following data types on my personal website:

- BIGINT
- INTEGER
- SMALLINT
- DECIMAL(31,6)
- FLOAT
- DATE
- TIME
- TIMESTAMP
- VARCHAR(4000)

Usage Examples

The query below lists those tables that have never had RUNSTATS run (i.e. the stats-time is null), and that currently have more than 1,000 rows:

```

SELECT  CHAR(tabschema,8) AS schema
        ,CHAR(tabname,20) AS tabname
        ,#rows
FROM    (SELECT  tabschema
          ,tabname
          ,return_INTEGER(
            ' SELECT COUNT(*)' ||
            ' FROM ' || tabschema || '.' || tabname ||
            ' FOR FETCH ONLY WITH UR'
          ) AS #rows
        FROM    syscat.tables tab
        WHERE   tabschema LIKE 'SYS%'
          AND   type = 'T'
          AND   stats_time IS NULL
        )AS xxx
WHERE   #rows > 1000
ORDER BY #rows DESC
FOR FETCH ONLY
WITH UR;

```

SCHEMA	TABNAME	#ROWS
SYSIBM	SYSCOLUMNS	3518
SYSIBM	SYSROUTINEPARMS	2035

Figure 903, *List tables never had RUNSTATS*

Efficient Queries

The query shown above would typically process lots of rows, but this need not be the case. The next example lists all tables with a department column and at least one row for the 'A00'

department. Only a single matching row is fetched from each table, so as long as there is a suitable index on the department column, the query should fly:

```

SELECT  CHAR(tab.tabname,15)  AS tabname
        ,CHAR(col.colname,10)  AS colname
        ,CHAR(COALESCE(return_VARCHAR(
          ' SELECT 'Y'' ' ||
          ' FROM ' || tab.tabschema || '.' || tab.tabname ||
          ' WHERE ' || col.colname || ' = 'A00'' ' ||
          ' FETCH FIRST 1 ROWS ONLY ' ||
          ' OPTIMIZE FOR 1 ROW ' ||
          ' WITH UR'
        ),'N'),1) AS has_dept
FROM    syscat.columns col
        ,syscat.tables tab
WHERE   col.tabschema = USER
        AND col.colname IN ('DEPTNO','WORKDEPT')
        AND col.tabschema = tab.tabschema
        AND col.tabname = tab.tabname
        AND tab.type = 'T'
FOR FETCH ONLY
WITH UR;

```

ANSWER

TABNAME	COLNAME	HAS_DEPT
DEPARTMENT	DEPTNO	Y
EMPLOYEE	WORKDEPT	Y
PROJECT	DEPTNO	N

Figure 904, List tables with a row for A00 department

The next query is the same as the previous, except that it only searches those matching tables that have a suitable index on the department field:

```

SELECT  CHAR(tab.tabname,15)  AS tabname
        ,CHAR(col.colname,10)  AS colname
        ,CHAR(COALESCE(return_VARCHAR(
          ' SELECT 'Y'' ' ||
          ' FROM ' || tab.tabschema || '.' || tab.tabname ||
          ' WHERE ' || col.colname || ' = 'A00'' ' ||
          ' FETCH FIRST 1 ROWS ONLY ' ||
          ' OPTIMIZE FOR 1 ROW ' ||
          ' WITH UR'
        ),'N'),1) AS has_dept
FROM    syscat.columns col
        ,syscat.tables tab
WHERE   col.tabschema = USER
        AND col.colname IN ('DEPTNO','WORKDEPT')
        AND col.tabschema = tab.tabschema
        AND col.tabname = tab.tabname
        AND tab.type = 'T'
        AND col.colname IN
        (SELECT SUBSTR(idx.colnames,2,LENGTH(col.colname))
         FROM syscat.indexes idx
         WHERE tab.tabschema = idx.tabschema
              AND tab.tabname = idx.tabname)
FOR FETCH ONLY
WITH UR;

```

ANSWER

TABNAME	COLNAME	HAS_DEPT
DEPARTMENT	DEPTNO	Y

Figure 905, List suitably-indexed tables with a row for A00 department

Using logic very similar to the above, one can efficiently ask questions like: "list all tables in the application that have references to customer-number 1234 in indexed fields". Even if the

query has to process hundreds of tables, each with billions of rows, it should return an answer in less than ten seconds.

In the above examples we knew what columns we wanted to process, but not the tables. But for some questions we don't even need to know the column name. For example, we could scan all indexed DATE columns in an application - looking for date values that are more than five years old. Once again, such a query should run in seconds.

Java Functions

We can do the same as the above by calling a user-defined-function that invokes a java program, but we can also do much more. This section will cover the basics.

Scalar Functions

The following code creates a user-defined scalar function that sends a query to a java program, and gets back the first row/column fetched when the query is run:

```
CREATE FUNCTION get_Integer(VARCHAR(4000))
RETURNS INTEGER
LANGUAGE JAVA
EXTERNAL NAME 'Graeme2!get_Integer'
PARAMETER STYLE DB2GENERAL
NO EXTERNAL ACTION
NOT DETERMINISTIC
READS SQL DATA
FENCED;
```

Figure 906, CREATE FUNCTION code

Below is the corresponding java code:

```
import java.lang.*;
import COM.ibm.db2.app.*;
import java.sql.*;
import java.math.*;
import java.io.*;

public class Graeme2 extends UDF {
    public void get_Integer(String inStmt,
                           int      outValue)
        throws Exception {
        try {
            Connection      con = DriverManager.getConnection
                ("jdbc:default:connection");
            PreparedStatement stmt = con.prepareStatement(inStmt);
            ResultSet        rs = stmt.executeQuery();
            if (rs.next() == true && rs.getString(1) != null) {
                set(2, rs.getInt(1));
            }
            rs.close();
            stmt.close();
            con.close();
        }
        catch (SQLException sqle) {
            setSQLState("38999");
            setSQLMessage("SQLCODE = " + sqle.getSQLState());
            return;
        }
    }
}
```

Figure 907, CREATE FUNCTION java code

Java Logic

- Establish connection.
- Prepare the SQL statement (i.e. input string).
- Execute the SQL statement (i.e. open cursor).
- If a row is found, and the value (of the first column) is not null, return value.
- Close cursor.
- Return.

Usage Example

	ANSWER
	=====
	DEPT EMPNO SALARY #ROWS

<pre>SELECT workdept AS dept ,empno ,salary ,get_Integer(' SELECT count(*)' ' FROM employee' ' where workdept = ''' workdept ''' ') AS #rows FROM employee WHERE salary < 35500 ORDER BY workdept ,empno;</pre>	<pre> E11 000290 35340.00 7 E21 200330 35370.00 6 E21 200340 31840.00 6</pre>

Figure 908, Java function usage example

I have posted suitable examples (i.e. java code, plus related CREATE FUNCTION code) for the following data types on my personal website:

- BIGINT
- INTEGER
- SMALLINT
- DOUBLE
- DECIMAL(31,6)
- VARCHAR(254)

Tabular Functions

So far, all we have done in this chapter is get single values from tables. Now we will retrieve sets of rows from tables. To do this we need to define a tabular function:

```
CREATE FUNCTION tab_Varchar (VARCHAR(4000))
RETURNS TABLE (row_number INTEGER
               ,row_value VARCHAR(254))

LANGUAGE JAVA
EXTERNAL NAME 'Graeme2!tab_Varchar'
PARAMETER STYLE DB2GENERAL
NO EXTERNAL ACTION
NOT DETERMINISTIC
DISALLOW PARALLEL
READS SQL DATA
FINAL CALL
FENCED;
```

Figure 909, CREATE FUNCTION code

Below is the corresponding java code. Observe that two columns are returned – a row-number and the value fetched:

```
import java.lang.*;
import COM.ibm.db2.app.*;
import java.sql.*;
import java.math.*;
import java.io.*;

public class Graeme2 extends UDF {
    Connection      con;
    Statement       stmt;
    ResultSet       rs;
    int             rowNum;
    public void tab_Varchar(String inStmt,
                           int      outNumber,
                           String   outValue)
        throws Exception {
        switch (getCallType()) {
            case SQLUDF_TF_FIRST:
                break;
            case SQLUDF_TF_OPEN:
                rowNum = 1;
                try {
                    con = DriverManager.getConnection
                        ("jdbc:default:connection");
                    stmt = con.createStatement();
                    rs = stmt.executeQuery(inStmt);
                }
                catch(SQLException sqle) {
                    setSQLstate("38999");
                    setSQLmessage("SQLCODE = " + sqle.getSQLState());
                    return;
                }
                break;
            case SQLUDF_TF_FETCH:
                if (rs.next() == true) {
                    set(2, rowNum);
                    if (rs.getString(1) != null) {
                        set(3, rs.getString(1));
                    }
                    rowNum++;
                }
                else {
                    setSQLstate ("02000");
                }
                break;
            case SQLUDF_TF_CLOSE:
                rs.close();
                stmt.close();
                con.close();
                break;
            case SQLUDF_TF_FINAL:
                break;
        }
    }
}
```

Figure 910, CREATE FUNCTION java code

Java Logic

Java programs that send data to DB2 table functions use a particular type of CASE logic to return the output data. In particular, a row is returned at the end of every FETCH process.

OPEN:

- Establish connection.

- Prepare the SQL statement (i.e. input string).
- Execute the SQL statement (i.e. open cursor).
- Set row-number variable to one.

FETCH:

- If row exists, set row-number output value.
- If value fetched is not null, set output value.
- Increment row-number variable.

CLOSE:

- Close cursor.
- Return.

Usage Example

The following query lists all EMPNO values that exist in more than four tables:

```

WITH
make_queries AS
  (SELECT   tab.tabschema
           ,tab.tabname
           , ' SELECT EMPNO ' ||
           ' FROM ' || tab.tabschema || '.' || tab.tabname
           AS sql_text
  FROM     syscat.tables tab
           ,syscat.columns col
  WHERE    tab.tabschema = USER
           AND tab.type = 'T'
           AND col.tabschema = tab.tabschema
           AND col.tabname = tab.tabname
           AND col.colname = 'EMPNO'
           AND col.typename = 'CHARACTER'
           AND col.length = 6
  )
,
run_queries AS
  (SELECT   qqq.*
           ,ttt.*
  FROM     make_queries qqq
           ,TABLE(tab_Varchar(sql_text)) AS ttt
  )
SELECT   CHAR(row_value,10)           AS empno
           ,COUNT(*)                 AS #rows
           ,COUNT(DISTINCT tabschema || tabname) AS #tabs
           ,CHAR(MIN(tabname),18)     AS min_tab
           ,CHAR(MAX(tabname),18)     AS max_tab
  FROM   run_queries
  GROUP BY row_value
  HAVING COUNT(DISTINCT tabschema || tabname) > 3
  ORDER BY row_value
  FOR FETCH ONLY
  WITH UR;

```

ANSWER

```

=====
EMPNO  #ROWS#TABS  MIN_TAB  MAX_TAB
-----
000130      7      4 EMP_PHOTO EMPPROJACT
000140     10      4 EMP_PHOTO EMPPROJACT
000150      7      4 EMP_PHOTO EMPPROJACT
000190      7      4 EMP_PHOTO EMPPROJACT

```

Figure 911, Use Tabular Function

Transpose Function

Below is some pseudo-code for a really cool query:

```
SELECT  all columns
FROM    unknown tables
WHERE   any unknown columns = '%ABC%'
```

Figure 912, Cool query pseudo-code

In the above query we want to retrieve an unknown number of unknown types of columns (i.e. all columns in each matching row) from an unknown set of tables where any unknown column in the row equals 'ABC'. Needless to say, the various (unknown) tables will have differing types and numbers of columns.

The above query is remarkably easy to write in SQL (see page: 355) and reasonably efficient to run, if we invoke a cute little java program that transposes columns into rows. The act of transposition means that each row/column instance retrieved becomes a separate row. So the following result:

```
SELECT  *
FROM    empproject
WHERE   empno = '000150';
```

ANSWER

```
=====
EMPNO  PROJNO  ACTNO  EMPTIME  EMSTDATE  EMENDATE
-----
000150 MA2112   60     1.00  01/01/2002  07/15/2002
000150 MA2112  180     1.00  07/15/2002  02/01/2003
```

Figure 913, Select rows

Becomes this result:

```
SELECT  SMALLINT(row_number)      AS row#
        ,col_num                  AS col#
        ,CHAR(col_name,13)        AS col_name
        ,CHAR(col_type,10)        AS col_type
        ,col_length               AS col_len
        ,SMALLINT(LENGTH(col_value)) AS val_len
        ,SUBSTR(col_value,1,20)   AS col_value
FROM    TABLE(tab_Transpose(
        ' SELECT *
          FROM   empproject'
        ' WHERE  empno = ''000150'' '
        )) AS ttt
```

ANSWER

```
=====
ROW#  COL#  COL_NAME  COL_TYPE  COL_LEN  VAL_LEN  COL_VALUE
-----
1     1  EMPNO     CHAR       6         6  000150
1     2  PROJNO    CHAR       6         6  MA2112
1     3  ACTNO     SMALLINT   6         2  60
1     4  EMPTIME   DECIMAL    7         4  1.00
1     5  EMSTDATE  DATE       10        10  2002-01-01
1     6  EMENDATE  DATE       10        10  2002-07-15
2     1  EMPNO     CHAR       6         6  000150
2     2  PROJNO    CHAR       6         6  MA2112
2     3  ACTNO     SMALLINT   6         3  180
2     4  EMPTIME   DECIMAL    7         4  1.00
2     5  EMSTDATE  DATE       10        10  2002-07-15
2     6  EMENDATE  DATE       10        10  2003-02-01
```

Figure 914, Select rows – then transpose

The user-defined transpose function invoked above accepts a query as input. It executes the query then returns the query result as one row per row/column instance found. The function output table has the following columns:

- **ROW_NUMBER:** The number of the row fetched.
- **NUM_COLS:** The number of columns fetched per row.
- **COL_NUM:** The column-number for the current row. This value, in combination with the prior row-number value, identifies a unique output row.
- **COL_NAME:** The name of the data column - as given in the query. If there is no name, the value is the column number.
- **COL_TYPE:** The DB2 column-type for the value.
- **COL_LENGTH:** The DB2 column-length (note: not data item length) for the value.
- **COL_VALUE:** The row/column instance value itself. If the data column is too long, or of an unsupported type (e.g. CLOB, DBCLOB, or XML), null is returned.

The transpose function always returns the same set of columns, regardless of which table is being accessed. So we can use it to write a query where we don't know which tables we want to select from. In the next example, we select all columns from all rows in all tables where the EMPNO column has a certain value:

```

WITH
make_queries AS
  (SELECT   tab.tabschema
           ,tab.tabname
           , ' SELECT   *' ||
           , ' FROM '   || tab.tabname ||
           , ' WHERE empno = '000150''
           AS sql_text
  FROM     syscat.tables tab
           ,syscat.columns col
  WHERE    tab.tabschema = USER
           AND tab.type = 'T'
           AND col.tabschema = tab.tabschema
           AND col.tabname = tab.tabname
           AND col.colname = 'EMPNO'
           AND col.type = 'CHARACTER'
           AND col.length = 6
  ),
run_queries AS
  (SELECT   qqq.*
           ,ttt.*
  FROM     make_queries qqq
           ,TABLE(tab_Transpose(sql_text)) AS ttt
  )
SELECT     SUBSTR(tabname,1,11)           AS tab_name
           ,SMALLINT(row_number)        AS row#
           ,col_num                      AS col#
           ,CHAR(col_name,13)           AS col_name
           ,CHAR(col_type,10)           AS col_type
           ,col_length                   AS col_len
           ,SMALLINT(LENGTH(col_value)) AS val_len
           ,SUBSTR(col_value,1,20)      AS col_value
  FROM     run_queries
  ORDER BY 1,2,3;

```

Figure 915, Select rows in any table – then transpose

When we run the above, we get the following answer:

TAB_NAME	ROW#	COL#	COL_NAME	COL_TYPE	COL_LEN	VAL_LEN	COL_VALUE
EMP_PHOTO	1	1	EMPNO	CHAR	6	6	000150
EMP_PHOTO	1	2	PHOTO_FORMAT	VARCHAR	10	6	bitmap
EMP_PHOTO	1	3	PICTURE	BLOB	204800	-	-
EMP_PHOTO	1	4	EMP_ROWID	CHAR	40	40	
EMP_PHOTO	2	1	EMPNO	CHAR	6	6	000150
EMP_PHOTO	2	2	PHOTO_FORMAT	VARCHAR	10	3	gif
EMP_PHOTO	2	3	PICTURE	BLOB	204800	-	-
EMP_PHOTO	2	4	EMP_ROWID	CHAR	40	40	
EMP_RESUME	1	1	EMPNO	CHAR	6	6	000150
EMP_RESUME	1	2	RESUME_FORMAT	VARCHAR	10	5	ascii
EMP_RESUME	1	3	RESUME	CLOB	5120	-	-
EMP_RESUME	1	4	EMP_ROWID	CHAR	40	40	
EMP_RESUME	2	1	EMPNO	CHAR	6	6	000150
EMP_RESUME	2	2	RESUME_FORMAT	VARCHAR	10	4	html
EMP_RESUME	2	3	RESUME	CLOB	5120	-	-
EMP_RESUME	2	4	EMP_ROWID	CHAR	40	40	
EMPLOYEE	1	1	EMPNO	CHAR	6	6	000150
EMPLOYEE	1	2	FIRSTNME	VARCHAR	12	5	BRUCE
EMPLOYEE	1	3	MIDINIT	CHAR	1	1	
EMPLOYEE	1	4	LASTNAME	VARCHAR	15	7	ADAMSON
EMPLOYEE	1	5	WORKDEPT	CHAR	3	3	D11
EMPLOYEE	1	6	PHONENO	CHAR	4	4	4510
EMPLOYEE	1	7	HIREDATE	DATE	10	10	2002-02-12
EMPLOYEE	1	8	JOB	CHAR	8	8	DESIGNER
EMPLOYEE	1	9	EDLEVEL	SMALLINT	6	2	16
EMPLOYEE	1	10	SEX	CHAR	1	1	M
EMPLOYEE	1	11	BIRTHDATE	DATE	10	10	1977-05-17
EMPLOYEE	1	12	SALARY	DECIMAL	11	8	55280.00
EMPLOYEE	1	13	BONUS	DECIMAL	11	6	500.00
EMPLOYEE	1	14	COMM	DECIMAL	11	7	2022.00
EMPPROJACT	1	1	EMPNO	CHAR	6	6	000150
EMPPROJACT	1	2	PROJNO	CHAR	6	6	MA2112
EMPPROJACT	1	3	ACTNO	SMALLINT	6	2	60
EMPPROJACT	1	4	EMPTIME	DECIMAL	7	4	1.00
EMPPROJACT	1	5	EMSTDATE	DATE	10	10	2002-01-01
EMPPROJACT	1	6	EMENDATE	DATE	10	10	2002-07-15
EMPPROJACT	2	1	EMPNO	CHAR	6	6	000150
EMPPROJACT	2	2	PROJNO	CHAR	6	6	MA2112
EMPPROJACT	2	3	ACTNO	SMALLINT	6	3	180
EMPPROJACT	2	4	EMPTIME	DECIMAL	7	4	1.00
EMPPROJACT	2	5	EMSTDATE	DATE	10	10	2002-07-15
EMPPROJACT	2	6	EMENDATE	DATE	10	10	2003-02-01

Figure 916, Select rows in any table – answer

We are obviously on a roll, so now we will write the pseudo-query that we began this chapter with (see page: 352). We will fetch every row/column instance in all matching tables where any qualifying column in the row is a particular value.

Query Logic

- Define the search parameters.
- Get the list of matching tables and columns to search.
- Recursively work through the list of columns to search (for each table), building a search query with multiple EQUAL predicates – one per searchable column (see page: 357).
- Run the generated queries (i.e. the final line of generated query for each table).
- Select the output.

Now for the query:

```

WITH
search_values (search_type,search_length,search_value) AS
  (VALUES ('CHARACTER',6,'000150')
  ),

list_columns AS
  (SELECT val.search_value
    ,tab.tabschema
    ,tab.tabname
    ,col.colname
    ,ROW_NUMBER() OVER(PARTITION BY val.search_value
    ,tab.tabschema
    ,tab.tabname
    ORDER BY col.colname ASC) AS col_a
    ,ROW_NUMBER() OVER(PARTITION BY val.search_value
    ,tab.tabschema
    ,tab.tabname
    ORDER BY col.colname DESC) AS col_d

  FROM search_values val
    ,syscat.tables tab
    ,syscat.columns col

  WHERE tab.tabschema = USER
    AND tab.type = 'T'
    AND tab.tabschema = col.tabschema
    AND tab.tabname = col.tabname
    AND col.typeName = val.search_type
    AND col.length = val.search_length
  ),

make_queries (search_value
  ,tabschema
  ,tabname
  ,colname
  ,col_a
  ,col_d
  ,sql_text) AS
  (SELECT tb1.*
    ,VARCHAR(' SELECT * ' || tabname || ' FROM ' ||
    ' WHERE ' || colname || ' = ' || search_value ||
    ,4000)
  FROM list_columns tb1
  WHERE col_a = 1
  UNION ALL
  SELECT tb2.*
    ,mqy.sql_text ||
    ' OR ' || tb2.colname ||
    ' = ' || tb2.search_value ||
  FROM list_columns tb2
    ,make_queries mqy
  WHERE tb2.search_value = mqy.search_value
    AND tb2.tabschema = mqy.tabschema
    AND tb2.tabname = mqy.tabname
    AND tb2.col_a = mqy.col_a + 1
  ),

run_queries AS
  (SELECT qqq.*
    ,ttt.*
  FROM make_queries qqq
    ,TABLE(tab_Transpose_4K(sql_text)) AS ttt
  WHERE col_d = 1
  )

```

Figure 917, Select rows in any table – then transpose (part 1 of 2)

```

SELECT  SUBSTR(tabname,1,11)          AS tab_name
        ,SMALLINT(row_number)        AS row#
        ,col_num                     AS col#
        ,CHAR(col_name,13)           AS col_name
        ,CHAR(col_type,10)           AS col_type
        ,col_length                   AS col_len
        ,SMALLINT(LENGTH(col_value)) AS val_len
        ,SUBSTR(col_value,1,20)      AS col_value
FROM    run_queries
ORDER BY 1,2,3;

```

Figure 918, Select rows in any table – then transpose (part 2 of 2)

Below is the answer (with a few values truncated to fit):

TAB_NAME	ROW#	COL#	COL_NAME	COL_TYPE	COL_LEN	VAL_LEN	COL_VALUE
EMP_PHOTO	1	1	EMPNO	CHAR	6	6	000150
EMP_PHOTO	1	2	PHOTO_FORMAT	VARCHAR	10	6	bitmap
EMP_PHOTO	1	3	PICTURE	BLOB	204800	-	-
EMP_PHOTO	1	4	EMP_ROWID	CHAR	40	40	
EMP_PHOTO	2	1	EMPNO	CHAR	6	6	000150
EMP_PHOTO	2	2	PHOTO_FORMAT	VARCHAR	10	3	gif
EMP_PHOTO	2	3	PICTURE	BLOB	204800	-	-
EMP_PHOTO	2	4	EMP_ROWID	CHAR	40	40	
EMP_RESUME	1	1	EMPNO	CHAR	6	6	000150
EMP_RESUME	1	2	RESUME_FORMAT	VARCHAR	10	5	ascii
EMP_RESUME	1	3	RESUME	CLOB	5120	-	-
EMP_RESUME	1	4	EMP_ROWID	CHAR	40	40	
EMP_RESUME	2	1	EMPNO	CHAR	6	6	000150
EMP_RESUME	2	2	RESUME_FORMAT	VARCHAR	10	4	html
EMP_RESUME	2	3	RESUME	CLOB	5120	-	-
EMP_RESUME	2	4	EMP_ROWID	CHAR	40	40	
EMPLOYEE	1	1	EMPNO	CHAR	6	6	000150
EMPLOYEE	1	2	FIRSTNAME	VARCHAR	12	5	BRUCE
EMPLOYEE	1	3	MIDINIT	CHAR	1	1	
EMPLOYEE	1	4	LASTNAME	VARCHAR	15	7	ADAMSON
EMPLOYEE	1	5	WORKDEPT	CHAR	3	3	D11
EMPLOYEE	1	6	PHONENO	CHAR	4	4	4510
EMPLOYEE	1	7	HIREDATE	DATE	10	10	2002-02-12
EMPLOYEE	1	8	JOB	CHAR	8	8	DESIGNER
EMPLOYEE	1	9	EDLEVEL	SMALLINT	6	2	16
EMPLOYEE	1	10	SEX	CHAR	1	1	M
EMPLOYEE	1	11	BIRTHDATE	DATE	10	10	1977-05-17
EMPLOYEE	1	12	SALARY	DECIMAL	11	8	55280.00
EMPLOYEE	1	13	BONUS	DECIMAL	11	6	500.00
EMPLOYEE	1	14	COMM	DECIMAL	11	7	2022.00
EMPPROJACT	1	1	EMPNO	CHAR	6	6	000150
EMPPROJACT	1	2	PROJNO	CHAR	6	6	MA2112
EMPPROJACT	1	3	ACTNO	SMALLINT	6	2	60
EMPPROJACT	1	4	EMPTIME	DECIMAL	7	4	1.00
EMPPROJACT	1	5	EMSTDATE	DATE	10	10	2002-01-01
EMPPROJACT	1	6	EMENDATE	DATE	10	10	2002-07-15
EMPPROJACT	2	1	EMPNO	CHAR	6	6	000150
EMPPROJACT	2	2	PROJNO	CHAR	6	6	MA2112
EMPPROJACT	2	3	ACTNO	SMALLINT	6	3	180
EMPPROJACT	2	4	EMPTIME	DECIMAL	7	4	1.00
EMPPROJACT	2	5	EMSTDATE	DATE	10	10	2002-07-15
EMPPROJACT	2	6	EMENDATE	DATE	10	10	2003-02-01
PROJECT	1	1	PROJNO	CHAR	6	6	MA2112
PROJECT	1	2	PROJNAME	VARCHAR	24	16	W L ROBOT
PROJECT	1	3	DEPTNO	CHAR	3	3	D11
PROJECT	1	4	RESPEMP	CHAR	6	6	000150
PROJECT	1	5	PRSTAFF	DECIMAL	7	4	3.00
PROJECT	1	6	PRSTDATE	DATE	10	10	2002-01-01
PROJECT	1	7	PRENDATE	DATE	10	10	1982-12-01
PROJECT	1	8	MAJPROJ	CHAR	6	6	MA2110

Figure 919, Select rows in any table – answer

Below are the queries that were generated and run to get the above answer:

```
SELECT * FROM ACT WHERE ACTKWD = '000150'
SELECT * FROM DEPARTMENT WHERE MGRNO = '000150'
SELECT * FROM EMP_PHOTO WHERE EMPNO = '000150'
SELECT * FROM EMP_RESUME WHERE EMPNO = '000150'
SELECT * FROM EMPLOYEE WHERE EMPNO = '000150'
SELECT * FROM EXPLAIN_OPERATOR WHERE OPERATOR_TYPE = '000150'
SELECT * FROM PROJECT WHERE PROJNO = '000150'
SELECT * FROM EMPPROJECT WHERE EMPNO = '000150' OR PROJNO = '000150'
SELECT * FROM PROJECT WHERE MAJPROJ = '000150' OR PROJNO = '000150' OR
RESPEMP = '000150'
```

Figure 920, Queries generated above

Function Definition

The DB2 user-defined tabular function that does the transposing is defined thus:

```
CREATE FUNCTION tab_Transpose (VARCHAR(4000))
RETURNS TABLE (row_number      INTEGER
                ,num_cols       SMALLINT
                ,col_num        SMALLINT
                ,col_name       VARCHAR(128)
                ,col_type       VARCHAR(128)
                ,col_length     INTEGER
                ,col_value      VARCHAR(254))

LANGUAGE JAVA
EXTERNAL NAME 'Graeme2!tab_Transpose'
PARAMETER STYLE DB2GENERAL
NO EXTERNAL ACTION
NOT DETERMINISTIC
DISALLOW PARALLEL
READS SQL DATA
FINAL CALL
FENCED;
```

Figure 921, Create transpose function

Java Code

```
import java.lang.*;
import COM.ibm.db2.app.*;
import java.sql.*;
import java.math.*;
import java.io.*;

public class Graeme2 extends UDF {

    Connection      con;
    Statement       stmt;
    ResultSet       rs;
    ResultSetMetaData rsmtadta;
    int             rowNum;
    int             i;
    int             outLength;
    short           colNum;
    int             colCount;
    String[] colName = new String[1100];
    String[] colType = new String[1100];
    int[] colSize = new int[1100];
    public void writeRow()
    throws Exception {
        set(2, rowNum);
        set(3, (short) colCount);
        set(4, colNum);
        set(5, colName[colNum]);
        set(6, colType[colNum]);
    }
}
```

Figure 922, CREATE FUNCTION java code (part 1 of 3)

```

set(7, colSize[colNum]);
if (colType[colNum].equals("XML")
    colType[colNum].equals("BLOB")
    colType[colNum].equals("CLOB")
    colType[colNum].equals("DBLOB")
    colType[colNum].equals("GRAPHIC")
    colType[colNum].equals("VARGRAPHIC")
    colSize[colNum] > outLength) {
    // DON'T DISPLAY THIS VALUE
    return;
}
else if (rs.getString(colNum) != null) {
    // DISPLAY THIS COLUMN VALUE
    set(8, rs.getString(colNum));
}
}

public void tab_Transpose(String inStmt
                        ,int    rowNum
                        ,short  numColumns
                        ,short  outColNumber
                        ,String outColName
                        ,String outColType
                        ,int    outColSize
                        ,String outColValue)
throws Exception {
    switch (getCallType()) {
    case SQLUDF_TF_FIRST:
        break;
    case SQLUDF_TF_OPEN:
        try {
            con      = DriverManager.getConnection
                ("jdbc:default:connection");
            stmt     = con.createStatement();
            rs       = stmt.executeQuery(inStmt);
            // GET COLUMN NAMES
            rsmdatda = rs.getMetaData();
            colCount = rsmdatda.getColumnCount();
            for (i=1; i <= colCount; i++) {
                colName[i] = rsmdatda.getColumnName(i);
                colType[i] = rsmdatda.getColumnTypeName(i);
                colSize[i] = rsmdatda.getColumnDisplaySize(i);
            }
            rowNum    = 1;
            colNum    = 1;
            outLength = 254;
        }
        catch(SQLException sqle) {
            setSQLstate("38999");
            setSQLmessage("SQLCODE = " + sqle.getSQLState());
            return;
        }
        break;
    case SQLUDF_TF_FETCH:
        if (colNum == 1 && rs.next() == true) {
            writeRow();
            colNum++;
            if (colNum > colCount) {
                colNum = 1;
                rowNum++;
            }
        }
    }
}

```

Figure 923, CREATE FUNCTION java code (part 2 of 3)

```

        else if (colNum > 1 && colNum <= colCount) {
            writeRow();
            colNum++;
            if (colNum > colCount) {
                colNum = 1;
                rowNum++;
            }
        }
        else {
            setSQLstate ("02000");
        }
        break;
    case SQLUDF_TF_CLOSE:
        rs.close();
        stmt.close();
        con.close();
        break;
    case SQLUDF_TF_FINAL:
        break;
}
}}
}

```

Figure 924, CREATE FUNCTION java code (part 3 of 3)

Java Logic

OPEN (run once):

- Establish connection.
- Prepare the SQL statement (i.e. input string).
- Execute the SQL statement (i.e. open cursor).
- Get meta-data for each column returned by query.
- Set row-number and column-number variables to one.
- Set the maximum output length accepted to 254.

FETCH (run for each row/column instance):

- If row exists and column-number is 1, fetch row.
- For value is not null and of valid DB2 type, return row.
- Increment row-number and column-number variables.

CLOSE (run once):

- Close the cursor.
- Return.

Update Real Data using Meta-Data

DB2 does not allow one to do DML or DDL using a scalar function, but one can do something similar by calling a table function. Thus if the table function defined below is joined to in a query, the following happens:

- User query joins to table function - sends DML or DDL statement to be executed.
- Table function calls stored procedure - sends statement to be executed.

- Stored procedure executes statement.
- Stored procedure returns SQLCODE of statement to the table function.
- Table function joins back to the user query a single-row table with two columns: The SQLCODE and the original input statement.

Now for the code:

```
CREATE PROCEDURE execute_immediate (IN in_stmt    VARCHAR(1000)
                                  ,OUT out_sqlcode INTEGER)

LANGUAGE SQL
MODIFIES SQL DATA
BEGIN
    DECLARE sqlcode INTEGER;
    DECLARE EXIT HANDLER FOR sqlexception
        SET out_sqlcode = sqlcode;
    EXECUTE IMMEDIATE in_stmt;
    SET out_sqlcode = sqlcode;
    RETURN;
END!

CREATE FUNCTION execute_immediate (in_stmt VARCHAR(1000))
RETURNS TABLE (sqltext VARCHAR(1000)
                ,sqlcode INTEGER)

LANGUAGE SQL
MODIFIES SQL DATA
BEGIN ATOMIC
    DECLARE out_sqlcode INTEGER;
    CALL execute_immediate(in_stmt, out_sqlcode);
    RETURN VALUES (in_stmt, out_sqlcode);
END!
```

IMPORTANT
=====

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

Figure 925, Define function and stored-procedure

WARNING: This code is extremely dangerous! Use with care. As we shall see, it is very easy for the above code to do some quite unexpected.

Usage Examples

The following query gets a list of materialized query tables for a given table-schema that need to be refreshed, and then refreshes the table:

```
WITH temp1 AS
  (SELECT   tabschema
           ,tabname
    FROM     syscat.tables
    WHERE    tabschema = 'FRED'
           AND type     = 'S'
           AND status   = 'C'
           AND tabname  LIKE '%DEPT%'
  )
SELECT     CHAR(tab.tabname,20) AS tabname
           ,stm.sqlcode      AS sqlcode
           ,CHAR(stm.sqltext,100) AS sqltext
FROM       temp1 AS tab
           ,TABLE(execute_immediate(
                'REFRESH TABLE ' ||
                RTRIM(tab.tabschema) || '.' || tab.tabname
            ))AS stm
ORDER BY  tab.tabname
WITH UR;
```

Figure 926, Refresh matching tables

I had two matching tables that needed to be refreshed, so I got the following answer:


```

TABNAME      SQLCODE  SQLTEXT
-----
STAFF_DEPT1      0  REFRESH TABLE FRED.STAFF_DEPT1
STAFF_DEPT2      0  REFRESH TABLE FRED.STAFF_DEPT2

```

Figure 927, Refresh matching tables - answer

Observe above that the set of matching tables to be refreshed was defined in a common-table-expression, and then joined to the table function. It is very important that one always code thus, because in an ordinary join it is possible for the table function to be called before all of the predicates have been applied. To illustrate this concept, the next query is supposed to make a copy of two matching tables. The answer indicates that it did just this. But what it actually did was make copies of many more tables - because the table function was called before all of the predicates on SYSCAT.TABLES were applied. The other tables that were created don't show up in the query output, because they were filtered out later in the query processing:

```

SELECT  CHAR(tab.tabname,20) AS tabname
        ,stm.sqlcode        AS sqlcode
        ,CHAR(stm.sqltext,100) AS sqltext
FROM    syscat.tables AS tab
        ,TABLE(execute_immediate(
              ' CREATE TABLE ' ||
              RTRIM(tab.tabschema) || '.' || tab.tabname || '_C1' ||
              ' LIKE ' || RTRIM(tab.tabschema) || '.' || tab.tabname
            )) AS stm
WHERE   tab.tabschema = USER
        AND tab.tabname LIKE 'S%'
ORDER BY tab.tabname
FOR FETCH ONLY
WITH UR;

```

ANSWER

```

=====
TABNAME SQLCODE SQLTEXT
-----
SALES      0  CREATE TABLE FRED.SALES_C1 LIKE FRED.SALES
STAFF      0  CREATE TABLE FRED.STAFF_C1 LIKE FRED.STAFF

```

Figure 928, Create copies of tables - wrong

The above is bad enough, but I once managed to do much worse. In a variation of the above code, the query created a copy, of a copy, of a copy, etc. The table function kept finding the table just created, and making a copy of it - until the TABNAME reached the length limit.

The correct way to create a copy of a set of tables is shown below. In this query, the list of tables to be copied is identified in a common table expression before the table function is called:

```

WITH temp1 AS
  (SELECT  tabschema
         ,tabname
    FROM    syscat.tables
   WHERE   tabschema = USER
   AND     tabname    LIKE 'S%'
  )
SELECT  CHAR(tab.tabname,20) AS tabname
       ,stm.sqlcode         AS sqlcode
       ,CHAR(stm.sqltext,100) AS sqltext
FROM    temp1 tab
       ,TABLE(execute_immediate(
              ' CREATE TABLE ' ||
              RTRIM(tab.tabschema) || '.' || tab.tabname || '_C1' ||
              ' LIKE ' || RTRIM(tab.tabschema) || '.' || tab.tabname
            ))AS stm
ORDER BY tab.tabname
FOR FETCH ONLY
WITH UR;

```

ANSWER

```

=====
TABNAME  SQLCODE  SQLTEXT
-----
SALES    0  CREATE TABLE FRED.SALES_C1 LIKE FRED.SALES
STAFF    0  CREATE TABLE FRED.STAFF_C1 LIKE FRED.STAFF

```

Figure 929, Create copies of tables - right

The next example is similar to the previous, except that it creates a copy, and then populates the new table with the contents of the original table:

```

WITH
temp0 AS
  (SELECT  RTRIM(tabschema) AS schema
         ,tabname          AS old_tabname
         ,tabname || '_C2' AS new_tabname
    FROM    syscat.tables
   WHERE   tabschema = USER
   AND     tabname    LIKE 'S%'
  ),
temp1 AS
  (SELECT  tab.*
         ,stm.sqlcode         AS sqlcode1
         ,CHAR(stm.sqltext,200) AS sqltext1
    FROM    temp0 AS tab
         ,TABLE(execute_immediate(
              ' CREATE TABLE ' || schema || '.' || new_tabname ||
              ' LIKE ' || schema || '.' || old_tabname
            ))AS stm
  ),
temp2 AS
  (SELECT  tab.*
         ,stm.sqlcode         AS sqlcode2
         ,CHAR(stm.sqltext,200) AS sqltext2
    FROM    temp1 AS tab
         ,TABLE(execute_immediate(
              ' INSERT INTO ' || schema || '.' || new_tabname ||
              ' SELECT * FROM ' || schema || '.' || old_tabname
            ))AS stm
  )
SELECT  CHAR(old_tabname,20) AS tabname
       ,sqlcode1
       ,sqlcode2
FROM    temp2
ORDER BY old_tabname
FOR FETCH ONLY
WITH UR;

```

ANSWER

```

=====
TABNAME  SQLCODE1  SQLCODE2
-----
SALES    0          0
STAFF    0          0

```

Figure 930, Create copies of tables, then populate

Query Processing Sequence

In order to explain the above, we need to understand in what sequence the various parts of a query are executed in order to avoid semantic ambiguity:

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

Figure 931, Query Processing Sequence

Observe above that the FROM clause is resolved before any WHERE predicates are applied. This is why the query in figure 928 did the wrong thing.

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.

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 templ (coll) AS
  (VALUES      0
   UNION ALL
   SELECT coll + 1
   FROM  templ
   WHERE coll + 1 < 100
  )
SELECT *
FROM  templ;

```

ANSWER
=====
COL1

0
1
2
3
etc

Figure 932, Use recursion to get list of 100 numbers

Instead of coding a recursion join every time, we use the table function described on page 176 to create the required rows. Assuming that the function exists, one would write the following:

```

SELECT *
FROM  TABLE(NumList(100)) AS xxx;

```

Figure 933, Use user-defined-function to get list of 100 numbers

Make Reproducible Random Data

So far, all we have done is create sets of fixed values. These are usually not suitable for testing purposes because they are too consistent. To mess things up a bit we need to use the RAND function, which generates random numbers in the range of zero to one inclusive. In the next example we will get a (reproducible) list of five random numeric values:

```

WITH temp1 (s1, r1) AS
(VALUES (0, RAND(1))
 UNION ALL
 SELECT s1+1, RAND()
 FROM temp1
 WHERE s1+1 < 5
 )
SELECT SMALLINT(s1) AS seq#
      ,DECIMAL(r1,5,3) AS ran1
FROM temp1;

```

ANSWER	
SEQ#	RAN1
0	0.001
1	0.563
2	0.193
3	0.808
4	0.585

Figure 934, 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 133) 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
(VALUES (0, RAND(2))
 UNION ALL
 SELECT s1+1, RAND()
 FROM temp1
 WHERE s1+1 < 5
 )
SELECT SMALLINT(s1) AS seq#
      ,SMALLINT(r1*10000) AS ran2
      ,DECIMAL(r1,6,4) AS ran1
      ,SMALLINT(r1*10) AS ran3
FROM temp1;

```

ANSWER			
SEQ#	RAN2	RAN1	RAN3
0	13	0.0013	0
1	8916	0.8916	8
2	7384	0.7384	7
3	5430	0.5430	5
4	8998	0.8998	8

Figure 935, Make differing ranges of random numbers

Make Random Data - Varying Distribution

In the real world, there is a tendency for certain data values to show up much more frequently than others. Likewise, separate fields in a table usually have independent semi-random data distribution patterns. In the next statement we create three independently random fields. The first has the usual 32K distinct values evenly distributed in the range of zero to one. The sec-

ond and third have random numbers that are skewed towards the low end of the range, and have many more distinct values:

		ANSWER			
		=====			
		S#	RAN1	RAN2	RAN3
		---	---	---	---
WITH					
temp1 (s1) AS		0	1251	365370	114753
(VALUES (0)		1	350291	280730	88106
UNION ALL		2	710501	149549	550422
SELECT s1 + 1		3	147312	33311	2339
FROM temp1		4	8911	556	73091
WHERE s1 + 1 < 5					
)					
SELECT SMALLINT(s1)	AS s#				
, INTEGER((RAND(1))	* 1E6) AS ran1				
, INTEGER((RAND() * RAND())	* 1E6) AS ran2				
, INTEGER((RAND() * RAND() * RAND())	* 1E6) AS ran3				
FROM temp1;					

Figure 936, Create RAND data with different distributions

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.

		ANSWER			
		=====			
		SEQ#	RAN2	RAN3	RAN4
		---	---	---	---
WITH temp1 (s1, r1) AS					
(VALUES (0, RAND(2))					
UNION ALL					
SELECT s1+1, RAND()		0	65 A		65
FROM temp1		1	88 X		88
WHERE s1+1 < 5		2	84 T		84
)		3	79 O		79
SELECT SMALLINT(s1)	AS seq#	4	88 X		88
, SMALLINT(r1*26+65)	AS ran2				
, CHR(SMALLINT(r1*26+65))	AS ran3				
, CHAR(SMALLINT(r1*26)+65)	AS ran4				
FROM temp1;					

Figure 937, Converting RAND output from number to character

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' ));

CREATE UNIQUE INDEX PEX2 ON PERSONNEL (SOCSEC#);
CREATE UNIQUE INDEX PEX3 ON PERSONNEL (DEPT, EMP#);

```

Figure 938, 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
        ,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 939, Production-like test table INSERT

Some sample data follows:

EMP#	SOCSEC#	JOB_	DEPT	SALARY	DATE_BN	F_NME	L_NME
100000	484-10-9999	WKR	47	13.63	1979-01-01	Ammaef	Mimytmbi
100001	449-38-9998	SEC	53	35758.87	1962-04-10	Ilojff	Liiiemea
100002	979-90-9997	WKR	1	8155.23	1975-01-03	Xzacia	Zytaebma
100003	580-50-9993	WKR	31	16643.50	1971-02-05	Lpiedd	Pimmeet
100004	264-87-9994	WKR	21	962.87	1979-01-01	Wgfacc	Geimteei
100005	661-84-9995	WKR	19	4648.38	1977-01-02	Wrebbs	Rbiybeet
100006	554-53-9990	WKR	8	375.42	1979-01-01	Mobaaa	Oiaiaia
100007	482-23-9991	SEC	36	23170.09	1968-03-07	Emjgdd	Mimtmamb
100008	536-41-9992	WKR	6	10514.11	1974-02-03	Jnbcaa	Nieebayt

Figure 940, 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 368).
- 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 these two fields have related values.
- The BIRTH DATE is a random date value somewhere between 1930 and 1981.
- The FIRST NAME is derived using seven independent invocation of the CHR function, each of which is going to give a somewhat different result.
- The LAST NAME is (mostly) made by using the TRANSLATE function to convert a large random number into a corresponding character value. The output is skewed towards some of the vowels and the lower-range characters during the translation.

Time-Series Processing

The following table holds data for a typical time-series application. Observe is that each row has both a beginning and ending date, and that there are three cases where there is a gap between the end-date of one row and the begin-date of the next (with the same key).

```

CREATE TABLE time_series
(KYY      CHAR(03)      NOT NULL
,bgn_dt   DATE          NOT NULL
,end_dt   DATE          NOT NULL
,CONSTRAINT 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 941, 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 KYY value. The following diagram illustrates our task. The bold line at the top represents the begin and end date for a row. This row is overlapped (in time) by the six lower rows, but the nature of the overlap differs in each case.

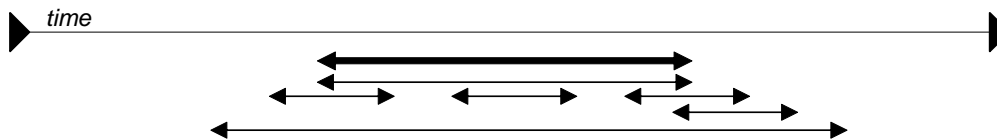


Figure 942, Overlapping Time-Series rows - Definition

The general types of overlap are:

- The related row has identical date ranges.
- The related row begins before the start-date and ends after the same.
- The row begins and ends between the start and finish dates.

WARNING: When writing SQL to check overlapping data ranges, make sure that all possible types of overlap (see diagram above) are tested. Some SQL statements work with some flavors of overlap, but not with others.

The relevant SQL follows. When reading it, think of the "A" table as being the bold line above and "B" table as being the four overlapping rows shown as single lines.

```

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 943, 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 370. This means that the END_DT is always greater than or equal to the BGN_DT, and each KYY, BGN_DT combination is unique.

Find Gaps in Time-Series

We want to find all those cases in the TIME_SERIES table when the ending of one row is not exactly one day less than the beginning of the next (if there is a next). The following query will answer this question. It consists of both a join and a sub-query. In the join (which is done first), we match each row with every other row that has the same key and a BGN_DT that is more than one day greater than the current END_DT. Next, the sub-query excludes from the result those join-rows where there is an intermediate third row.

```

SELECT a.kyy
      ,a.bgn_dt
      ,a.end_dt
      ,b.bgn_dt
      ,b.end_dt
      ,DAYS(b.bgn_dt) -
      DAYS(A.end_dt)
      as diff
FROM   time_series a
      ,time_series b
WHERE  a.kyy = b.kyy
      AND a.end_dt < b.bgn_dt - 1 DAY
      AND NOT EXISTS
      (SELECT *
       FROM time_series z
       WHERE z.kyy = a.kyy
            AND z.kyy = b.kyy
            AND z.bgn_dt > a.bgn_dt
            AND z.bgn_dt < b.bgn_dt)
ORDER BY 1,2;

```

TIME_SERIES		
KYY	BGN_DT	END_DT
AAA	1995-10-01	1995-10-04
AAA	1995-10-06	1995-10-06
AAA	1995-10-07	1995-10-07
AAA	1995-10-15	1995-10-19
BBB	1995-10-01	1995-10-01
BBB	1995-10-03	1995-10-03

Figure 944, Find gap in Time-Series, SQL

KEYCOL	BGN_DT	END_DT	BGN_DT	END_DT	DIFF
AAA	1995-10-01	1995-10-04	1995-10-06	1995-10-06	2
AAA	1995-10-07	1995-10-07	1995-10-15	1995-10-19	8
BBB	1995-10-01	1995-10-01	1995-10-03	1995-10-03	2

Figure 945, 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          AS kyy
      ,a.end_dt + 1 DAY AS bgn_gap
      ,b.bgn_dt - 1 DAY AS end_gap
      ,(DAYS(b.bgn_dt) -
        DAYS(a.end_dt) - 1) AS sz
FROM   time_series a
      ,time_series b
WHERE  a.kyy = b.kyy
      AND a.end_dt < b.bgn_dt - 1 DAY
      AND NOT EXISTS
      (SELECT *
       FROM   time_series z
        WHERE z.kyy = a.kyy
              AND z.kyy = b.kyy
              AND z.bgn_dt > a.bgn_dt
              AND z.bgn_dt < b.bgn_dt)
ORDER BY 1,2;

```

KYY	BGN_DT	END_DT
AAA	1995-10-01	1995-10-04
AAA	1995-10-06	1995-10-06
AAA	1995-10-07	1995-10-07
AAA	1995-10-15	1995-10-19
BBB	1995-10-01	1995-10-01
BBB	1995-10-03	1995-10-03

```

ANSWER
=====
KYY BGN_GAP END_GAP SZ
---
AAA 1995-10-05 1995-10-05 1
AAA 1995-10-08 1995-10-14 7
BBB 1995-10-02 1995-10-02 1

```

Figure 946, Find gap in Time-Series

Show Each Day in Gap

Imagine that we wanted to see each individual day in a gap. The following statement does this by taking the result obtained above and passing it into a recursive SQL statement which then generates additional rows - one for each day in the gap after the first.

```

WITH temp
(kyy, gap_dt, gsize) AS
(SELECT a.kyy
      ,a.end_dt + 1 DAY
      ,(DAYS(b.bgn_dt) -
        DAYS(a.end_dt) - 1)
FROM   time_series a
      ,time_series b
WHERE  a.kyy = b.kyy
      AND a.end_dt < b.bgn_dt - 1 DAY
      AND NOT EXISTS
      (SELECT *
       FROM   time_series z
        WHERE z.kyy = a.kyy
              AND z.kyy = b.kyy
              AND z.bgn_dt > a.bgn_dt
              AND z.bgn_dt < b.bgn_dt)
UNION ALL
SELECT kyy
      ,gap_dt + 1 DAY
      ,gsize - 1
FROM   temp
WHERE  gsize > 1
)
SELECT *
FROM   temp
ORDER BY 1,2;

```

KYY	BGN_DT	END_DT
AAA	1995-10-01	1995-10-04
AAA	1995-10-06	1995-10-06
AAA	1995-10-07	1995-10-07
AAA	1995-10-15	1995-10-19
BBB	1995-10-01	1995-10-01
BBB	1995-10-03	1995-10-03

```

ANSWER
=====
KEYCOL GAP_DT GSIZE
---
AAA 1995-10-05 1
AAA 1995-10-08 7
AAA 1995-10-09 6
AAA 1995-10-10 5
AAA 1995-10-11 4
AAA 1995-10-12 3
AAA 1995-10-13 2
AAA 1995-10-14 1
BBB 1995-10-02 1

```

Figure 947, Show each day in Time-Series gap

Other Fun Things

Randomly Sample Data

One can use the TABLESAMPLE schema to randomly sample rows for subsequent analysis.

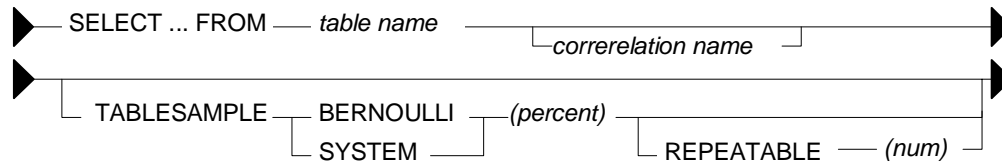


Figure 948, TABLESAMPLE Syntax

Notes

- The table-name must refer to a real table. This can include a declared global temporary table, or a materialized query table. It cannot be a nested table expression.
- The sampling is an addition to any predicates specified in the where clause. Under the covers, sampling occurs before any other query processing, such as applying predicates or doing a join.
- The BERNOULLI option checks each row individually.
- The SYSTEM option lets DB2 find the most efficient way to sample the data. This may mean that all rows on each page that qualifies are included. For small tables, this method often results in an misleading percentage of rows selected.
- The "percent" number must be equal to or less than 100, and greater than zero. It determines what percentage of the rows processed are returns.
- The REPEATABLE option and number is used if one wants to get the same result every time the query is run (assuming no data changes). Without this option, each run will be both random and different.

Examples

Sample 5% of the rows in the staff table. Get the same result each time:

```
SELECT *
FROM   staff TABLESAMPLE BERNOULLI(5) REPEATABLE(1234)
ORDER BY id;
```

Figure 949, Sample rows in STAFF table

Sample 18% of the rows in the employee table and 25% of the rows in the employee-activity table, then join the two tables together. Because each table is sampled independently, the fraction of rows that join will be much less either sampling rate:

```
SELECT *
FROM   employee ee TABLESAMPLE BERNOULLI(18)
       ,emp_act ea TABLESAMPLE BERNOULLI(25)
WHERE  ee.empno = ea.empno
ORDER BY ee.empno;
```

Figure 950, Sample rows in two tables

Sample a declared global temporary table, and also apply other predicates:

```
DECLARE GLOBAL TEMPORARY TABLE session.nyc_staff
LIKE   staff;

SELECT *
FROM   session.nyc_staff TABLESAMPLE SYSTEM(34.55)
WHERE  id < 100
       AND salary > 100
ORDER BY id;
```

Figure 951, Sample Views used in Join Examples

Convert Character to Numeric

The DOUBLE, DECIMAL, INTEGER, SMALLINT, and BIGINT functions can all be used to convert a character field into its numeric equivalent:

```

WITH temp1 (c1) AS
  (VALUES '123 ', '345 ', '567')
SELECT c1
      ,DOUBLE(c1)      AS dbl
      ,DECIMAL(c1,3)  AS dec
      ,SMALLINT(c1)   AS sml
      ,INTEGER(c1)    AS int
FROM   temp1;

```

ANSWER (numbers shortened)					
C1	DBL	DEC	SML	INT	
123	+1.2300E+2	123.	123	123	
345	+3.4500E+2	345.	345	345	
567	+5.6700E+2	567.	567	567	

Figure 952, Convert 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 953, Acceptable conversion values

Checking the Input

There are several ways to check that the input character string is a valid representation of a number - before doing the conversion. One simple solution involves converting all digits to blank, then removing the blanks. If the result is not a zero length string, then the input must have had a character other than a digit:

```

WITH temp1 (c1) AS (VALUES '123', '456 ', '1 2', '33%', NULL)
SELECT c1
      ,TRANSLATE(c1, ' ', '1234567890') AS c2
      ,LENGTH(LTRIM(TRANSLATE(c1, ' ', '1234567890'))) AS c3
FROM   temp1;

```

ANSWER		
C1	C2	C3
123		0
456		0
1 2		0
33%	%	1

Figure 954, Checking for non-digits

One can also write a user-defined scalar function to check for non-numeric input, which is what is done below. This function returns "Y" if the following is true:

- The input is not null.
- There are no non-numeric characters in the input.
- The only blanks in the input are to the left of the digits.
- There is only one "+" or "-" sign, and it is next to the left-side blanks, if any.
- There is at least one digit in the input.

Now for the code:

```

--#SET DELIMITER !

CREATE FUNCTION isnumeric(instr VARCHAR(40))
RETURNS CHAR(1)
BEGIN ATOMIC
  DECLARE is_number CHAR(1) DEFAULT 'Y';
  DECLARE bgn_blank CHAR(1) DEFAULT 'Y';
  DECLARE found_num CHAR(1) DEFAULT 'N';
  DECLARE found_pos CHAR(1) DEFAULT 'N';
  DECLARE found_neg CHAR(1) DEFAULT 'N';
  DECLARE found_dot CHAR(1) DEFAULT 'N';
  DECLARE ctr SMALLINT DEFAULT 1;
  IF instr IS NULL THEN
    RETURN NULL;
  END IF;
  wloop:
  WHILE ctr <= LENGTH(instr) AND
    is_number = 'Y'
  DO
    -----
    --- ERROR CHECKS ---
    -----
    IF SUBSTR(instr,ctr,1) NOT IN (' ','.','+','-','0','1','2',
      '3','4','5','6','7','8','9') THEN
      SET is_number = 'N';
      ITERATE wloop;
    END IF;
    IF SUBSTR(instr,ctr,1) = ' ' AND
      bgn_blank = 'N' THEN
      SET is_number = 'N';
      ITERATE wloop;
    END IF;
    IF SUBSTR(instr,ctr,1) = '.' AND
      found_dot = 'Y' THEN
      SET is_number = 'N';
      ITERATE wloop;
    END IF;
    IF SUBSTR(instr,ctr,1) = '+' AND
      (found_neg = 'Y' OR
      bgn_blank = 'N') THEN
      SET is_number = 'N';
      ITERATE wloop;
    END IF;
    IF SUBSTR(instr,ctr,1) = '-' AND
      (found_neg = 'Y' OR
      bgn_blank = 'N') THEN
      SET is_number = 'N';
      ITERATE wloop;
    END IF;
    -----
    --- MAINTAIN FLAGS & CTR ---
    -----
    IF SUBSTR(instr,ctr,1) IN ('0','1','2','3','4',
      '5','6','7','8','9') THEN
      SET found_num = 'Y';
    END IF;
    IF SUBSTR(instr,ctr,1) = '.' THEN
      SET found_dot = 'Y';
    END IF;
    IF SUBSTR(instr,ctr,1) = '+' THEN
      SET found_pos = 'Y';
    END IF;
    IF SUBSTR(instr,ctr,1) = '-' THEN
      SET found_neg = 'Y';
    END IF;
  END IF;

```

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

Figure 955, Check Numeric function, part 1 of 2

```

        IF SUBSTR(instr,ctr,1) <> ' ' THEN
            SET bgn_blank = 'N';
        END IF;
        SET ctr = ctr + 1;
    END WHILE wloop;
    IF found_num = 'N' THEN
        SET is_number = 'N';
    END IF;
    RETURN is_number;
END!

WITH TEMP1 (C1) AS
(VVALUES ' 123'
, '+123.45'
, '456'
, ' 10 2 '
, ' -.23'
, '++12356'
, '.012349'
, ' 33%'
, ' '
, NULL)
SELECT C1 AS C1
, isnumeric(C1) AS C2
, CASE
    WHEN isnumeric(C1) = 'Y'
    THEN DECIMAL(C1,10,6)
    ELSE NULL
    END AS C3
FROM TEMP1!

```

			ANSWER		
			C1	C2	C3
			123	Y	123.00000
			+123.45	Y	123.45000
			456	N	-
			10 2	N	-
			-.23	Y	-0.23000
			++12356	N	-
			.012349	Y	0.01234
			33%	N	-
				N	-
			-	-	-

Figure 956, Check Numeric function, part 2 of 2

NOTE: See page 178 for a much simpler function that is similar to the above.

Convert Number to Character

The CHAR and DIGITS functions can be used to convert a DB2 numeric field to a character representation of the same, but as the following example demonstrates, both functions return problematic output:

```

SELECT d_sal
, CHAR(d_sal) AS d_chr
, DIGITS(d_sal) AS d_dgt
, i_sal
, CHAR(i_sal) AS i_chr
, DIGITS(i_sal) AS i_dgt
FROM (SELECT DEC(salary - 11000,6,2) AS d_sal
, SMALLINT(salary - 11000) AS i_sal
FROM staff
WHERE salary > 10000
AND salary < 12200
)AS xxx
ORDER BY d_sal;

```

						ANSWER
D_SAL	D_CHR	D_DGT	I_SAL	I_CHR	I_DGT	
-494.10	-0494.10	049410	-494	-494	00494	
-12.00	-0012.00	001200	-12	-12	00012	
508.60	0508.60	050860	508	508	00508	
1009.75	1009.75	100975	1009	1009	01009	

Figure 957, CHAR and DIGITS function usage

The DIGITS function discards both the sign indicator and the decimal point, while the CHAR function output is (annoyingly) left-justified, and (for decimal data) has leading zeros. We can do better.

Below are three user-defined functions that convert integer data from numeric to character, displaying the output right-justified, and with a sign indicator if negative. There is one function for each flavor of integer that is supported in DB2:

```
CREATE FUNCTION char_right(inval SMALLINT)
RETURNS CHAR(06)
RETURN  RIGHT(CHAR(' ',06) CONCAT RTRIM(CHAR(inval)),06);

CREATE FUNCTION char_right(inval INTEGER)
RETURNS CHAR(11)
RETURN  RIGHT(CHAR(' ',11) CONCAT RTRIM(CHAR(inval)),11);

CREATE FUNCTION char_right(inval BIGINT)
RETURNS CHAR(20)
RETURN  RIGHT(CHAR(' ',20) CONCAT RTRIM(CHAR(inval)),20);
```

Figure 958, User-defined functions - convert integer to character

Each of the above functions works the same way (working from right to left):

- First, convert the input number to character using the CHAR function.
- Next, use the RTRIM function to remove the right-most blanks.
- Then, concatenate a set number of blanks to the left of the value. The number of blanks appended depends upon the input type, which is why there are three separate functions.
- Finally, use the RIGHT function to get the right-most "n" characters, where "n" is the maximum number of digits (plus the sign indicator) supported by the input type.

The next example uses the first of the above functions:

SELECT	i_sal	ANSWER
	,char_right(i_sal) AS i_chr	=====
FROM	(SELECT SMALLINT(salary - 11000) AS i_sal	I_SAL I_CHR
	FROM staff	-----
	WHERE salary > 10000	-494 -494
	AND salary < 12200	-12 -12
)AS xxx	508 508
ORDER BY	i_sal;	1009 1009

Figure 959, Convert SMALLINT to CHAR

Decimal Input

Creating a similar function to handle decimal input is a little more tricky. One problem is that the CHAR function adds leading zeros to decimal data, which we don't want. A more serious problem is that there are many sizes and scales of decimal data, but we can only create one function (with a given name) for a particular input data type.

Decimal values can range in both length and scale from 1 to 31 digits. This makes it impossible to define a single function to convert any possible decimal value to character with possibly running out of digits, or losing some precision.

NOTE: The fact that one can only have one user-defined function, with a given name, per DB2 data type, presents a problem for all variable-length data types - notably character, varchar, and decimal. For character and varchar data, one can address the problem, to some extent, by using maximum length input and output fields. But decimal data has both a scale and a length, so there is no way to make an all-purpose decimal function.

Despite the above, below is a function that converts decimal data to character. It compromises by assuming an input of type decimal(22,2), which should handle most monetary values:

```

CREATE FUNCTION char_right(inval DECIMAL(22,2))
RETURNS CHAR(23)
RETURN  RIGHT(CHAR(' ',20) CONCAT RTRIM(CHAR(BIGINT(inval))),20)
        CONCAT '.'
        CONCAT SUBSTR(DIGITS(inval),21,2);

```

Figure 960, User-defined function - convert decimal to character

The function works as follows:

- The non-fractional part of the number is converted to BIGINT, then converted to CHAR as previously described.
- A period (dot) is added to the back of the output.
- The fractional digits (converted to character using the DIGITS function) are appended to the back of the output.

Below is the function in action:

```

SELECT  d_sal
        ,char_right(d_sal)  AS d_chr
FROM    (SELECT DEC(salary - 11000,6,2) AS d_sal
        FROM    staff
        WHERE   salary > 10000
        AND    salary < 12200
        )AS xxx
ORDER BY d_sal;

```

ANSWER	
=====	
D_SAL	D_CHR

-494.10	-494.10
-12.00	-12.00
508.60	508.60
1009.75	1009.75

Figure 961, Convert DECIMAL to CHAR

Floating point data can be processed using the above function, as long as it is first converted to decimal using the standard DECIMAL function.

Adding Commas

The next function converts decimal input to character, with embedded commas. It first converts the value to character - as per the above function. It then steps through the output string, three bytes at a time, from right to left, checking to see if the next-left character is a number. If it is, it inserts a comma, else it adds a blank byte to the front of the string:

```

CREATE FUNCTION comma_right(inval DECIMAL(20,2))
RETURNS CHAR(27)
LANGUAGE SQL
DETERMINISTIC
NO EXTERNAL ACTION
BEGIN ATOMIC
  DECLARE i INTEGER DEFAULT 17;
  DECLARE abs_inval BIGINT;
  DECLARE out_value CHAR(27);
  SET abs_inval = ABS(BIGINT(inval));
  SET out_value = RIGHT(CHAR(' ',19) CONCAT
    RTRIM(CHAR(BIGINT(inval))),19)
    CONCAT '.'
    CONCAT SUBSTR(DIGITS(inval),19,2);
  WHILE i > 2 DO
    IF SUBSTR(out_value,i-1,1) BETWEEN '0' AND '9' THEN
      SET out_value = SUBSTR(out_value,1,i-1) CONCAT
        ','
        CONCAT
          SUBSTR(out_value,i);
    ELSE
      SET out_value = ' ' CONCAT out_value;
    END IF;
    SET i = i - 3;
  END WHILE;
  RETURN out_value;
END

```

Figure 962, User-defined function - convert decimal to character - with commas

Below is an example of the above function in use:

	ANSWER	
	INPUT	OUTPUT
WITH temp1 (num) AS (VALUES (DEC(+1,20,2)) , (DEC(-1,20,2)))		
UNION ALL	-975460660753.97	-975,460,660,753.97
SELECT num * 987654.12	-987655.12	-987,655.12
FROM temp1	-2.00	-2.00
WHERE ABS(num) < 1E10),	0.00	0.00
temp2 (num) AS (SELECT num - 1 FROM temp1)	987653.12	987,653.12
SELECT num AS input , comma_right(num) AS output	975460660751.97	975,460,660,751.97
FROM temp2		
ORDER BY num;		

Figure 963, Convert DECIMAL to CHAR with commas

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 964, Convert 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 965, 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 966, Sample query with column control

Making Charts Using SQL

Imagine that one had a string of numeric values that one wants to display as a line-bar chart. With a little coding, this is easy to do in SQL:

```
SELECT  id
        ,salary
        ,INT(salary / 1500) AS len
        ,REPEAT('*',INT(salary / 1500)) AS salary_chart
FROM    staff
WHERE   id > 120
        AND id < 190
ORDER BY id;
```

ANSWER

ID	SALARY	LEN	SALARY_CHART
130	10505.90	7	*****
140	21150.00	14	*****
150	19456.50	12	*****
160	22959.20	15	*****
170	12258.50	8	*****
180	12009.75	8	*****

Figure 967, Make chart using SQL

To create the above graph we first converted the column of interest to an integer field of a manageable length, and then used this value to repeat a single "*" character a set number of times.

One problem with the above query is that we won't know how long the chart will be until we run the statement. This may cause problems if we guess wrongly and we are tight for space, so the next query addresses this issue by creating a chart of known length. To do this, it does the following:

- First select all of the matching rows and columns and store them in a temporary table.
- Next, obtain the MAX value from the field of interest. Then convert this value to an integer and divide by the maximum desired chart length (e.g. 20).
- Finally, join the two temporary tables together and display the chart. Because the chart will never be longer than 20 bytes, we can display it in a 20 byte field.

Now for the code:

```

                                ANSWER
                                =====
                                ID      SALARY  SALARY_CHART
                                -----
WITH
temp1 (id, salary) AS
  (SELECT   id
   ,salary
   FROM     staff
   WHERE    id > 120
   AND      id < 190),
temp2 (max_sal) AS
  (SELECT   INT(MAX(salary)) / 20
   FROM     temp1)
SELECT     id
   ,salary
   ,VARCHAR(REPEAT('*',INT(salary / max_sal)),20) AS salary_chart
FROM       temp1
   ,temp2
ORDER BY  id;

```

Figure 968, Make chart of fixed length

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
   ,COUNT(*) AS num
FROM       stats
GROUP BY  sex
ORDER BY  sex;
                                ANSWER >>
                                SEX NUM
                                --- ---
                                F 595
                                M 405

```

Figure 969, 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 970, 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 971, 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 972, Use CASE and SUM to get counts

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 973, 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	(SELECT	2 0
	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 974, 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
  (VALUES SMALLINT(0)
   UNION ALL
   SELECT year# + 1
   FROM list_years
   WHERE year# < 5)
SELECT year# AS years
      ,COALESCE(#stff,0) AS #staff
FROM list_years
LEFT OUTER JOIN
  (SELECT years
    ,COUNT(*) AS #stff
   FROM staff
   WHERE UCASE(name) LIKE '%E%'
   AND years <= 5
   GROUP BY years
  )AS xxx
ON year# = years
ORDER BY 1;

```

ANSWER	
=====	
YEARS	#STAFF
-----	-----
0	0
1	1
2	0
3	0
4	2
5	3

Figure 975, 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
  (VALUES SMALLINT(0)
   UNION ALL
   SELECT year# + 1
   FROM list_years
   WHERE year# < 5)
SELECT year#
FROM list_years y
WHERE NOT EXISTS
  (SELECT *
   FROM staff s
   WHERE UCASE(s.name) LIKE '%E%'
   AND s.years = y.year#)
ORDER BY 1;

```

ANSWER	
=====	
YEAR#	

0	
2	
3	

Figure 976, List years when no staff joined

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 977, 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 978, Multiple counts in one pass, Answer

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 979, 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 980, Break text into words - Answer

Denormalize Normalized Data

In the next example, we shall use recursion to string together all of the employee NAME fields in the STAFF table (by department):

```

WITH temp1 (dept,w#,name,all_names) AS
(SELECT   dept
         ,SMALLINT(1)
         ,MIN(name)
         ,VARCHAR(MIN(name),50)
FROM     staff a
GROUP BY dept
UNION ALL
SELECT   a.dept
         ,SMALLINT(b.w#+1)
         ,a.name
         ,b.all_names || ' ' || a.name
FROM     staff a
         ,temp1 b
WHERE    a.dept = b.dept
        AND a.name > b.name
        AND a.name =
        (SELECT MIN(c.name)
         FROM   staff c
         WHERE  c.dept = b.dept
         AND   c.name > b.name)
)
SELECT   dept
         ,w#
         ,name AS max_name
         ,all_names
FROM     temp1 d
WHERE    w# =
        (SELECT MAX(w#)
         FROM   temp1 e
         WHERE  d.dept = e.dept)
ORDER BY dept;

```

Figure 981, Denormalize Normalized Data - SQL

The above statement begins by getting the minimum name in each department. It then recursively gets the next to lowest name, then the next, and so on. As we progress, we store the current name in the temporary NAME field, maintain a count of names added, and append the same to the end of the ALL_NAMES field. Once we have all of the names, the final SELECT eliminates from the answer-set all rows, except the last for each department.

DEPT	W#	MAX_NAME	ALL_NAMES
10	4	Molinare	Daniels Jones Lu Molinare
15	4	Rothman	Hanes Kermisch Ngan Rothman
20	4	Sneider	James Pernal Sanders Sneider
38	5	Quigley	Abrahams Marengi 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 982, Denormalize Normalized Data - Answer

If there are no suitable indexes, the above query may be horribly inefficient. If this is the case, one can create a user-defined function to string together the names in a department:

```

CREATE FUNCTION list_names(indept SMALLINT)
RETURNS VARCHAR(50)
BEGIN ATOMIC
  DECLARE outstr VARCHAR(50) DEFAULT '';
  FOR list_names AS
    SELECT name
    FROM staff
    WHERE dept = indept
    ORDER BY name
  DO
    SET outstr = outstr || name || ' ';
  END FOR;
  SET outstr = rtrim(outstr);
  RETURN outstr;
END!

SELECT dept AS DEPT
      ,SMALLINT(cnt) AS W#
      ,mxx AS MAX_NAME
      ,list_names(dept) AS ALL_NAMES
FROM (SELECT dept
      ,COUNT(*) as cnt
      ,MAX(name) AS mxx
      FROM staff
      GROUP BY dept
      )as ddd
ORDER BY dept!

```

```

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

```

Figure 983, Creating a function to denormalize names

Even the above might have unsatisfactory performance - if there is no index on department. If adding an index to the STAFF table is not an option, it might be faster to insert all of the rows into a declared temporary table, and then add an index to that.

Transpose Numeric Data

In this section we will turn rows of numeric data into columns. This cannot be done directly in SQL because the language does not support queries where the output columns are unknown at query start. We will get around this limitation by sending the transposed output to a suitably long VARCHAR field.

Imagine that we want to group the data in the STAFF sample table by DEPT and JOB to get the SUM salary for each instance, but not in the usual sense with one output row per DEPT and JOB value. Instead, we want to generate one row per DEPT, with a set of "columns" (in a VARCHAR field) that hold the SUM salary values for each JOB in the department. We will also put column titles on the first line of output.

To make the following query simpler, three simple scalar functions will be used to convert data from one type to another:

- Convert decimal data to character - similar to the one on page 377.
- Convert smallint data to character - same as the one page 377.
- Right justify and add leading blanks to character data.

Now for the functions:

```

CREATE FUNCTION num_to_char(ival SMALLINT)
RETURNS CHAR(06)
RETURN RIGHT(CHAR(' ',06) CONCAT RTRIM(CHAR(ival)),06);

CREATE FUNCTION num_to_char(ival DECIMAL(9,2))
RETURNS CHAR(10)
RETURN RIGHT(CHAR(' ',7) CONCAT RTRIM(CHAR(BIGINT(ival))),7)
        CONCAT '.'
        CONCAT SUBSTR(DIGITS(ival),8,2);

CREATE FUNCTION right_justify(ival CHAR(5))
RETURNS CHAR(10)
RETURN RIGHT(CHAR(' ',10) || RTRIM(ival),10);

```

Figure 984, Data Transformation Functions

The query consists of lots of little steps that are best explained by describing each temporary table built:

- **DATA_INPUT:** This table holds the set of matching rows in the STAFF table, grouped by DEPT and JOB as per a typical query (see page 390 for the contents). This is the only time that we touch the original STAFF table. All subsequent queries directly or indirectly reference this table.
- **JOBS_LIST:** The list of distinct jobs in all matching rows. Each job is assigned two row-numbers, one ascending, and one descending.
- **DEPT_LIST:** The list of distinct departments in all matching rows.
- **DEPT_JOB_LIST:** The list of all matching department/job combinations. We need this table because not all departments have all jobs.
- **DATA_ALL_JOBS:** The DEPT_JOB_LIST table joined to the original DATA_INPUT table using a left outer join, so we now have one row with a sum-salary value for every JOB and DEPT instance.
- **DATA_TRANSFORM:** Recursively go through the DATA_ALL_JOBS table (for each department), adding the a character representation of the current sum-salary value to the back of a VARCHAR column.
- **DATA_LAST_ROW:** For each department, get the row with the highest ascending JOB# value. This row has the concatenated string of sum-salary values.

At this point we are done, except that we don't have any column headings in our output. The rest of the query gets these.

- **JOBS_TRANSFORM:** Recursively go through the list of distinct jobs, building a VARCHAR string of JOB names. The job names are right justified - to match the sum-salary values, and have the same output length.
- **JOBS_LAST_ROW:** Get the one row with the lowest descending job number. This row has the complete string of concatenated job names.
- **DATA_AND_JOBS:** Use a UNION ALL to vertically combine the JOBS_LAST_ROW and DATA_LAST_ROW tables. The result is a new table with both column titles and sum-salary values.

Finally, we select the list of column names and sum-salary values. The output is ordered so that the column names are on the first line fetched.

Now for the query:

```

WITH
data_input AS
  (SELECT   dept
           ,job
           ,SUM(salary) AS sum_sal
    FROM     staff
   WHERE    id      < 200
           AND     name <> 'Sue'
           AND     salary > 10000
   GROUP BY dept
           ,job),
jobs_list AS
  (SELECT   job
           ,ROW_NUMBER() OVER(ORDER BY job ASC) AS job#A
           ,ROW_NUMBER() OVER(ORDER BY job DESC) AS job#D
    FROM     data_input
   GROUP BY job),
dept_list AS
  (SELECT   dept
    FROM     data_input
   GROUP BY dept),
dept_jobs_list AS
  (SELECT   dpt.dept
           ,job.job
           ,job.job#A
           ,job.job#D
    FROM     jobs_list job
   FULL OUTER JOIN
           dept_list dpt
    ON      1 = 1),
data_all_jobs AS
  (SELECT   djb.dept
           ,djb.job
           ,djb.job#A
           ,djb.job#D
           ,COALESCE(dat.sum_sal,0) AS sum_sal
    FROM     dept_jobs_list djb
   LEFT OUTER JOIN
           data_input dat
    ON      djb.dept = dat.dept
           AND     djb.job = dat.job),
data_transform (dept, job#A, job#D, outvalue) AS
  (SELECT   dept
           ,job#A
           ,job#D
           ,VARCHAR(num_to_char(sum_sal),250)
    FROM     data_all_jobs
   WHERE    job#A = 1
   UNION ALL
  SELECT   dat.dept
           ,dat.job#A
           ,dat.job#D
           ,trn.outvalue || ',' || num_to_char(dat.sum_sal)
    FROM     data_transform trn
           ,data_all_jobs dat
   WHERE    trn.dept = dat.dept
           AND     trn.job#A = dat.job#A - 1),
data_last_row AS
  (SELECT   dept
           ,num_to_char(dept) AS dept_char
           ,outvalue
    FROM     data_transform
   WHERE    job#D = 1),

```

Figure 985, Transform numeric data - part 1 of 2

```

jobs_transform (job#A, job#D, outvalue) AS
  (SELECT   job#A
    ,job#D
    ,VARCHAR(right_justify(job),250)
  FROM     jobs_list
  WHERE    job#A = 1
  UNION ALL
  SELECT   job.job#A
    ,job.job#D
    ,trn.outvalue || ',' || right_justify(job.job)
  FROM     jobs_transform trn
    ,jobs_list      job
  WHERE    trn.job#A = job.job#A - 1),
jobs_last_row AS
  (SELECT   0          AS dept
    , '  DEPT' AS dept_char
    ,outvalue
  FROM     jobs_transform
  WHERE    job#D = 1),
data_and_jobs AS
  (SELECT   dept
    ,dept_char
    ,outvalue
  FROM     jobs_last_row
  UNION ALL
  SELECT   dept
    ,dept_char
    ,outvalue
  FROM     data_last_row)
SELECT   dept_char || ',' ||
  outvalue AS output
FROM     data_and_jobs
ORDER BY dept;

```

Figure 986, Transform numeric data - part 2 of 2

For comparison, below is the contents of the first temporary table, and the final output:

DATA_INPUT			OUTPUT			
=====			=====			
DEPT	JOB	SUM_SAL	DEPT,	Clerk,	Mgr,	Sales
----	----	----	----	----	----	----
10	Mgr	22959.20	10,	0.00,	22959.20,	0.00
15	Clerk	24766.70	15,	24766.70,	20659.80,	16502.83
15	Mgr	20659.80	20,	27757.35,	18357.50,	18171.25
15	Sales	16502.83	38,	24964.50,	17506.75,	34814.30
20	Clerk	27757.35	42,	10505.90,	18352.80,	18001.75
20	Mgr	18357.50	51,	0.00,	21150.00,	19456.50
20	Sales	18171.25				
38	Clerk	24964.50				
38	Mgr	17506.75				
38	Sales	34814.30				
42	Clerk	10505.90				
42	Mgr	18352.80				
42	Sales	18001.75				
51	Mgr	21150.00				
51	Sales	19456.50				

Figure 987, Contents of first temporary table and final output

Reversing Field Contents

DB2 lacks a simple function for reversing the contents of a data field. Fortunately, we can create a function to do it ourselves.

Input vs. Output

Before we do any data reversing, we have to define what the reversed output should look like relative to a given input value. For example, if we have a four-digit numeric field, the reverse of the number 123 could be 321, or it could be 3210. The latter value implies that the input has a leading zero. It also assumes that we really are working with a four digit field. Likewise, the reverse of the number 123.45 might be 54.321, or 543.21.

Another interesting problem involves reversing negative numbers. If the value "-123" is a string, then the reverse is probably "321-". If it is a number, then the desired reverse is more likely to be "-321".

Trailing blanks in character strings are a similar problem. Obviously, the reverse of "ABC" is "CBA", but what is the reverse of "ABC "? There is no general technical answer to any of these questions. The correct answer depends upon the business needs of the application.

Below is a user defined function that can reverse the contents of a character field:

```
--#SET DELIMITER !                                IMPORTANT
                                                    =====
CREATE FUNCTION reverse(instr VARCHAR(50))         This example
RETURNS VARCHAR(50)                               uses an "!"
BEGIN ATOMIC                                       as the stmt
  DECLARE outstr  VARCHAR(50) DEFAULT '';          delimiter.
  DECLARE curbyte SMALLINT  DEFAULT 0;
  SET curbyte = LENGTH(RTRIM(instr));
  WHILE curbyte >= 1 DO
    SET outstr = outstr || SUBSTR(instr,curbyte,1);
    SET curbyte = curbyte - 1;
  END WHILE;
  RETURN outstr;
END!
```

```
SELECT  id          AS ID
        ,name       AS NAME1
        ,reverse(name) AS NAME2
FROM    staff
WHERE   id < 40
ORDER  BY id!
```

```
ANSWER
=====
ID NAME1      NAME2
--  -
10 Sanders   srednaS
20 Pernal    lanreP
30 Marenghi ihgneraM
```

Figure 988, Reversing character field

The same function can be used to reverse numeric values, as long as they are positive:

```
SELECT  id          AS ID
        ,salary     AS SALARY1
        ,DEC(reverse(CHAR(salary)),7,4) AS SALARY2
FROM    staff
WHERE   id < 40
ORDER  BY id;
```

```
ANSWER
=====
ID SALARY1    SALARY2
--  -
10 18357.50   5.7538
20 18171.25   52.1718
30 17506.75   57.6057
```

Figure 989, Reversing numeric field

Simple CASE logic can be used to deal with negative values (i.e. to move the sign to the front of the string, before converting back to numeric), if they exist.

Fibonacci Series

A Fibonacci Series is a series of numbers where each value is the sum of the previous two. Regardless of the two initial (seed) values, if run for long enough, the division of any two adjacent numbers will give the value 0.618 or inversely 1.618.

The following user defined function generates a Fibonacci series using three input values:

- First seed value.
- Second seed value.
- Number values to generate in series.

Observe that that the function code contains a check to stop series generation if there is not enough space in the output field for more numbers:

```
--#SET DELIMITER !                                IMPORTANT
                                                    =====
CREATE FUNCTION Fibonacci (inval1 INTEGER          This example
                        ,inval2 INTEGER          uses an "!"
                        ,loopno INTEGER)         as the stmt
RETURNs VARCHAR(500)                             delimiter.
BEGIN ATOMIC
  DECLARE loopctr INTEGER DEFAULT 0;
  DECLARE tempval1 BIGINT;
  DECLARE tempval2 BIGINT;
  DECLARE tempval3 BIGINT;
  DECLARE outvalue VARCHAR(500);
  SET tempval1 = inval1;
  SET tempval2 = inval2;
  SET outvalue = RTRIM(LTRIM(CHAR(tempval1))) || ', ' ||
                RTRIM(LTRIM(CHAR(tempval2)));
  calc: WHILE loopctr < loopno DO
    SET tempval3 = tempval1 + tempval2;
    SET tempval1 = tempval2;
    SET tempval2 = tempval3;
    SET outvalue = outvalue || ', ' || RTRIM(LTRIM(CHAR(tempval3)));
    SET loopctr = loopctr + 1;
    IF LENGTH(outvalue) > 480 THEN
      SET outvalue = outvalue || ' etc...';
      LEAVE calc;
    END IF;
  END WHILE;
  RETURN outvalue;
END!
```

Figure 990, Fibonacci Series function

The following query references the function:

```
WITH temp1 (v1,v2,lp) AS
  (VALUES (00,01,11)
         ,(12,61,10)
         ,(02,05,09)
         ,(01,-1,08))
SELECT t1.*
      ,Fibonacci(v1,v2,lp) AS sequence
FROM   temp1 t1;
```

ANSWER

```
=====
V1 V2 LP SEQUENCE
-----
 0  1 11 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144
12 61 10 12, 61, 73, 134, 207, 341, 548, 889, 1437, 2326, 3763, 6089
 2  5  9  2, 5, 7, 12, 19, 31, 50, 81, 131, 212, 343
 1 -1  8  1, -1, 0, -1, -1, -2, -3, -5, -8, -13
```

Figure 991, Fibonacci Series generation

The above example generates the complete series of values. If needed, the code could easily be simplified to simply return only the last value in the series. Likewise, a recursive join can be used to create a set of rows that are a Fibonacci series.

Business Day Calculation

The following function will calculate the number of business days (i.e. Monday to Friday) between two dates:

```
CREATE FUNCTION business_days (lo_date DATE, hi_date DATE)
RETURNS INTEGER
BEGIN ATOMIC
  DECLARE bus_days INTEGER DEFAULT 0;
  DECLARE cur_date DATE;
  SET cur_date = lo_date;
  WHILE cur_date < hi_date DO
    IF DAYOFWEEK(cur_date) IN (2,3,4,5,6) THEN
      SET bus_days = bus_days + 1;
    END IF;
    SET cur_date = cur_date + 1 DAY;
  END WHILE;
  RETURN bus_days;
END!
```

IMPORTANT
=====

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

Figure 992, Calculate number of business days between two dates

Below is an example of the function in use:

```
WITH temp1 (ld, hd) AS
  (VALUES (DATE('2006-01-10'),DATE('2007-01-01'))
        ,(DATE('2007-01-01'),DATE('2007-01-01'))
        ,(DATE('2007-02-10'),DATE('2007-01-01')))
SELECT t1.*
      ,DAYS(hd) - DAYS(ld) AS diff
      ,business_days(ld,hd) AS bdays
FROM   temp1 t1;
```

				ANSWER
				=====
LD	HD	DIFF	BDAYS	

2006-01-10	2007-01-01	356	254	
2007-01-01	2007-01-01	0	0	
2007-02-10	2007-01-01	-40	0	

Figure 993, Use business-day function

Query Runs for "n" Seconds

Imagine that one wanted some query to take exactly four seconds to run. The following query does just this - by looping (using recursion) until such time as the current system timestamp is four seconds greater than the system timestamp obtained at the beginning of the query:

```
WITH temp1 (num,ts1,ts2) AS
  (VALUES (INT(1)
        ,TIMESTAMP(GENERATE_UNIQUE())
        ,TIMESTAMP(GENERATE_UNIQUE()))
  UNION ALL
  SELECT num + 1
        ,ts1
        ,TIMESTAMP(GENERATE_UNIQUE())
  FROM   temp1
  WHERE  TIMESTAMPDIFF(2,CHAR(ts2-ts1)) < 4
  )
SELECT MAX(num) AS #loops
      ,MIN(ts2) AS bgn_timestamp
      ,MAX(ts2) AS end_timestamp
FROM   temp1;
```

			ANSWER
			=====
#LOOPS	BGN_TIMESTAMP	END_TIMESTAMP	

58327	2001-08-09-22.58.12.754579	2001-08-09-22.58.16.754634	

Figure 994, 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.

Function to Pause for "n" Seconds

We can take the above query and convert it into a user-defined function that will loop for "n" seconds, where "n" is the value passed to the function. However, there are several caveats:

- Looping in SQL is a "really stupid" way to hang around for a couple of seconds. A far better solution would be to call a stored procedure written in an external language that has a true pause command.
- The number of times that the function is invoked may differ, depending on the access path used to run the query.
- The recursive looping is going to result in the calling query getting a warning message.

Now for the code:

```
CREATE FUNCTION pause(inval INT)
RETURNS INTEGER
NOT DETERMINISTIC
EXTERNAL ACTION
RETURN
WITH ttt (num, strt, stop) AS
  (VALUES (1
           ,TIMESTAMP(GENERATE_UNIQUE()))
           ,TIMESTAMP(GENERATE_UNIQUE()))
  UNION ALL
  SELECT  num + 1
           ,strt
           ,TIMESTAMP(GENERATE_UNIQUE())
  FROM    ttt
  WHERE   TIMESTAMPDIF(2,CHAR(stop - strt)) < inval
  )
SELECT  MAX(num)
FROM    ttt;
```

Figure 995, Function that pauses for "n" seconds

Below is a query that calls the above function:

```
SELECT  id
        ,SUBSTR(CHAR(TIMESTAMP(GENERATE_UNIQUE())),18) AS ss_mmmmmm
        ,pause(id / 10) AS #loops
        ,SUBSTR(CHAR(TIMESTAMP(GENERATE_UNIQUE())),18) AS ss_mmmmmm
FROM    staff
WHERE   id < 31;
```

ANSWER

```
=====
ID SS_MMMMMM #LOOPS SS_MMMMMM
-----
10 50.068593  76386 50.068587
20 52.068744 144089 52.068737
30 55.068930 206101 55.068923
```

Figure 996, Query that uses pause function

Sort Character Field Contents

The following user-defined scalar function will sort the contents of a character field in either ascending or descending order. There are two input parameters:

- The input string: As written, the input can be up to 20 bytes long. To sort longer fields, change the input, output, and OUT-VAL (variable) lengths as desired.

- The sort order (i.e. 'A' or 'D').

The function uses a very simple, and not very efficient, bubble-sort. In other words, the input string is scanned from left to right, comparing two adjacent characters at a time. If they are not in sequence, they are swapped - and flag indicating this is set on. The scans are repeated until all of the characters in the string are in order:

```
--#SET DELIMITER !

CREATE FUNCTION sort_char(in_val VARCHAR(20),sort_dir VARCHAR(1))
RETURNS VARCHAR(20)
BEGIN ATOMIC
  DECLARE cur_pos SMALLINT;
  DECLARE do_sort CHAR(1);
  DECLARE out_val VARCHAR(20);
  IF UCASE(sort_dir) NOT IN ('A','D') THEN
    SIGNAL SQLSTATE '75001'
    SET MESSAGE_TEXT = 'Sort order not ''A'' or ''D''';
  END IF;
  SET out_val = in_val;
  SET do_sort = 'Y';
  WHILE do_sort = 'Y' DO
    SET do_sort = 'N';
    SET cur_pos = 1;
    WHILE cur_pos < length(in_val) DO
      IF (UCASE(sort_dir) = 'A'
        AND SUBSTR(out_val,cur_pos+1,1) <
          SUBSTR(out_val,cur_pos,1))
        OR (UCASE(sort_dir) = 'D'
        AND SUBSTR(out_val,cur_pos+1,1) >
          SUBSTR(out_val,cur_pos,1)) THEN
        SET do_sort = 'Y';
        SET out_val = CASE
          WHEN cur_pos = 1
            THEN ''
          ELSE SUBSTR(out_val,1,cur_pos-1)
        END
        CONCAT SUBSTR(out_val,cur_pos+1,1)
        CONCAT SUBSTR(out_val,cur_pos,1)
        CONCAT
        CASE
          WHEN cur_pos = length(in_val) - 1
            THEN ''
          ELSE SUBSTR(out_val,cur_pos+2)
        END;
      END IF;
      SET cur_pos = cur_pos + 1;
    END WHILE;
  END WHILE;
  RETURN out_val;
END!
```

IMPORTANT
=====

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

Figure 997, Define sort-char function

Here is the function in action:

```

WITH word1 (w#, word_val) AS
  (VALUES(1, '12345678')
    , (2, 'ABCDEFGH')
    , (3, 'AaBbCc')
    , (4, 'abccb')
    , (5, '%#. ')
    , (6, 'bB')
    , (7, 'a')
    , (8, ''))
SELECT  w#
        ,word_val
        ,sort_char(word_val, 'a') sa
        ,sort_char(word_val, 'D') sd
FROM    word1
ORDER BY w#;

```

```

ANSWER
=====
W# WORD_VAL SA          SD
-----
1 12345678 12345678 87654321
2 ABCDEFGH ABCDEFGH GFEDCBA
3 AaBbCc   aAbBcC   CcBbAa
4 abccb    abbcc    ccbba
5 '%#. '   .'#%     %#'.
6 bB       bB       Bb
7 a        a        a
8

```

Figure 998, Use sort-char function

Calculating the Median

The median is defined as that value in a series of values where half of the values are higher to it and the other half are lower. The median is a useful number to get when the data has a few very extreme values that skew the average.

If there are an odd number of values in the list, then the median value is the one in the middle (e.g. if 7 values, the median value is #4). If there is an even number of matching values, there are two formulas that one can use:

- The most commonly used definition is that the median equals the sum of the two middle values, divided by two.
- A less often used definition is that the median is the smaller of the two middle values.

DB2 does not come with a function for calculating the median, but it can be obtained using the ROW_NUMBER function. This function is used to assign a row number to every matching row, and then one searches for the row with the middle row number.

Using Formula #1

Below is some sample code that gets the median SALARY, by JOB, for some set of rows in the STAFF table. Two JOB values are referenced - one with seven matching rows, and one with four. The query logic goes as follows:

- Get the matching set of rows from the STAFF table, and give each row a row-number, within each JOB value.
- Using the set of rows retrieved above, get the maximum row-number, per JOB value, then add 1.0, then divide by 2, then add or subtract 0.6. This will give one two values that encompass a single row-number, if an odd number of rows match, and two row-numbers, if an even number of rows match.
- Finally, join the one row per JOB obtained in step 2 above to the set of rows retrieved in step 1 - by common JOB value, and where the row-number is within the high/low range. The average salary of whatever is retrieved is the median.

Now for the code:

```

WITH numbered_rows AS
  (SELECT   s.*
           ,ROW_NUMBER() OVER(PARTITION BY job
                               ORDER   BY salary, id) AS row#
   FROM     staff s
   WHERE    comm > 0
           AND name LIKE '%e%'),
median_row_num AS
  (SELECT   job
           ,(MAX(row# + 1.0) / 2) - 0.5 AS med_lo
           ,(MAX(row# + 1.0) / 2) + 0.5 AS med_hi
   FROM     numbered_rows
   GROUP BY job)
SELECT   nn.job
        ,DEC(AVG(nn.salary),7,2) AS med_sal
FROM     numbered_rows nn
        ,median_row_num mr
WHERE    nn.job = mr.job
        AND nn.row# BETWEEN mr.med_lo AND mr.med_hi
GROUP BY nn.job
ORDER BY nn.job;

```

	ANSWER
	=====
	JOB MED_SAL

	Clerk 13030.50
	Sales 17432.10

Figure 999, Calculating the median

IMPORTANT: To get consistent results when using the ROW_NUMBER function, one must ensure that the ORDER BY column list encompasses the unique key of the table. Otherwise the row-number values will be assigned randomly - if there are multiple rows with the same value. In this particular case, the ID has been included in the ORDER BY list, to address duplicate SALARY values.

The next example is the essentially the same as the prior, but there is additional code that gets the average SALARY, and a count of the number of matching rows per JOB value. Observe that all this extra code went in the second step:

```

WITH numbered_rows AS
  (SELECT   s.*
           ,ROW_NUMBER() OVER(PARTITION BY job
                               ORDER   BY salary, id) AS row#
   FROM     staff s
   WHERE    comm > 0
           AND name LIKE '%e%'),
median_row_num AS
  (SELECT   job
           ,(MAX(row# + 1.0) / 2) - 0.5 AS med_lo
           ,(MAX(row# + 1.0) / 2) + 0.5 AS med_hi
           ,DEC(AVG(salary),7,2) AS avg_sal
           ,COUNT(*) AS #rows
   FROM     numbered_rows
   GROUP BY job)
SELECT   nn.job
        ,DEC(AVG(nn.salary),7,2) AS med_sal
        ,MAX(mr.avg_sal) AS avg_sal
        ,MAX(mr.#rows) AS #r
FROM     numbered_rows nn
        ,median_row_num mr
WHERE    nn.job = mr.job
        AND nn.row# BETWEEN mr.med_lo
        AND mr.med_hi
GROUP BY nn.job
ORDER BY nn.job;

```

	ANSWER
	=====
	JOB MED_SAL AVG_SAL #R

	Clerk 13030.50 12857.56 7
	Sales 17432.10 17460.93 4

Figure 1000, Get median plus average

Using Formula #2

Once again, the following sample code gets the median SALARY, by JOB, for some set of rows in the STAFF table. Two JOB values are referenced - one with seven matching rows,

and the other with four. In this case, when there are an even number of matching rows, the smaller of the two middle values is chosen. The logic goes as follows:

- Get the matching set of rows from the STAFF table, and give each row a row-number, within each JOB value.
- Using the set of rows retrieved above, get the maximum row-number per JOB, then add 1, then divide by 2. This will get the row-number for the row with the median value.
- Finally, join the one row per JOB obtained in step 2 above to the set of rows retrieved in step 1 - by common JOB and row-number value.

```

WITH numbered_rows AS
  (SELECT   s.*
           ,ROW_NUMBER() OVER(PARTITION BY job
                               ORDER   BY salary, id) AS row#
   FROM     staff s
   WHERE    comm   > 0
           AND    name LIKE '%e%'),
median_row_num AS
  (SELECT   job
           ,MAX(row# + 1) / 2 AS med_row#
   FROM     numbered_rows
   GROUP BY job)
SELECT   nn.job
        ,nn.salary AS med_sal
FROM     numbered_rows nn
        ,median_row_num mr
WHERE    nn.job = mr.job
        AND    nn.row# = mr.med_row#
ORDER BY nn.job;

```

ANSWER	
JOB	MED_SAL
----	-----
Clerk	13030.50
Sales	16858.20

Figure 1001, Calculating the median

The next query is the same as the prior, but it uses a sub-query, instead of creating and then joining to a second temporary table:

```

WITH numbered_rows AS
  (SELECT   s.*
           ,ROW_NUMBER() OVER(PARTITION BY job
                               ORDER   BY salary, id) AS row#
   FROM     staff s
   WHERE    comm   > 0
           AND    name LIKE '%e%')
SELECT   job
        ,salary AS med_sal
FROM     numbered_rows
WHERE    (job,row#) IN
        (SELECT   job
           ,MAX(row# + 1) / 2
   FROM     numbered_rows
   GROUP BY job)
ORDER BY job;

```

ANSWER	
JOB	MED_SAL
----	-----
Clerk	13030.50
Sales	16858.20

Figure 1002, Calculating the median

The next query lists every matching row in the STAFF table (per JOB), and on each line of output, shows the median salary:

```

WITH numbered_rows AS
  (SELECT   s.*
    ,ROW_NUMBER() OVER(PARTITION BY job
                       ORDER   BY salary, id) AS row#
  FROM     staff s
  WHERE    comm > 0
  AND     name LIKE '%e%')
SELECT   r1.*
  ,(SELECT  r2.salary
    FROM    numbered_rows r2
    WHERE   r2.job = r1.job
    AND     r2.row# = (SELECT  MAX(r3.row# + 1) / 2
                          FROM    numbered_rows r3
                          WHERE   r2.job = r3.job)) AS med_sal
FROM     numbered_rows r1
ORDER BY job
        ,salary;

```

Figure 1003, List matching rows and median

Converting HEX Data to Number

The following trigger accepts as input a hexadecimal representation of an integer value, and returns a BIGINT number. It works for any integer type:

```

CREATE FUNCTION hex_to_int(input_val VARCHAR(16))
RETURNS BIGINT
BEGIN ATOMIC
    DECLARE parse_val VARCHAR(16) DEFAULT '';
    DECLARE sign_val BIGINT DEFAULT 1;
    DECLARE out_val BIGINT DEFAULT 0;
    DECLARE cur_exp BIGINT DEFAULT 1;
    DECLARE input_len SMALLINT DEFAULT 0;
    DECLARE cur_byte SMALLINT DEFAULT 1;
    IF LENGTH(input_val) NOT IN (4,8,16) THEN
        SIGNAL SQLSTATE VALUE '70001' SET MESSAGE_TEXT = 'Length wrong';
    END IF;
    SET input_len = LENGTH(input_val);
    WHILE cur_byte <= input_len DO
        SET parse_val = parse_val
            SUBSTR(input_val,cur_byte + 1,1) ||
            SUBSTR(input_val,cur_byte + 0,1);
        SET cur_byte = cur_byte + 2;
    END WHILE;
    IF SUBSTR(parse_val,input_len,1) BETWEEN '8' AND 'F' THEN
        SET sign_val = -1;
        SET out_val = -1;
        SET parse_val = TRANSLATE(parse_val
            , '0123456789ABCDEF'
            , 'FEDCBA9876543210');
    END IF;
    SET cur_byte = 1;
    WHILE cur_byte <= input_len DO
        SET out_val = out_val +
            (cur_exp *
            sign_val *
            CASE SUBSTR(parse_val,cur_byte,1)
                WHEN '0' THEN 00    WHEN '1' THEN 01
                WHEN '2' THEN 02    WHEN '3' THEN 03
                WHEN '4' THEN 04    WHEN '5' THEN 05
                WHEN '6' THEN 06    WHEN '7' THEN 07
                WHEN '8' THEN 08    WHEN '9' THEN 09
                WHEN 'A' THEN 10    WHEN 'B' THEN 11
                WHEN 'C' THEN 12    WHEN 'D' THEN 13
                WHEN 'E' THEN 14    WHEN 'F' THEN 15
            END);
        IF cur_byte < input_len THEN
            SET cur_exp = cur_exp * 16;
        END IF;
        SET cur_byte = cur_byte + 1;
    END WHILE;
    RETURN out_val;
END

```

Figure 1004, Trigger to convert HEX value to integer

Trigger Logic

The trigger does the following:

- Check that the input value is the correct length for an integer value. If not, flag an error.
- Transpose every second byte in the input value. This is done because the HEX representation of an integer does not show the data as it really is.
- Check the high-order bit of what is now the last byte. If it is a "1", the value is a negative number, so the processing will be slightly different.
- Starting with the first byte in the (transposed) input, covert each byte to a integer value using CASE logic. Multiply each digit obtained by the (next) power of sixteen.
- Return the final result.

Usage Examples

WITH temp1 (num) AS			ANSWER
(VALUES (SMALLINT(+0))			=====
, (SMALLINT(+1))		NUM	HEX
, (SMALLINT(-1))			H2I
, (SMALLINT(+32767))		-----	-----
, (SMALLINT(-32768)))			
SELECT num		0	0000
, HEX(num) AS hex		1	0100
, hex_to_int(HEX(num)) AS h2i		-1	FFFF
FROM temp1;		32767	FF7F
		-32768	0080
			-32768

Figure 1005, Using trigger to convert data

WITH			ANSWER
temp1 (num) AS			=====
(VALUES (INTEGER(0))		NUM	HEX
UNION ALL			H2I
SELECT (num + 1) * 7		-----	-----
FROM temp1		-87432800	A0E1C9FA
WHERE num < 1E6),		-12490387	6D6941FF
temp2 (sgn) AS		-1784328	F8C5E4FF
(VALUES (+1)		-254891	551CFCFF
, (-13)),		-36400	D071FFFF
temp3 (num) AS		-5187	BDEBFFFF
(SELECT DISTINCT		-728	28FDFFFF
num * sgn		-91	A5FFFFFF
FROM temp1		0	00000000
, temp2)		7	07000000
SELECT num		56	38000000
, HEX(num) AS hex		399	8F010000
, hex_to_int(HEX(num)) AS h2i		2800	F00A0000
FROM temp3		19607	974C0000
ORDER BY num;		137256	28180200
		960799	1FA90E00
		6725600	E09F6600
			6725600

*Figure 1006, Using trigger to convert data***Usage Notes**

- The above function won't work on the mainframe because the internal representation of an integer value is different (see below). The modifications required to make it work are minor.
- The above function won't work on the HEX representation of packed-decimal or floating-point data.
- One could have three different flavors of the above function - one for each type of integer value. The input value length would determine the output type.

Endianness

Most computers use one of two internal formats to store binary data. In big-endian, which is used on z/OS, the internal representation equals the HEX value. So the four-byte integer value 1,234,567,890 is stored as "49 96 02 D2". In little-endian, which is used on all Intel chips, the bytes are reversed, so the above value is stored internally as "D2 02 96 49". This is why the above trigger transposed every two-byte block before converting the HEX value to numeric.

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
  ('A'
   ,TIMESTAMP('1996-05-01-24.00.00.000000')
   ,TIMESTAMP('1996-05-02-00.00.00.000000') ))
SELECT c1
FROM   temp1
WHERE  t1 = t2;
```

ANSWER
=====
<no rows>

Figure 1007, 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
  ('A'
   ,TIMESTAMP('1996-05-01-24.00.00.000000')
   ,TIMESTAMP('1996-05-02-00.00.00.000000') ))
SELECT c1
FROM   temp1
WHERE  t1 + 0 MICROSECOND = t2 + 0 MICROSECOND;
```

ANSWER
=====
C1
--
A

Figure 1008, Timestamp comparison - Correct

Be aware that, as with everything else in this section, what is shown above is not a bug. It is the way that it is because it makes perfect sense, even if it is not intuitive.

Using 24 Hour Notation

One might have to use the 24-hour notation, if one needs to record (in DB2) external actions that happen just before midnight - with the correct date value. To illustrate, imagine that we have the following table, which records supermarket sales:

```
CREATE TABLE supermarket_sales
(sales_ts  TIMESTAMP      NOT NULL
 ,sales_val DECIMAL(8,2)  NOT NULL
 ,PRIMARY KEY(sales_ts));
```

Figure 1009, Sample Table

In this application, anything that happens before midnight, no matter how close, is deemed to have happened on the specified day. So if a transaction comes in with a timestamp value that is a tiny fraction of a microsecond before midnight, we should record it thus:

```
INSERT INTO supermarket_sales VALUES
('2003-08-01-24.00.00.000000',123.45);
```

Figure 1010, Insert row

Now, if we want to select all of the rows that are for a given day, we can write this:

```
SELECT *
FROM   supermarket_sales
WHERE  DATE(sales_ts) = '2003-08-01'
ORDER BY sales_ts;
```

Figure 1011, Select rows for given date

Or this:

```
SELECT *
FROM   supermarket_sales
WHERE  sales_ts BETWEEN '2003-08-01-00.00.00'
      AND '2003-08-01-24.00.00'
ORDER BY sales_ts;
```

Figure 1012, Select rows for given date

DB2 will never internally generate a timestamp value that uses the 24 hour notation. But it is provided so that you can use it yourself, if you need to.

No Rows Match

How many rows 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
FROM   sysibm.systables
WHERE  creator = 'ZZZ';
```

ANSWER
=====
<no row>

Figure 1013, Query with no matching rows (1 of 8)

```
SELECT MAX(creator)
FROM   sysibm.systables
WHERE  creator = 'ZZZ';
```

ANSWER
=====
<null>

Figure 1014, Query with no matching rows (2 of 8)

```
SELECT MAX(creator)
FROM   sysibm.systables
WHERE  creator = 'ZZZ'
HAVING MAX(creator) IS NOT NULL;
```

ANSWER
=====
<no row>

Figure 1015, Query with no matching rows (3 of 8)

```
SELECT MAX(creator)
FROM   sysibm.systables
WHERE  creator = 'ZZZ'
HAVING MAX(creator) = 'ZZZ';
```

ANSWER
=====
<no row>

Figure 1016, Query with no matching rows (4 of 8)

```
SELECT MAX(creator)
FROM   sysibm.systables
WHERE  creator = 'ZZZ'
GROUP BY creator;
```

ANSWER
=====
<no row>

Figure 1017, Query with no matching rows (5 of 8)

```
SELECT creator
FROM   sysibm.systables
WHERE  creator = 'ZZZ'
GROUP BY creator;
```

ANSWER
=====
<no row>

Figure 1018, Query with no matching rows (6 of 8)

```
SELECT COUNT(*)
FROM   sysibm.systables
WHERE  creator = 'ZZZ'
GROUP BY creator;
```

ANSWER
=====
<no row>

Figure 1019, Query with no matching rows (7 of 8)

```

SELECT    COUNT(*)                                ANSWER
FROM      sysibm.systables                        =====
WHERE     creator = 'ZZZ';                        0

```

Figure 1020, Query with no matching rows (8 of 8)

There is a pattern to the above, and it goes thus:

- When there is no column function (e.g. MAX, COUNT) in the SELECT then, if there are no matching rows, no row is returned.
- If there is a column function in the SELECT, but nothing else, then the query will always return a row - with zero if the function is a COUNT, and null if it is something else.
- If there is a column function in the SELECT, and also a HAVING phrase in the query, a row will only be returned if the HAVING predicate is true.
- If there is a column function in the SELECT, and also a GROUP BY phrase in the query, a row will only be returned if there was one that matched.

Imagine that one wants to retrieve a list of names from the STAFF table, but when no names match, one wants to get a row/column with the phrase "NO NAMES", rather than zero rows. The next query does this by first generating a "not found" row using the SYSDUMMY1 table, and then left-outer-joining to the set of matching rows in the STAFF table. The COALESCE function will return the STAFF data, if there is any, else the not-found data:

```

SELECT    COALESCE(name,noname) AS nme           ANSWER
          ,COALESCE(salary,nosal) AS sal       =====
FROM      (SELECT 'NO NAME' AS noname          NME      SAL
          ,0 AS nosal
          FROM    sysibm.sysdummy1
          )AS nnn
LEFT OUTER JOIN
  (SELECT *
   FROM    staff
   WHERE   id < 5
   )AS xxx
ON        1 = 1
ORDER BY name;
          NO NAME 0.00

```

Figure 1021, Always get a row, example 1 of 2

The next query is logically the same as the prior, but it uses the WITH phrase to generate the "not found" row in the SQL statement:

```

WITH nnn (noname, nosal) AS                    ANSWER
  (VALUES ('NO NAME',0))                      =====
SELECT    COALESCE(name,noname) AS nme       NME      SAL
          ,COALESCE(salary,nosal) AS sal     -----
FROM      nnn                                NO NAME 0.00
LEFT OUTER JOIN
  (SELECT *
   FROM    staff
   WHERE   id < 5
   )AS xxx
ON        1 = 1
ORDER BY NAME;

```

Figure 1022, Always get a row, example 2 of 2

Dumb Date Usage

Imagine that you have some character value that you convert to a DB2 date. The correct way to do it is given below:

```

SELECT    DATE( '2001-09-22' )
FROM      sysibm.sysdummy1;

```

ANSWER
=====

2001-09-22

Figure 1023, 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)
FROM      sysibm.sysdummy1;

```

ANSWER
=====

0006-05-24

Figure 1024, 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
          ,name
FROM      staff
WHERE     id <= 100
        AND id = (INT(RAND()* 10) * 10) + 10
ORDER BY id;

```

ANSWER
=====

ID NAME
-- -----
30 Marenghi
60 Quigley

Figure 1025, 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
(
SELECT    id
          ,name
          ,(INT(RAND(0)* 10) * 10) + 10 AS ran
FROM      staff
WHERE     id <= 100
)
SELECT    t.*
          ,CASE id
            WHEN ran THEN 'Y'
            ELSE          ''
          END AS eql
FROM      temp t
ORDER BY id;

```

ANSWER
=====

ID	NAME	RAN	EQL
10	Sanders	10	Y
20	Pernal	30	
30	Marenghi	70	
40	O'Brien	10	
50	Hanes	30	
60	Quigley	40	
70	Rothman	30	
80	James	100	
90	Koonitz	40	
100	Plotz	100	Y

Figure 1026, Get random rows - Explanation

NOTE: To randomly select some fraction of the rows in a table efficiently and consistently, use the TABLESAMPLE feature. See page 372 for more details.

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
staff_numbered AS
  (SELECT  s.*
         ,ROW_NUMBER() OVER() AS row#
    FROM    staff s
   WHERE   id <= 100
  ),
count_rows AS
  (SELECT  MAX(row#) AS #rows
    FROM    staff_numbered
  ),
random_values (RAN#) AS
  (VALUES (RAND())
         ,(RAND())
         ,(RAND())
  ),
rows_t0_get AS
  (SELECT  INT(ran# * #rows) + 1 AS get_row
    FROM    random_values
         ,count_rows
  )
SELECT    id
         ,name
  FROM      staff_numbered
         ,rows_t0_get
  WHERE     row# = get_row
  ORDER BY id;

```

```

ANSWER
=====
ID  NAME
---  -----
10  Sanders
20  Pernal
90  Koonitz

```

Figure 1027, 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
        ,name
FROM    (SELECT  s2.*
        ,ROW_NUMBER() OVER(ORDER BY r1) AS r2
        FROM    (SELECT  s1.*
        ,RAND() AS r1
        FROM      staff s1
        WHERE     id <= 100
        )AS s2
        )as s3
WHERE   r2 <= 3
ORDER BY id;

```

```

ANSWER
=====
ID NAME
-- -----
10 Sanders
40 O'Brien
60 Quigley

```

Figure 1028, 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 1029, Date/Time manipulation - wrong

As you can see, my end-time is incorrect. In particular, the fractional 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 1030, 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, one has to convert the value into some whole number of a more precise time period. For example:

- 5.1 days times 86,400 returns the equivalent number of seconds.
- 6.2 seconds times 1,000,000 returns the equivalent number of microseconds.

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
  ('A',CHAR(' ',1),VARCHAR(' ',1)),
  ('B',CHAR(' ',1),VARCHAR(' ',1)))
SELECT c0
FROM   temp1
WHERE  c1 = v1
      AND c1 LIKE ' ';

```

```

ANSWER
=====
C0
--
A
B

```

Figure 1031, 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
  ('A',CHAR(' ',1),VARCHAR(' ',1)),
  ('B',CHAR(' ',1),VARCHAR(' ',1)))
SELECT c0
FROM   temp1
WHERE  c1 = v1
      AND v1 LIKE ' ';

```

```

ANSWER
=====
C0
--
A

```

Figure 1032, 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
    ('A',CHAR(' ',1),VARCHAR(' ',1)),
    ('B',CHAR(' ',1),VARCHAR(' ',1)))
SELECT c0
FROM temp1
WHERE c1 = v1
AND RTRIM(v1) LIKE ' ';

```

	ANSWER
	=====
	C0
	--
	A
	B

Figure 1033, 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
(VVALUES DATE('2000-01-01')
UNION ALL
SELECT yymmdd + 1 DAY
FROM temp1
WHERE yymmdd < '2010-12-31'
)
SELECT yy AS year
,CHAR(MIN(yymmdd),ISO) AS min_dt
,CHAR(MAX(yymmdd),ISO) AS max_dt
FROM (SELECT yymmdd
,YEAR(yymmdd) yy
, WEEK(yymmdd) wk
FROM temp1
WHERE WEEK(yymmdd) = 33
)AS xxx
GROUP BY yy
,wk;

```

	ANSWER
	=====
	YEAR MIN_DT MAX_DT

	2000 2000-08-06 2000-08-12
	2001 2001-08-12 2001-08-18
	2002 2002-08-11 2002-08-17
	2003 2003-08-10 2003-08-16
	2004 2004-08-08 2004-08-14
	2005 2005-08-07 2005-08-13
	2006 2006-08-13 2006-08-19
	2007 2007-08-12 2007-08-18
	2008 2008-08-10 2008-08-16
	2009 2009-08-09 2009-08-15
	2010 2010-08-08 2010-08-14

Figure 1034, 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
,INTEGER(SUM(salary)) AS s2
FROM staff;

```

	ANSWER
	=====
	S1 S2

	583633 583647

Figure 1035, 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 1036, 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 1037, 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 1038, 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 1039, Division and Average

Arguably, either answer could be correct - depending upon what the user wants. In practice, the first answer is almost always what they intended. The second answer is somewhat flawed because it gives no weighting to the absolute size of the values in each row (i.e. a big SALARY divided by a big COMM is the same as a small divided by a small).

Date Output Order

DB2 has a bind option (called DATETIME) that specifies the default output format of date-time data. This bind option has no impact on the sequence with which date-time data is presented. It simply defines the output template used. To illustrate, the plan that was used to run the following SQL defaults to the USA date-time-format bind option. Observe that the month is the first field printed, but the rows are sequenced by year:

```

SELECT  hiredate
FROM    employee
WHERE   hiredate < '1960-01-01'
ORDER BY 1;

```

ANSWER
=====
1947-05-05
1949-08-17
1958-05-16

Figure 1040, 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)
FROM    employee
WHERE   hiredate < '1960-01-01'
ORDER BY 1;

```

ANSWER
=====
05/05/1947
05/16/1958
08/17/1949

Figure 1041, 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 1042, 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.

Multiple User Interactions

There was once a mythical company that wrote a query to list all orders in the ORDER table for a particular DATE, with the output sequenced by REGION and STATUS. To make the query fly, there was a secondary index on the DATE, REGION, and STATUS columns, in addition to the primary unique index on the ORDER-NUMBER column:

```
SELECT    region_code  AS region
         ,order_status AS status
         ,order_number AS order#
         ,order_value  AS value
FROM      order_table
WHERE     order_date   =  '2006-03-12'
ORDER BY  region_code
         ,order_status
WITH CS;
```

Figure 1043, Select from ORDER table

When the users ran the above query, they found that some orders were seemingly listed twice:

REGION	STATUS	ORDER#	VALUE	
-----	-----	-----	-----	
EAST	PAID	111	4.66	<----- Same ORDER#
EAST	PAID	222	6.33	
EAST	PAID	333	123.45	
EAST	SHIPPED	111	4.66	<-----
EAST	SHIPPED	444	123.45	

Figure 1044, Sample query output

While the above query was running (i.e. traversing the secondary index) another user had come along and updated the STATUS for ORDER# 111 from PAID to SHIPPED, and then committed the change. This update moved the pointer for the row down the secondary index, so that the query subsequently fetched the same row twice.

Explanation

In the above query, DB2 is working exactly as intended. Because the result may seem a little odd to some people, a simple example will be used to explain what is going on:

Imagine that one wants to count the number of cars parked on a busy street by walking down the road from one end to the other, counting each parked car as you walk past. By the time you get to the end of the street, you will have a number, but that number will not represent the number of cars parked on the street at any point in time. And if a car that you counted at the start of the street was moved to the end of the street while you were walking, you will have counted that particular car twice. Likewise, a car that was moved from the end of the street to the start of the street while you were walking in the middle of the street would not have been counted by you, even though it never left the street during your walk.

One way to get a true count of cars on the street is to prevent car movement while you do your walk. This can be unpopular, but it works. The same can be done in DB2 by changing the WITH phrase (i.e. isolation level) at the bottom of the above query:

WITH RR - Repeatable Read

A query defined with repeatable read can be run multiple times and will always return the same result, with the following qualifications:

- References to special registers, like CURRENT_TIMESTAMP, may differ.
- Rows changed by the user will show in the query results.

No row will ever be seen twice with this solution, because once a row is read it cannot be changed. And the query result is a valid representation of the state of the table, or at least of the matching rows, as of when the query finished.

In the car-counting analogy described above, this solution is akin to locking down sections of the street as you walk past, regardless of whether there is a car parked there or not. As long as you do not move a car yourself, each traverse of the street will always get the same count, and no car will ever be counted more than once.

In many cases, defining a query with repeatable read will block all changes by other users to the target table for the duration. In theory, rows can be changed if they are outside the range of the query predicates, but this is not always true. In the case of the order system described above, it was not possible to use this solution because orders were coming in all the time.

WITH RS - Read Stability

A query defined with read-stability can be run multiple times, and each row processed previously will always look the same the next time that the query is run - with the qualifications listed above. But rows can be inserted into the table that match the query predicates. These will show in the next run. No row will ever be inadvertently read twice.

In our car-counting analogy, this solution is akin to putting a wheel-lock on each parked car as you walk past. The car can't move, but new cars can be parked in the street while you are counting. The new cars can also leave subsequently, as long as you don't lock them in your next walk down the street. No car will ever be counted more than once in a single pass, but nor will your count ever represent the true state of the street.

As with repeatable read, defining a query with read stability will often block all updates by other users to the target table for the duration. It is not a great way to win friends.

WITH CS - Cursor Stability

A query defined with cursor stability will read every committed matching row, occasionally more than once. If the query is run multiple times, it may get a different result each time.

In our car-counting analogy, this solution is akin to putting a wheel-lock on each parked car as you count it, but then removing the lock as soon as you move on to the next car. A car that you are not currently counting can be moved anywhere in the street, including to where you have yet to count. In the latter case, you will count it again. This is what happened during our mythical query of the ORDER table.

Queries defined with cursor stability still need to take locks, and thus can be delayed if another user has updated a matching row, but not yet done a commit. In extreme cases, the query may get a timeout or deadlock.

WITH UR - Uncommitted Read

A query defined with uncommitted read will read every matching row, including those that have not yet been committed. Rows may occasionally be read more than once. If the query is run multiple times, it may get a different result each time.

In our car-counting analogy, this solution is akin to counting each stationary car as one walks past, regardless of whether or not the car is permanently parked.

Queries defined with uncommitted read do not take locks, and thus are not delayed by other users who have changed rows, but not yet committed. But some of the rows read may be subsequently rolled back, and so were never valid.

Below is a summary of the above options:

CURSOR "WITH" OPTION	SAME RESULT IF RUN TWICE	FETCH SAME ROW > ONCE	UNCOMMITTED ROWS SEEN	ROWS LOCKED
RR - Repeatable Read	Yes	Never	Never	Many/All
RS - Read Stability	No (inserts)	Never	Never	Many/All
CS - Cursor Stability	No (all DML)	Maybe	Never	Current
UR - Uncommitted Read	No (all DML)	Maybe	Yes	None

Figure 1045, WITH Option vs. Actions

Other Solutions - Good and Bad

Below are some alternatives to the above:

- **Lock Table:** If one wanted to see the state of the table as it was at the start of the query, one could use a LOCK TABLE command - in share or exclusive mode. Doing this may not win you many friends with other users.
- **Drop Secondary Indexes:** The problem described above does not occur if one accesses the table using a tablespace scan, or via the primary index. However, if the table is large, secondary indexes will probably be needed to get the job done.
- **Check for Changes:** The target table can have a column of type timestamp that is set to the current timestamp value every time a row is changed (this rule can be enforced using triggers). The query scanning the table can have a predicate (see below) so it only fetches those rows that were updated before the current timestamp, which is the time when the query was opened:

```

SELECT   region_code AS region
        ,order_status AS status
        ,order_number AS order#
        ,order_value  AS value
FROM     order_table
WHERE    order_date = '2006-03-12'
        AND update_ts < CURRENT_TIMESTAMP    <= New predicate
ORDER BY region_code
        ,order_status
WITH CS;

```

Figure 1046, Select from ORDER table

This solution is almost certainly going to do the job, but it is not quite perfect. There is a very small chance that one can still fetch the same row twice. To illustrate, imagine the following admittedly very improbable sequence of events:

```

#1 UPDATE statement begins (will run for a long time).
#2 QUERY begins (will also run for a long time).
#3 QUERY fetches target row (via secondary index).
#4 QUERY moves on to the next row, etc...
#5 UPDATE changes target row - moves it down index.
#6 UPDATE statement finishes, and commits.
#7 QUERY fetches target row again (bother).

```

Figure 1047, Sequence of events required to fetch same row twice

- **Two-part Query:** One can do the query in two parts: First get a list of DISTINCT primary key values, then join back to the original table using the primary unique index to get the rest of the row:

```

SELECT   region_code   AS region
        ,order_status  AS status
        ,order_number  AS order#
        ,order_value   AS value
FROM     (SELECT   DISTINCT
          order_number AS distinct_order#
        FROM     order_table
        WHERE    order_date = '2006-03-12'
        )AS xxx
        ,order_table
WHERE    order_number = distinct_order#
ORDER BY region_code
        ,order_status
WITH CS;

```

Figure 1048, Two-part query

This solution will do the job, but it is probably going to take about twice as long to complete as the original query.

- **Use Versions:** See the chapter titled "Retaining a Record" for a schema that uses lots of complex triggers and views, and that lets one see consistent views of the rows in the table as of any point in time.

Floating Point Numbers

The following SQL repetitively multiplies a floating-point number by ten:

```

WITH temp (f1) AS
(VVALUES 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 1049, 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 1050, 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
(VVALUES (FLOAT(1.23456789E1 * 10 * 10 * 10 * 10 * 10 * 10 * 10)
,FLOAT(1.23456789E8)))
SELECT f1
      ,f2
FROM   temp
WHERE  f1 <> f2;
```

ANSWER	
F1	F2
+1.23456789000000E+008	+1.23456789000000E+008

Figure 1051, 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
(VVALUES (FLOAT(1.23456789E1 * 10 * 10 * 10 * 10 * 10 * 10 * 10)
,FLOAT(1.23456789E8)))
SELECT HEX(f1) AS hex_f1
      ,HEX(f2) AS hex_f2
FROM   temp
WHERE  f1 <> f2;
```

ANSWER	
HEX_F1	HEX_F2
FFFFFF53346F9D41	00000054346F9D41

Figure 1052, 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 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
      ,dbl1 / 3 AS dbl2
    FROM   temp1)
SELECT *
FROM   temp2
WHERE  dbl2 <> dec2;
```

ANSWER (1 row returned)
=====

DEC1 =	1.0
DEC2 =	0.33333333333333333333
DBL1 =	+1.0000000000000000E+000
DBL2 =	+3.3333333333333333E-001

Figure 1053, 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 1054, Comparing float and decimal multiplication, SQL

Here is the answer:

F1	D1	COMPARE
+1.2345678900000000E+000	1.2345678900	SAME
+1.2345678900000000E+001	12.3456789000	SAME
+1.2345678900000000E+002	123.4567890000	DIFF
+1.2345678900000000E+003	1234.5678900000	DIFF
+1.2345678900000000E+004	12345.6789000000	DIFF
+1.2345678900000000E+005	123456.7890000000	DIFF
+1.2345678900000000E+006	1234567.8900000000	SAME
+1.2345678900000000E+007	12345678.9000000000	DIFF
+1.2345678900000000E+008	123456789.0000000000	DIFF
+1.2345678900000000E+009	1234567890.0000000000	DIFF

Figure 1055, 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
  (VALUES FLOAT(0.1))
SELECT f1
       ,HEX(f1) AS hex_f1
FROM   temp;

```

	ANSWER	
	=====	
	F1	HEX_F1
	-----	-----
	+1.0000000000000000E-001	9A9999999999B93F

Figure 1056, 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 1057, Sample view definition

Now imagine that we run the following query against this view:

```

SELECT db2/v5 AS answer
FROM   damn_lawyers;

```

	ANSWER

	0
	617

Figure 1058, 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
FROM   damn_lawyers;

```

	ANSWER

	0
	617

Figure 1059, Trademark Valid SQL

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

Appendix

DB2 Sample Tables

Sample table DDL follows. A text file containing the same can be found on my website.

ACT

```
CREATE TABLE ACT
( ACTNO          SMALLINT          NOT NULL
, ACTKWD         CHARACTER(6)      NOT NULL
, ACTDESC       VARCHAR(20)       NOT NULL)
IN USERSPACE1;
```

```
ALTER TABLE ACT
ADD CONSTRAINT PK_ACT PRIMARY KEY
( ACTNO );
```

```
CREATE UNIQUE INDEX XACT2 ON ACT
( ACTNO          ASC
, ACTKWD         ASC)
ALLOW REVERSE SCANS;
```

Figure 1060, ACT sample table – DDL

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

Figure 1061, ACT sample table – DATA

CATALOG

```
CREATE TABLE CATALOG
( NAME          VARCHAR(128)      NOT NULL
, CATLOG       XML)
IN IBMDB2SAMPLEXML;
```

```
ALTER TABLE CATALOG
ADD CONSTRAINT PK_CATALOG PRIMARY KEY
( NAME );
```

Figure 1062, CATALOG sample table – DDL

There is no data in this table.

CL_SCHED

```
CREATE TABLE CL_SCHED
(CLASS_CODE CHARACTER(7)
, DAY SMALLINT
, STARTING TIME
, ENDING TIME)
IN USERSPACE1;
```

Figure 1063, CL_SCHED sample table – DDL

CLASS_CODE	DAY	STARTING	ENDING
042:BF	4	12:10:00	14:00:00
553:MJA	1	10:30:00	11:00:00
543:CWM	3	09:10:00	10:30:00
778:RES	2	12:10:00	14:00:00
044:HD	3	17:12:30	18:00:00

*Figure 1064, CL_SCHED sample table – DATA***CUSTOMER**

```
CREATE TABLE CUSTOMER
(CID BIGINT NOT NULL
, INFO XML
, HISTORY XML)
IN IBMDB2SAMPLEXML;
```

```
ALTER TABLE CUSTOMER
ADD CONSTRAINT PK_CUSTOMER PRIMARY KEY
(CID);
```

Figure 1065, CUSTOMER sample table – DDL

CID	INFO	HISTORY
1000	<<xml>>	<<xml>>
1001	<<xml>>	<<xml>>
1002	<<xml>>	<<xml>>
1003	<<xml>>	<<xml>>
1004	<<xml>>	<<xml>>
1005	<<xml>>	<<xml>>

*Figure 1066, CUSTOMER sample table – DATA***DEPARTMENT**

```
CREATE TABLE DEPARTMENT
(DEPTNO CHARACTER(3) NOT NULL
, DEPTNAME VARCHAR(36) NOT NULL
, MGRNO CHARACTER(6)
, ADMRDEPT CHARACTER(3) NOT NULL
, LOCATION CHARACTER(16))
IN USERSPACE1;
```

```
ALTER TABLE DEPARTMENT
ADD CONSTRAINT PK_DEPARTMENT PRIMARY KEY
(DEPTNO);
```

```
CREATE INDEX XDEPT2 ON DEPARTMENT
(MGRNO ASC)
ALLOW REVERSE SCANS;
```

```
CREATE INDEX XDEPT3 ON DEPARTMENT
(ADMRDEPT ASC)
ALLOW REVERSE SCANS;
```

```
CREATE ALIAS DEPT FOR DEPARTMENT;
```

Figure 1067, 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	-
F22	BRANCH OFFICE F2	-	E01	-
G22	BRANCH OFFICE G2	-	E01	-
H22	BRANCH OFFICE H2	-	E01	-
I22	BRANCH OFFICE I2	-	E01	-
J22	BRANCH OFFICE J2	-	E01	-

Figure 1068, DEPARTMENT sample table – Data

EMP_PHOTO

```
CREATE TABLE EMP_PHOTO
(EMPNO          CHARACTER(6)          NOT NULL
,PHOTO_FORMAT   VARCHAR(10)           NOT NULL
,PICTURE        BLOB(102400)
,EMP_ROWID      CHARACTER(40)       NOT NULL)
IN USERSPACE1;

ALTER TABLE EMP_PHOTO
ADD CONSTRAINT PK_EMP_PHOTO PRIMARY KEY
(EMPNO
,PHOTO_FORMAT);
```

Figure 1069, EMP_PHOTO sample table – DDL

EMPNO	PHOTO_FORMAT	PICTURE
000130	bitmap	<<photo>>
000130	gif	<<photo>>
000140	bitmap	<<photo>>
000140	gif	<<photo>>
000150	bitmap	<<photo>>
000150	gif	<<photo>>
000190	bitmap	<<photo>>
000190	gif	<<photo>>

Figure 1070, EMP_PHOTO sample table – DATA

EMP_RESUME

```
CREATE TABLE EMP_RESUME
(EMPNO          CHARACTER(6)          NOT NULL
,RESUME_FORMAT  VARCHAR(10)           NOT NULL
,RESUME         CLOB(5120)
,EMP_ROWID      CHARACTER(40)       NOT NULL)
IN USERSPACE1;

ALTER TABLE EMP_RESUME
ADD CONSTRAINT PK_EMP_RESUME PRIMARY KEY
(EMPNO
,RESUME_FORMAT);
```

Figure 1071, EMP_RESUME sample table – DDL

```

EMPNO  RESUME_FORMAT  RESUME
-----
000130  ascii             <<clob>>
000130  html             <<clob>>
000140  ascii             <<clob>>
000140  html             <<clob>>
000150  ascii             <<clob>>
000150  html             <<clob>>
000190  ascii             <<clob>>
000190  html             <<clob>>

```

Figure 1072, EMP_RESUME sample table – DATA

EMPLOYEE

```

CREATE TABLE EMPLOYEE
(EMPNO          CHARACTER(6)          NOT NULL
, FIRSTNME     VARCHAR(12)          NOT NULL
, MIDINIT      CHARACTER(1)
, LASTNAME     VARCHAR(15)          NOT NULL
, WORKDEPT     CHARACTER(3)
, PHONENO     CHARACTER(4)
, HIREDATE     DATE
, JOB          CHARACTER(8)
, EDLEVEL     SMALLINT             NOT NULL
, SEX         CHARACTER(1)
, BIRTHDATE   DATE
, SALARY      DECIMAL(9,2)
, BONUS       DECIMAL(9,2)
, COMM        DECIMAL(9,2)
IN USERSPACE1;

ALTER TABLE EMPLOYEE
ADD CONSTRAINT PK_EMPLOYEE PRIMARY KEY
(EMPNO);

CREATE INDEX KEMP2 ON EMPLOYEE
(WORKDEPT      ASC)
ALLOW REVERSE SCANS;

CREATE ALIAS EMP FOR EMPLOYEE;

```

Figure 1073, EMPLOYEE sample table – DDL

Some of the columns are excluded below – due to lack of space:

```

EMPNO  FIRSTNME  M  LASTNAME  WDP  PHON  HIREDATE  S  BIRTHDATE  SALRY
-----
000010  CHRISTINE  I  HAAS      A00  3978  01/01/1995  F  08/24/1963  52750
000020  MICHAEL   L  THOMPSON  B01  3476  10/10/2003  M  02/02/1978  94250
000030  SALLY     A  KWAN      C01  4738  04/05/2005  F  05/11/1971  98250
000050  JOHN      B  GEYER     E01  6789  08/17/1979  M  09/15/1955  80175
000060  IRVING    F  STERN     D11  6423  09/14/2003  M  07/07/1975  72250
000070  EVA       D  PULASKI  D21  7831  09/30/2005  F  05/26/2003  96170
000090  EILEEN    W  HENDERSON E11  5498  08/15/2000  F  05/15/1971  89750
000100  THEODORE  Q  SPENSER   E21  0972  06/19/2000  M  12/18/1980  86150
000110  VINCENZO  G  LUCCHESI  A00  3490  05/16/1988  M  11/05/1959  66500
000120  SEAN      O'CONNELL A00  2167  12/05/1993  M  10/18/1972  49250
000130  DELORES  M  QUINTANA  C01  4578  07/28/2001  F  09/15/1955  73800
000140  HEATHER  A  NICHOLLS  C01  1793  12/15/2006  F  01/19/1976  68420
000150  BRUCE     ADAMSON   D11  4510  02/12/2002  M  05/17/1977  55280
000160  ELIZABETH R  PIANKA    D11  3782  10/11/2006  F  04/12/1980  62250
000170  MASATOSHI J  YOSHIMURA D11  2890  09/15/1999  M  01/05/1981  44680
000180  MARILYN  S  SCOUTTEN  D11  1682  07/07/2003  F  02/21/1979  51340
000190  JAMES     H  WALKER    D11  2986  07/26/2004  M  06/25/1982  50450
000200  DAVID     BROWN     D11  4501  03/03/2002  M  05/29/1971  57740
000210  WILLIAM  T  JONES     D11  0942  04/11/1998  M  02/23/2003  68270
000220  JENNIFER  K  LUTZ      D11  0672  08/29/1998  F  03/19/1978  49840

```

Figure 1074, EMPLOYEE sample table – DATA (part 1 of 2)

EMPNO	FIRSTNME	M	LASTNAME	WDP	PHON	HIREDATE	S	BIRTHDATE	SALRY
000230	JAMES	J	JEFFERSON	D21	2094	11/21/1996	M	05/30/1980	42180
000240	SALVATORE	M	MARINO	D21	3780	12/05/2004	M	03/31/2002	48760
000250	DANIEL	S	SMITH	D21	0961	10/30/1999	M	11/12/1969	49180
000260	SYBIL	P	JOHNSON	D21	8953	09/11/2005	F	10/05/1976	47250
000270	MARIA	L	PEREZ	D21	9001	09/30/2006	F	05/26/2003	37380
000280	ETHEL	R	SCHNEIDER	E11	8997	03/24/1997	F	03/28/1976	36250
000290	JOHN	R	PARKER	E11	4502	05/30/2006	M	07/09/1985	35340
000300	PHILIP	X	SMITH	E11	2095	06/19/2002	M	10/27/1976	37750
000310	MAUDE	F	SETRIGHT	E11	3332	09/12/1994	F	04/21/1961	35900
000320	RAMLAL	V	MEHTA	E21	9990	07/07/1995	M	08/11/1962	39950
000330	WING		LEE	E21	2103	02/23/2006	M	07/18/1971	45370
000340	JASON	R	GOUNOT	E21	5698	05/05/1977	M	05/17/1956	43840
200010	DIAN	J	HEMMINGER	A00	3978	01/01/1995	F	08/14/1973	46500
200120	GREG		ORLANDO	A00	2167	05/05/2002	M	10/18/1972	39250
200140	KIM	N	NATZ	C01	1793	12/15/2006	F	01/19/1976	68420
200170	KIYOSHI		YAMAMOTO	D11	2890	09/15/2005	M	01/05/1981	64680
200220	REBA	K	JOHN	D11	0672	08/29/2005	F	03/19/1978	69840
200240	ROBERT	M	MONTEVERDE	D21	3780	12/05/2004	M	03/31/1984	37760
200280	EILEEN	R	SCHWARTZ	E11	8997	03/24/1997	F	03/28/1966	46250
200310	MICHELLE	F	SPRINGER	E11	3332	09/12/1994	F	04/21/1961	35900
200330	HELENA		WONG	E21	2103	02/23/2006	F	07/18/1971	35370
200340	ROY	R	ALONZO	E21	5698	07/05/1997	M	05/17/1956	31840

Figure 1075, EMPLOYEE sample table – DATA (part 2 of 2)

EMPMDC

```
CREATE TABLE EMPMDC
(EMPNO          INTEGER
,DEPT          INTEGER
,DIV           INTEGER)
IN IBMDB2SAMPLEREL;
```

Figure 1076, EMPMDC sample table – DDL

This table has 10,000 rows. The first twenty are shown below:

EMPNO	DEPT	DIV
0	1	1
10	1	1
20	1	1
30	1	1
40	1	1
50	1	1
60	1	1
70	1	1
80	1	1
90	1	1
100	1	1
110	1	1
120	1	1
130	1	1
140	1	1
150	1	1
160	1	1
170	1	1
180	1	1
190	1	1

Figure 1077, EMPMDC sample table – DATA

EMPPROJACT

```

CREATE TABLE EMPPROJACT
(EMPNO          CHARACTER(6)          NOT NULL
,PROJNO        CHARACTER(6)          NOT NULL
,ACTNO         SMALLINT              NOT NULL
,EMPTIME       DECIMAL(5,2)
,EMSTDATE      DATE
,EMENDATE      DATE)
IN USERSPACE1;

```

```
CREATE ALIAS EMP_ACT FOR EMPPROJACT;
```

```
CREATE ALIAS EMPACT FOR EMPPROJACT;
```

Figure 1078, EMPPROJACT sample table – DDL

EMPNO	PROJNO	ACTNO	EMPTIME	EMSTDATE	EMENDATE
000010	AD3100	10	0.50	01/01/2002	07/01/2002
000070	AD3110	10	1.00	01/01/2002	02/01/2003
000230	AD3111	60	1.00	01/01/2002	03/15/2002
000230	AD3111	60	0.50	03/15/2002	04/15/2002
000230	AD3111	70	0.50	03/15/2002	10/15/2002
000230	AD3111	80	0.50	04/15/2002	10/15/2002
000230	AD3111	180	0.50	10/15/2002	01/01/2003
000240	AD3111	70	1.00	02/15/2002	09/15/2002
000240	AD3111	80	1.00	09/15/2002	01/01/2003
000250	AD3112	60	1.00	01/01/2002	02/01/2002
000250	AD3112	60	0.50	02/01/2002	03/15/2002
000250	AD3112	60	1.00	01/01/2003	02/01/2003
000250	AD3112	70	0.50	02/01/2002	03/15/2002
000250	AD3112	70	1.00	03/15/2002	08/15/2002
000250	AD3112	70	0.25	08/15/2002	10/15/2002
000250	AD3112	80	0.25	08/15/2002	10/15/2002
000250	AD3112	80	0.50	10/15/2002	12/01/2002
000250	AD3112	180	0.50	08/15/2002	01/01/2003
000260	AD3113	70	0.50	06/15/2002	07/01/2002
000260	AD3113	70	1.00	07/01/2002	02/01/2003
000260	AD3113	80	1.00	01/01/2002	03/01/2002
000260	AD3113	80	0.50	03/01/2002	04/15/2002
000260	AD3113	180	0.50	03/01/2002	04/15/2002
000260	AD3113	180	1.00	04/15/2002	06/01/2002
000260	AD3113	180	1.00	06/01/2002	07/01/2002
000270	AD3113	60	0.50	03/01/2002	04/01/2002
000270	AD3113	60	1.00	04/01/2002	09/01/2002
000270	AD3113	60	0.25	09/01/2002	10/15/2002
000270	AD3113	70	0.75	09/01/2002	10/15/2002
000270	AD3113	70	1.00	10/15/2002	02/01/2003
000270	AD3113	80	1.00	01/01/2002	03/01/2002
000270	AD3113	80	0.50	03/01/2002	04/01/2002
000030	IF1000	10	0.50	06/01/2002	01/01/2003
000130	IF1000	90	1.00	10/01/2002	01/01/2003
000130	IF1000	100	0.50	10/01/2002	01/01/2003
000140	IF1000	90	0.50	10/01/2002	01/01/2003
000030	IF2000	10	0.50	01/01/2002	01/01/2003
000140	IF2000	100	1.00	01/01/2002	03/01/2002
000140	IF2000	100	0.50	03/01/2002	07/01/2002
000140	IF2000	110	0.50	03/01/2002	07/01/2002
000140	IF2000	110	0.50	10/01/2002	01/01/2003
000010	MA2100	10	0.50	01/01/2002	11/01/2002
000110	MA2100	20	1.00	01/01/2002	03/01/2003
000010	MA2110	10	1.00	01/01/2002	02/01/2003
000200	MA2111	50	1.00	01/01/2002	06/15/2002
000200	MA2111	60	1.00	06/15/2002	02/01/2003
000220	MA2111	40	1.00	01/01/2002	02/01/2003

Figure 1079, EMPPROJACT sample table – DATA (part 1 of 2)

EMPNO	PROJNO	ACTNO	EMPTIME	EMSTDATE	EMENDATE
000150	MA2112	60	1.00	01/01/2002	07/15/2002
000150	MA2112	180	1.00	07/15/2002	02/01/2003
000170	MA2112	60	1.00	01/01/2002	06/01/2003
000170	MA2112	70	1.00	06/01/2002	02/01/2003
000190	MA2112	70	1.00	01/01/2002	10/01/2002
000190	MA2112	80	1.00	10/01/2002	10/01/2003
000160	MA2113	60	1.00	07/15/2002	02/01/2003
000170	MA2113	80	1.00	01/01/2002	02/01/2003
000180	MA2113	70	1.00	04/01/2002	06/15/2002
000210	MA2113	80	0.50	10/01/2002	02/01/2003
000210	MA2113	180	0.50	10/01/2002	02/01/2003
000050	OP1000	10	0.25	01/01/2002	02/01/2003
000090	OP1010	10	1.00	01/01/2002	02/01/2003
000280	OP1010	130	1.00	01/01/2002	02/01/2003
000290	OP1010	130	1.00	01/01/2002	02/01/2003
000300	OP1010	130	1.00	01/01/2002	02/01/2003
000310	OP1010	130	1.00	01/01/2002	02/01/2003
000050	OP2010	10	0.75	01/01/2002	02/01/2003
000100	OP2010	10	1.00	01/01/2002	02/01/2003
000320	OP2011	140	0.75	01/01/2002	02/01/2003
000320	OP2011	150	0.25	01/01/2002	02/01/2003
000330	OP2012	140	0.25	01/01/2002	02/01/2003
000330	OP2012	160	0.75	01/01/2002	02/01/2003
000340	OP2013	140	0.50	01/01/2002	02/01/2003
000340	OP2013	170	0.50	01/01/2002	02/01/2003
000020	PL2100	30	1.00	01/01/2002	09/15/2002

Figure 1080, EMPPROJACT sample table – DATA (part 2 of 2)

IN_TRAY

```
CREATE TABLE IN_TRAY
(RECEIVED          TIMESTAMP
, SOURCE           CHARACTER(8)
, SUBJECT          CHARACTER(64)
, NOTE_TEXT       VARCHAR(3000))
IN USERSPACE1;
```

Figure 1081, IN_TRAY sample table – DDL

The data values in the last two columns below have been truncated:

RECEIVED	SOURCE	SUBJECT	NOTE_TEXT
1988-12-25-17.12.30.000000	BADAMSON	FWD: Fantastic	To: JWALKER Cc: QU
1988-12-23-08.53.58.000000	ISTERN	FWD: Fantastic	To: Dept_D11 Con
1988-12-22-14.07.21.136421	CHAAS	Fantastic year	To: All_Managers

Figure 1082, IN_TRAY sample table – DATA

INVENTORY

```
CREATE TABLE INVENTORY
(PID              VARCHAR(10)          NOT NULL
, QUANTITY        INTEGER
, LOCATION        VARCHAR(128))
IN IBMDB2SAMPLEXML;

ALTER TABLE INVENTORY
ADD CONSTRAINT PK_INVENTORY PRIMARY KEY
(PID);
```

Figure 1083, INVENTORY sample table – DDL

PID	QUANTITY	LOCATION
100-100-01	5	-
100-101-01	25	Store
100-103-01	55	Store
100-201-01	99	Warehouse

Figure 1084, INVENTORY sample table – DATA

ORG

```
CREATE TABLE ORG
(DEPTNUMB SMALLINT NOT NULL
,DEPTNAME VARCHAR(14)
,MANAGER SMALLINT
,DIVISION VARCHAR(10)
,LOCATION VARCHAR(13))
IN USERSPACE1;
```

Figure 1085, 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 1086, ORG sample table – DATA

PRODUCT

```
CREATE TABLE PRODUCT
(PID VARCHAR(10) NOT NULL
,NAME VARCHAR(128)
,PRICE DECIMAL(30,2)
,PROMOPRICE DECIMAL(30,2)
,PROMOSTART DATE
,PROMOEND DATE
,DESCRIPTION XML)
IN IBMDB2SAMPLEXML;

ALTER TABLE PRODUCT
ADD CONSTRAINT PK_PRODUCT PRIMARY KEY
(PID);
```

Figure 1087, PRODUCT sample table – DDL

The NAME column below has been truncated:

PID	NAME	PRICE	PROMOPRICE	PROMOSTART	PROMOEND
100-100-01	Snow Shovel, Ba	9.99	7.25	11/19/2004	12/19/2004
100-101-01	Snow Shovel, De	19.99	15.99	12/18/2005	02/28/2006
100-103-01	Snow Shovel, Su	49.99	39.99	12/22/2005	02/22/2006
100-201-01	Ice Scraper, Wi	3.99	-	-	-

Figure 1088, PRODUCT sample table – DATA

PRODUCTSUPPLIER

```
CREATE TABLE PRODUCTSUPPLIER
(PID VARCHAR(10) NOT NULL
,SID VARCHAR(10) NOT NULL)
IN IBMDB2SAMPLEXML;
```

Figure 1089, PRODUCTSUPPLIER sample table – DDL

There is no data in this table.

PROJACT

```
CREATE TABLE PROJACT
( PROJNO          CHARACTER(6)          NOT NULL
, ACTNO           SMALLINT              NOT NULL
, ACSTAFF         DECIMAL(5,2)          NOT NULL
, ACSTDATE        DATE                  NOT NULL
, ACENDATE        DATE)
IN USERSPACE1;
```

```
ALTER TABLE PROJACT
ADD CONSTRAINT PK_PROJACT PRIMARY KEY
( PROJNO
, ACTNO
, ACSTDATE );
```

Figure 1090, PROJACT sample table – DDL

PROJNO	ACTNO	ACSTAFF	ACSTDATE	ACENDATE
AD3100	10	-	2002-01-01	-
AD3110	10	-	2002-01-01	-
AD3111	60	-	2002-01-01	-
AD3111	60	-	2002-03-15	-
AD3111	70	-	2002-03-15	-
AD3111	80	-	2002-04-15	-
AD3111	180	-	2002-10-15	-
AD3111	70	-	2002-02-15	-
AD3111	80	-	2002-09-15	-
AD3112	60	-	2002-01-01	-
AD3112	60	-	2002-02-01	-
AD3112	60	-	2003-01-01	-
AD3112	70	-	2002-02-01	-
AD3112	70	-	2002-03-15	-
AD3112	70	-	2002-08-15	-
AD3112	80	-	2002-08-15	-
AD3112	80	-	2002-10-15	-
AD3112	180	-	2002-08-15	-
AD3113	70	-	2002-06-15	-
AD3113	70	-	2002-07-01	-
AD3113	80	-	2002-01-01	-
AD3113	80	-	2002-03-01	-
AD3113	180	-	2002-03-01	-
AD3113	180	-	2002-04-15	-
AD3113	180	-	2002-06-01	-
AD3113	60	-	2002-03-01	-
AD3113	60	-	2002-04-01	-
AD3113	60	-	2002-09-01	-
AD3113	70	-	2002-09-01	-
AD3113	70	-	2002-10-15	-
IF1000	10	-	2002-06-01	-
IF1000	90	-	2002-10-01	-
IF1000	100	-	2002-10-01	-
IF2000	10	-	2002-01-01	-
IF2000	100	-	2002-01-01	-
IF2000	100	-	2002-03-01	-
IF2000	110	-	2002-03-01	-
IF2000	110	-	2002-10-01	-
MA2100	10	-	2002-01-01	-
MA2100	20	-	2002-01-01	-
MA2110	10	-	2002-01-01	-
MA2111	50	-	2002-01-01	-
MA2111	60	-	2002-06-15	-
MA2111	40	-	2002-01-01	-

Figure 1091, PROJACT sample table – DATA (part 1 of 2)

PROJNO	ACTNO	ACSTAFF	ACSTDATE	ACENDATE
MA2112	60	-	2002-01-01	-
MA2112	180	-	2002-07-15	-
MA2112	70	-	2002-06-01	-
MA2112	70	-	2002-01-01	-
MA2112	80	-	2002-10-01	-
MA2113	60	-	2002-07-15	-
MA2113	80	-	2002-01-01	-
MA2113	70	-	2002-04-01	-
MA2113	80	-	2002-10-01	-
MA2113	180	-	2002-10-01	-
OP1000	10	-	2002-01-01	-
OP1010	10	-	2002-01-01	-
OP1010	130	-	2002-01-01	-
OP2010	10	-	2002-01-01	-
OP2011	140	-	2002-01-01	-
OP2011	150	-	2002-01-01	-
OP2012	140	-	2002-01-01	-
OP2012	160	-	2002-01-01	-
OP2013	140	-	2002-01-01	-
OP2013	170	-	2002-01-01	-
PL2100	30	-	2002-01-01	-

Figure 1092, *PROJACT* sample table – DATA (part 2 of 2)

PROJECT

```

CREATE TABLE PROJECT
( PROJNO          CHARACTER(6)          NOT NULL
, PROJNAME       VARCHAR(24)          NOT NULL
, DEPTNO        CHARACTER(3)          NOT NULL
, RESPEMP       CHARACTER(6)          NOT NULL
, PRSTAFF       DECIMAL(5,2)
, PRSTDATE      DATE
, PRENDATE      DATE
, MAJPROJ       CHARACTER(6)
IN USERSPACE1;

ALTER TABLE PROJECT
ADD CONSTRAINT PK_PROJECT PRIMARY KEY
( PROJNO );

CREATE INDEX XPROJ2 ON PROJECT
( RESPEMP          ASC)
ALLOW REVERSE SCANS;

CREATE ALIAS PROJ FOR PROJECT;

```

Figure 1093, *PROJECT* sample table – DDL

PROJNO	PROJNAME	DEP	RESPEMP	PRSTDATE	PRENDATE	MAJPROJ
AD3100	ADMIN SERVICES	D01	000010	01/01/2002	02/01/2003	-
AD3110	GENERAL ADMIN SYSTEMS	D21	000070	01/01/2002	02/01/2003	AD3100
AD3111	PAYROLL PROGRAMMING	D21	000230	01/01/2002	02/01/2003	AD3110
AD3112	PERSONNEL PROGRAMMING	D21	000250	01/01/2002	02/01/2003	AD3110
AD3113	ACCOUNT PROGRAMMING	D21	000270	01/01/2002	02/01/2003	AD3110
IF1000	QUERY SERVICES	C01	000030	01/01/2002	02/01/2003	-
IF2000	USER EDUCATION	C01	000030	01/01/2002	02/01/2003	-
MA2100	WELD LINE AUTOMATION	D01	000010	01/01/2002	02/01/2003	-
MA2110	W L PROGRAMMING	D11	000060	01/01/2002	02/01/2003	MA2100
MA2111	W L PROGRAM DESIGN	D11	000220	01/01/2002	12/01/1982	MA2110
MA2112	W L ROBOT DESIGN	D11	000150	01/01/2002	12/01/1982	MA2110
MA2113	W L PROD CONT PROGS	D11	000160	02/15/2002	12/01/1982	MA2110
OP1000	OPERATION SUPPORT	E01	000050	01/01/2002	02/01/2003	-
OP1010	OPERATION	E11	000090	01/01/2002	02/01/2003	OP1000
OP2000	GEN SYSTEMS SERVICES	E01	000050	01/01/2002	02/01/2003	-
OP2010	SYSTEMS SUPPORT	E21	000100	01/01/2002	02/01/2003	OP2000
OP2011	SCP SYSTEMS SUPPORT	E21	000320	01/01/2002	02/01/2003	OP2010
OP2012	APPLICATIONS SUPPORT	E21	000330	01/01/2002	02/01/2003	OP2010
OP2013	DB/DC SUPPORT	E21	000340	01/01/2002	02/01/2003	OP2010
PL2100	WELD LINE PLANNING	B01	000020	01/01/2002	09/15/2002	MA2100

Figure 1094, PROJECT sample table – DATA

PURCHASEORDER

```
CREATE TABLE PURCHASEORDER
(POID          BIGINT          NOT NULL
,STATUS        VARCHAR(10)    NOT NULL
,CUSTID        BIGINT
,ORDERDATE    DATE
,POORDER      XML
,COMMENTS     VARCHAR(1000))
IN IBMDB2SAMPLEXML;
```

```
ALTER TABLE PURCHASEORDER
ADD CONSTRAINT PK_PURCHASEORDER PRIMARY KEY
(POID);
```

Figure 1095, PURCHASEORDER sample table – DDL

POID	STATUS	CUSTID	ORDERDATE	POORDER	COMMENTS
5000	Unshipped	1002	02/18/2006	<<xml>>	THIS IS A NEW PURCHASE ORDER
5001	Shipped	1003	02/03/2005	<<xml>>	THIS IS A NEW PURCHASE ORDER
5002	Shipped	1001	02/29/2004	<<xml>>	THIS IS A NEW PURCHASE ORDER
5003	Shipped	1002	02/28/2005	<<xml>>	THIS IS A NEW PURCHASE ORDER
5004	Shipped	1005	11/18/2005	<<xml>>	THIS IS A NEW PURCHASE ORDER
5006	Shipped	1002	03/01/2006	<<xml>>	THIS IS A NEW PURCHASE ORDER

Figure 1096, PURCHASEORDER sample table – DATA

SALES

```
CREATE TABLE SALES
(SALES_DATE    DATE
,SALES_PERSON  VARCHAR(15)
,REGION        VARCHAR(15)
,SALES         INTEGER)
IN USERSPACE1;
```

Figure 1097, SALES sample table – DDL

SALES_DATE	SALES_PERSON	REGION	SALES
12/31/2005	LUCCHESSI	Ontario-South	1
12/31/2005	LEE	Ontario-South	3
12/31/2005	LEE	Quebec	1
12/31/2005	LEE	Manitoba	2
12/31/2005	GOUNOT	Quebec	1
03/29/2006	LUCCHESSI	Ontario-South	3
03/29/2006	LUCCHESSI	Quebec	1
03/29/2006	LEE	Ontario-South	2
03/29/1996	LEE	Ontario-North	2
03/29/2006	LEE	Quebec	3
03/29/2006	LEE	Manitoba	5
03/29/2006	GOUNOT	Ontario-South	3
03/29/2006	GOUNOT	Quebec	1
03/29/2006	GOUNOT	Manitoba	7
03/30/2006	LUCCHESSI	Ontario-South	1
03/30/2006	LUCCHESSI	Quebec	2
03/30/2006	LUCCHESSI	Manitoba	1
03/30/2006	LEE	Ontario-South	7
03/30/2006	LEE	Ontario-North	3
03/30/2006	LEE	Quebec	7
03/30/2006	LEE	Manitoba	4
03/30/2006	GOUNOT	Ontario-South	2
03/30/2006	GOUNOT	Quebec	18
03/31/2006	GOUNOT	Manitoba	1
03/31/2006	LUCCHESSI	Manitoba	1
03/31/2006	LEE	Ontario-South	14
03/31/2006	LEE	Ontario-North	3
03/31/2006	LEE	Quebec	7
03/31/2006	LEE	Manitoba	3
03/31/2006	GOUNOT	Ontario-South	2
03/31/2006	GOUNOT	Quebec	1
04/01/2006	LUCCHESSI	Ontario-South	3
04/01/2006	LUCCHESSI	Manitoba	1
04/01/2006	LEE	Ontario-South	8
04/01/2006	LEE	Ontario-North	-
04/01/2006	LEE	Quebec	8
04/01/2006	LEE	Manitoba	9
04/01/2006	GOUNOT	Ontario-South	3
04/01/2006	GOUNOT	Ontario-North	1
04/01/2006	GOUNOT	Quebec	3
04/01/2006	GOUNOT	Manitoba	7

Figure 1098, SALES sample table – DATA

STAFF

```

CREATE TABLE STAFF
  ( ID                SMALLINT                NOT NULL
  , NAME              VARCHAR(9)
  , DEPT              SMALLINT
  , JOB               CHARACTER(5)
  , YEARS            SMALLINT
  , SALARY            DECIMAL(7,2)
  , COMM              DECIMAL(7,2)
  IN USERSPACE1;

```

Figure 1099, STAFF sample table – DDL

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
10	Sanders	20	Mgr	7	98357.50	-
20	Pernal	20	Sales	8	78171.25	612.45
30	Marenghi	38	Mgr	5	77506.75	-
40	O'Brien	38	Sales	6	78006.00	846.55
50	Hanes	15	Mgr	10	80659.80	-
60	Quigley	38	Sales	-	66808.30	650.25
70	Rothman	15	Sales	7	76502.83	1152.00
80	James	20	Clerk	-	43504.60	128.20
90	Koonitz	42	Sales	6	38001.75	1386.70
100	Plotz	42	Mgr	7	78352.80	-
110	Ngan	15	Clerk	5	42508.20	206.60
120	Naughton	38	Clerk	-	42954.75	180.00
130	Yamaguchi	42	Clerk	6	40505.90	75.60
140	Fraye	51	Mgr	6	91150.00	-
150	Williams	51	Sales	6	79456.50	637.65
160	Molinare	10	Mgr	7	82959.20	-
170	Kermisch	15	Clerk	4	42258.50	110.10
180	Abrahams	38	Clerk	3	37009.75	236.50
190	Sneider	20	Clerk	8	34252.75	126.50
200	Scoutten	42	Clerk	-	41508.60	84.20
210	Lu	10	Mgr	10	90010.00	-
220	Smith	51	Sales	7	87654.50	992.80
230	Lundquist	51	Clerk	3	83369.80	189.65
240	Daniels	10	Mgr	5	79260.25	-
250	Wheeler	51	Clerk	6	74460.00	513.30
260	Jones	10	Mgr	12	81234.00	-
270	Lea	66	Mgr	9	88555.50	-
280	Wilson	66	Sales	9	78674.50	811.50
290	Quill	84	Mgr	10	89818.00	-
300	Davis	84	Sales	5	65454.50	806.10
310	Graham	66	Sales	13	71000.00	200.30
320	Gonzales	66	Sales	4	76858.20	844.00
330	Burke	66	Clerk	1	49988.00	55.50
340	Edwards	84	Sales	7	67844.00	1285.00
350	Gafney	84	Clerk	5	43030.50	188.00

Figure 1100, STAFF sample table – DATA

SUPPLIERS

```
CREATE TABLE SUPPLIERS
(SID          VARCHAR(10)          NOT NULL
,ADDR        XML)
IN IBMDB2SAMPLEXML;
```

```
ALTER TABLE SUPPLIERS
ADD CONSTRAINT PK_PRODUCTSUPPLIER PRIMARY KEY
(SID);
```

Figure 1101, SUPPLIERS sample table – DDL

There is no data in this table.

Book Binding

Below is a quick-and-dirty technique for making a book out of this book. The object of the exercise is to have a manual that will last a long time, and that will also lie flat when opened up. All suggested actions are done at your own risk.

Tools Required

Printer, to print the book.

- KNIFE, to trim the tape used to bind the book.
- BINDER CLIPS, (1" size), to hold the pages together while gluing. To bind larger books, or to do multiple books in one go, use two or more cheap screw clamps.
- CARDBOARD: Two pieces of thick card, to also help hold things together while gluing.

Consumables

Ignoring the capital costs mentioned above, the cost of making a bound book should work out to about \$4.00 per item, almost all of which is spent on the paper and toner. To bind an already printed copy should cost less than fifty cents.

- PAPER and TONER, to print the book.
- CARD STOCK, for the front and back covers.
- GLUE, to bind the book. Cheap rubber cement will do the job. The glue must come with an applicator brush in the bottle. Sears hardware stores sell a more potent flavor called Duro Contact Cement that is quite a bit better. This is toxic stuff, so be careful.
- CLOTH TAPE, (2" wide) to bind the spine. Pearl tape, available from Pearl stores, is fine. Wider tape will be required if you are not printing double-sided.
- TIME: With practice, this process takes less than five minutes work per book.

Before you Start

- Make that sure you have a well-ventilated space before gluing.
- Practice binding on some old scraps of paper.
- Kick all kiddies out off the room.

Instructions

- Print the book - double-sided if you can. If you want, print the first and last pages on card stock to make suitable protective covers.
- Jog the pages, so that they are all lined up along the inside spine. Make sure that every page is perfectly aligned, otherwise some pages won't bind. Put a piece of thick cardboard on either side of the set of pages to be bound. These will hold the pages tight during the gluing process.

- Place binder clips on the top and bottom edges of the book (near the spine), to hold everything in place while you glue. One can also put a couple on the outside edge to stop the pages from splaying out in the next step. If the pages tend to spread out in the middle of the spine, put one in the centre of the spine, then work around it when gluing. Make sure there are no gaps between leaves, 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 than 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 plenty of other binding methods that take a little more skill and better gear that can be used to make "store-quality" books. Search the web for more information.

Bibliography

This section documents various useful sources of information on DB2 and SQL. However, far and away the best source of DB2 help is Google. To check an error-code (e.g. SQL0402N) simply enter the code in Google, and you should get a link to a page that describes it. To get the official IBM webpage on the code, add "boulder" (as in Boulder, Colorado) to the search.

GOOGLE SEARCH TERM =====	FIND LINKS TO =====
DB2 9 MANUALS	DB2 manuals
DB2 9 REDBOOKS	DB2 Redbooks
DB2 ONLINE TUTORIALS	DB2 Online Tutorials
DB2 INFORMATION CENTER	Online DB2 information center
DB2 MAGAZINE	DB2 Magazine

Figure 1102, Google Search Terms

IBM Sources

DB2 Manuals

All of the DB2 manuals are available for free from the IBM website. A selected list is provided below:

```

ADMINISTRATION
Administration Guide: Planning
Administration Guide: Implementation
Administration Guide: Performance
Administrative API Reference
Administrative SQL Routines and Views
Data Recovery and High Availability Guide and Reference
Data Movement Utilities Guide and Reference
Performance Guide
APPLICATION DEVELOPMENT
Developing ADO NET and OLE DB Applications
Developing Embedded SQL Applications
Developing Java Applications
Developing Perl and PHP Applications
Developing SQL and External Routines
REFERENCE
Command Reference
Message Reference, Vol. 1
Message Reference, Vol. 2
SQL Reference, Vol. 1
SQL Reference, Vol. 2

```

Figure 1103, DB2 Manuals

Red Books

IBM occasionally publishes a "red book" on a technical subject that involves DB2. All of the Redbooks are available for free from the IBM website. Below is a selective list:

```

DB2 9 pureXML Guide
DB2 Express-C: The Developer Handbook for XML, PHP, C/C++, Java & .NET

```

Figure 1104, DB2 Redbooks

The IBM Redbook website has its own search engine. It is often better at finding suitable books than external search engines.

Online Tutorials

The IBM website has a series of online tutorials that are designed for DBAs who are familiar with the basic concepts, and who want to study for the DB2 certification tests. Some of these tutorials are free, some are not. One must register with IBM before using the tutorials. They can be run online (note: JavaScript must be enabled on your browser), or downloaded in PDF format.

Other Sources

Books Published

The following books have been published on DB2:

Understanding DB2 - Learning Visually with Examples
 Author: Chong/Liu/Qi/Snow
 ISBN: 0131859161

DB2 UDB v8.1 for Linux, UNIX, and Windows Database Administration
 Certification Guide
 Author: Baklarz & Wong
 ISBN: 0-13-046361-2
 Textbook Binding - 912 pages, 5th edition (Feb 2003)
 Prentice Hall

Teach Yourself DB2 Universal Database in 21 Days
 Author: Visser & Wong
 ISBN: 067232582
 Paperback - 640 pages, 2nd edition (Aug 2003)
 Sams

DB2 UDB V8.1 Application Development Certification Guide
 Author: Sanyal, et. al
 ISBN: 0-13-046391-4
 2nd edition (April 2003)
 Prentice Hall

The Official Introduction to DB2 UDB for z/OS Version 8
 Author: Sloan
 ISBN: 0-13-147750-1
 Textbook Binding - 496 pages, 1st edition (Apr 2004)
 Prentice Hall

DB2 UDB V8.1 Certification Exam 700 Study Guide
 Publication order number: :

Figure 1105, DB2 Books

Roger Sanders Books

Roger Sanders has written the following books:

DB2 UDB V8.1 Certification Exam 700 Study Guide
 DB2 UDB V8.1 Certification Exams 701 and 706 Study Guide
 DB2 UDB V8.1 Certification Exam 703 Study Guide

Figure 1106, Books by Roger Sanders

DB2 Magazine

The DB2 Magazine is published quarterly. One can subscribe from the magazine website, or read issues online. The website has a set of "electronic books", which a good introduction to certain subjects.

Index

A

ABS function, 117
 ACOS function, 118
 Adaptive Query, 343
 ADD function. *See* PLUS function
 AFTER trigger. *See* Triggers
 AGGREGATION function
 BETWEEN, 112
 Definition, 108
 ORDER BY, 110
 PARTITION, 115
 RANGE, 114
 ROWS, 111
 Alias, 21
 ALL, sub-query, 227, 237
 AND vs. OR, precedence rules, 39
 ANY, sub-query, 226, 235
 Arithmetic, precedence rules, 39
 AS statement
 Common table expression, 278
 Correlation name, 32
 Renaming fields, 32
 ASCII function, 118
 ASIN function, 118
 ATAN function, 118
 ATOMIC, BEGIN statement, 75
 AVG
 Compared to median, 396
 Date value, 86
 Function, 85, 397
 Null usage, 86

B

Balanced hierarchy, 297
 BEFORE trigger. *See* Triggers
 BEGIN ATOMIC statement, 75
 BERNOULI option. *See* TABLESAMPLE feature
 BETWEEN
 AGGREGATION function, 112
 Predicate, 36
 BIGINT function, 118, 416
 BLOB function, 119
 BLOCK LOCATE user defined function, 300
 Business day calculation, 393

C

Cartesian Product, 214
 CASE expression
 Character to number, 374
 Definition, 44
 Predicate use, 47
 Recursive processing, 309
 Sample data creation, usage, 368
 Selective column output, 380
 UPDATE usage, 46

 Wrong sequence, 411
 Zero divide (avoid), 46
 CAST expression
 Definition, 41
 CEIL function, 119
 CHAR function, 120, 376
 Character to number, convert, 178, 374
 Chart making using SQL, 380
 Check input is numeric, 178
 CHR function, 122
 Circular Reference. *See* You are lost
 Clean hierarchies, 305
 CLOB function, 123
 CLOSE cursor, 48
 COALESCE function, 123, 216
 Comma usage in number display, 378
 Comment in SQL, 19
 COMMIT statement, 53
 Common table expression
 Definition, 278
 Fullselect clause, 280
 Fullselect comparison, 28
 Subselect comparison, 28
 Compound SQL
 DECLARE variables, 76
 Definition, 75
 FOR statement, 77
 IF statement, 78
 LEAVE statement, 79
 Scalar function, 170
 SIGNAL statement, 79
 Table function, 173
 WHILE statement, 80
 CONCAT function, 124, 164
 Constraint, 321, 322
 Convergent hierarchy, 296
 Convert
 Character to number, 178, 374
 Commas added to display, 378
 Date-time to character, 121
 Decimal to character, 377
 HEX value to number, 399
 Integer to character, 376
 Timestamp to numeric, 379
 Correlated sub-query
 Definition, 232
 NOT EXISTS, 234
 CORRELATION function, 87
 Correlation name, 32
 COS function, 124
 COT function, 125
 COUNT DISTINCT function
 Definition, 87
 Null values, 98
 COUNT function
 Definition, 87

- No rows, 88, 201, 404
- Null values, 87
- COUNT_BIG function, 88
- COVARIANCE function, 88
- Create Table
 - Constraint, 321, 322
 - Dimensions, 255
 - Example, 20
 - Identity Column, 258, 261
 - Indexes, 254
 - Materialized query table, 245
 - Referential Integrity, 321, 322
 - Staging tables, 255
- CUBE, 195
- Current maintained types, 247
- Current query optimization, 247
- Current refresh age, 246
- Cursor, 48
- Cursor Stability, 414

D

- Data in view definition, 21
- Data types, 22, 27
- DATE
 - Arithmetic, 23
 - AVG calculation, 86
 - Convert to CHAR, 121
 - Duration, 25
 - Function, 125
 - Get prior, 174
 - Manipulation, 23, 405, 408
 - Output order, 411
- DAY function, 126
- DAYNAME function, 126
- DAYOFWEEK function, 126, 127, 393
- DAYOFYEAR function, 127
- DAYS function, 127
- DBPARTITIONNUM function, 128
- DECIMAL
 - Commas added to display, 378
 - Convert to character, 377
 - Function, 128, 416
 - Multiplication, 40, 143
 - Timestamp conversion, 379
- DECLARE cursor, 48
- DECLARE variables, 76
- Declared Global Temporary Table, 276, 284
- DECRYPT_BIN function, 129
- DECRYPT_CHAR function, 129
- Deferred Refresh tables, 248
- DEGRESS function, 129
- DELETE
 - Counting using triggers, 269
 - Definition, 64
 - Fetch first n rows, 65
 - Fullselect, 65
 - MERGE usage, 71
 - Multiple tables usage, 242
 - Nested table expression, 66
 - OLAP functions, 65
 - Select results, 68
 - Stop after n rows, 65
- Delimiter, statement, 20, 75
- Denormalize data, 385

- DENSE_RANK function, 96
- DESCRIBE statement, 51
- DETERMINISTIC statement, 167
- DIFFERENCE function, 129
- DIGITS function, 130, 376
- DISTINCT, 85, 116
- Distinct types, 22, 27
- Divergent hierarchy, 295
- DIVIDE "/" function, 163
- Divide by zero (avoid), 46
- DOUBLE function, 131
- Double quotes, 34
- Duration, Date/Time, 25

E

- ELSE. *See* CASE expression
- ENCRYPT function, 132
- ESCAPE phrase, 38
- EXCEPT, 240
- EXECUTE IMMEDIATE statement, 52
- EXECUTE statement, 52
- EXISTS, sub-query, 36, 228, 233, 234
- EXP function, 132

F

- FETCH FIRST n rows
 - Definition, 31
 - DELETE usage, 65
 - Duplicate value issues, 104
 - Efficient usage, 106
 - UPDATE usage, 62
- FETCH from cursor, 48
- Fibonacci Series, 391
- FLOAT function, 133, 416
- Floating-point numbers, 416
- FLOOR function, 133
- FOR statement, 77
- Foreign key, 321
- Fractional date manipulation, 408
- Frictionless Query, 343
- Full Outer join
 - Definition, 210
- Full Outer Join
 - COALESCE function, 216
- Fullselect
 - Definition, 280
 - Defintion, 28
 - DELETE usage, 65
 - INSERT usage, 58, 59
 - MERGE usage, 72
 - TABLE function, 281
 - UPDATE usage, 62, 63, 283
- Function (user defined). *See* User defined function

G

- GENERATE_UNIQUE function, 133, 366
- Generating SQL, 337, 341
- GET DIAGNOSTICS statement, 77
- GETHINT function, 135
- Global Temporary Table, 276, 284
- GROUP BY
 - CUBE, 195
 - Definition, 184
 - GROUPING SETS, 187
 - Join usage, 200

- ORDER BY usage, 200
- PARTITION comparison, 115
- ROLLUP, 191
 - Zero rows match, 404
- GROUPING function, 89, 189
- GROUPING SETS, 187

H

- HASHEDVALUE function, 135
- HAVING
 - Definition, 184
 - Sample queries, 186
 - Zero rows match, 404
- HEX
 - Covert value to number, 399
 - Function, 135
- Hierarchy
 - Balanced, 297
 - Convergent, 296
 - Denormalizing, 305
 - Divergent, 295
 - Recursive, 296
 - Summary tables, 305
 - Triggers, 305
- History tables, 327, 330
- HOUR function, 136

I

- Identity column
 - IDENTITY_VAL_LOCAL function, 263
 - Restart value, 261
 - Usage notes, 257
- IDENTITY_VAL_LOCAL function, 136, 263
- IF statement, 78
- Immediate Refresh tables, 249
- IN
 - Multiple predicates, 233
 - Predicate, 37
 - Sub-query, 231, 233
- Index on materialized query table, 254
- Inner Join
 - Definition, 206
 - ON and WHERE usage, 207
 - Outer followed by inner, 222
- INPUT SEQUENCE, 67
- INSERT
 - 24-hour timestamp notation, 403
 - Common table expression, 60, 280
 - Definition, 57
 - Fullselect, 58, 59, 282
 - Function, 137
 - MERGE usage, 71
 - Multiple tables usage, 60, 242
 - Select results, 67
 - UNION All usage, 60, 242
 - Unique timestamp generation, 134
 - WITH statement, 60, 280
- INSTEAD OF. *See* Triggers
- INTEGER
 - Arithmetic, 39
 - Convert to character, 376
 - Function, 137
 - Truncation, 410

- Intelligent Comment, 19
- INTERSECT, 240
- Isolation level, 413
- ITERATE statement, 78

J

- Java code
 - Scalar function, 348
 - Tabular function, 350
 - Transpose function, 357
- Join
 - Cartesian Product, 214
 - COALESCE function, 216
 - DISTINCT usage warning, 85
 - Full Outer Join, 210
 - GROUP BY usage, 200
 - Inner Join, 206
 - Left Outer Join, 207
 - Materialized query tables, 251, 252
 - Null usage, 216
 - Right Outer Join, 209
 - Syntax, 203
- Julian Date
 - Format defintion, 125
 - User defined function, 174
- JULIAN_DAY function
 - Definition, 137
 - History, 138

L

- LCASE function, 139
- LEAVE statement, 79
- LEFT function, 140
- Left Outer Join, 207
- LENGTH function, 140
- LIKE predicate
 - Definition, 37
 - ESCAPE usage, 38
 - Varchar usage, 409
- LN function, 140
- LOCATE function, 140
- LOCATE_BLOCK user defined function, 300
- LOG function, 141
- LOG10 function, 141
- Lousy Index. *See* Circular Reference
- LTRIM function, 141

M

- Matching rows, zero, 404
- Materialized query tables
 - Current maintained types, 247
 - Current query optimization, 247
 - Current refresh age, 246
 - DDL restrictions, 245
 - Dimensions, 255
 - Duplicate data, 250
 - Index usage, 254
 - Join usage, 251, 252
 - Optimizer options, 246
 - Refresh Deferred, 248
 - Refresh Immediate, 249
 - Staging tables, 255
 - Syntax diagram, 243

MAX

- Function, 89
- Rows, getting, 103
- Values, getting, 100, 104
- Median, 396
- MERGE
 - Definition, 69
 - DELETE usage, 71
 - Fullselect, 72
 - INSERT usage, 71
 - UPDATE usage, 71
- Meta-data
 - Generate SQL within SQL, 341
 - Java function queries, 348
 - SQL function queries, 344
 - Update real data, 359
- Meta-Data to Real-Data Join, 343
- MICROSECOND function, 142
- MIDNIGHT_SECONDS function, 142
- MIN function, 90
- MINUS "-" function, 163
- MINUTE function, 142
- Missing rows, 382
- MOD function, 143
- MONTH
 - Function, 143
 - Get prior, 175
 - User defined function example, 175
- MONTHNAME function, 143
- MULTIPLY_ALT function, 143
- Multiplication, overflow, 143
- MULTIPLY "*" function, 163

N

- Nested table expression
 - DELETE usage, 66, 69
 - SELECT usage, 224, 275, 279
 - Simplified syntax, 224
 - TABLE function, 155
 - UPDATE usage, 62
- NEXTVAL expression, 265
- Nickname, 22
- No rows match, 404
- Normalize data, 384
- NOT EXISTS, sub-query, 232, 234
- NOT IN, sub-query, 231, 234
- NOT predicate, 35
- NULLIF function, 144
- Nulls
 - CASE expression, 44
 - CAST expression, 41
 - COUNT DISTINCT function, 87, 98
 - COUNT function, 234
 - Definition, 33
 - GROUP BY usage, 184
 - Join usage, 216
 - Order sequence, 181
 - Predicate usage, 39
 - RANK function, 98
 - Ranking, 98
 - User defined function output, 168
- Numbered list generate - user defined function, 176
- Numeric input check, 178

O

- OLAP functions
 - AGGREGATION function, 108
 - DELETE usage, 65
 - DENSE_RANK function, 96
 - RANK function, 96
 - ROW_NUMBER function, 102
 - UPDATE usage, 63
- ON vs. WHERE, joins, 205, 207, 208, 210, 211
- OPEN cursor, 48
- OPTIMIZE FOR clause, 106
- OR vs. AND, precedence rules, 39
- ORDER BY
 - AGGREGATION function, 110
 - CONCAT function, 124
 - Date usage, 411
 - Definition, 181
 - FETCH FIRST, 31
 - GROUP BY usage, 200
 - Nulls processing, 98, 181
 - RANK function, 97
 - ROW_NUMBER function, 102
- Outer Join
 - COALESCE function, 216
 - Definition, 210
 - ON vs. WHERE, joins, 208, 210, 211
 - Outer followed by inner, 222
 - UNION usage, 208, 210
- Overflow errors, 143

P

- Partition
 - AGGREGATION function, 115
 - GROUP BY comparison, 115
 - RANK function, 99
 - ROW_NUMBER function, 103
- PARTITION function, 144
- PAUSE function (user defined), 394
- Percentage calculation, 276
- PLUS "+" function, 162
- POSSTR function, 144
- POWER function, 145
- Precedence rules, 39
- Predicate
 - Basic types, 35
 - BETWEEN predicate, 36
 - CASE expression, 47
 - EXISTS sub-query, 36
 - IN predicate, 37
 - LIKE predicate, 37
 - NOT predicate, 35
 - Null reference, 33, 39
- PREPARE statement, 50
- PREVVAL expression, 265
- Primary key, 321
- Processing Sequence, 40, 205, 363

Q

- Quotes, 34

R

- RAISE_ERROR function, 145
- RAND function

- Description, 146
- Predicate usage, 406
- Random row selection, 148
- Reproducible usage, 147, 365
- Random sampling. *See* TABLESAMPLE feature
- RANGE (AGGREGATION function), 114
- RANK function
 - Definition, 96
 - Nulls processing, 98
 - ORDER BY, 97
 - Partition, 99
- Read Stability, 414
- REAL function, 149
- Recursion
 - Fetch first n rows, 107
 - Halting processing, 298
 - How it works, 287
 - Level (in hierarchy), 291
 - List children, 290
 - Multiple invocations, 293
 - Normalize data, 384
 - Stopping, 298
 - Warning message, 294
 - When to use, 287
- Recursive hierarchy
 - Definition, 296
 - Denormalizing, 306, 308
 - Triggers, 306, 308
- Referential Integrity, 321, 322
- Refresh Deferred tables, 248
- Refresh Immediate tables, 249
- REGRESSION functions, 90
- RELEASE SAVEPOINT statement, 54
- REPEAT function, 149, 380
- REPEATABLE option. *See* TABLESAMPLE feature
- Repeatable Read, 414
- REPLACE function, 150
- Restart, Identity column, 261
- RETURN statement, 168
- Reversing values, 390
- RIGHT function, 150
- Right Outer Join, 209
- ROLLBACK statement, 54
- ROLLUP, 191
- ROUND function, 150
- ROW_NUMBER function, 397
 - Definition, 102
 - ORDER BY, 102
 - PARTITION BY, 103
- ROWS (AGGREGATION function), 111
- RTRIM function, 151

S

- Sample data. *See* TABLESAMPLE feature
- SAVEPOINT statement, 53
- Scalar function, user defined, 167
- SELECT statement
 - Correlation name, 32
 - Definition, 29
 - DELETE usage, 68
 - DML changes, 66
 - Fullselect, 282
 - INSERT usage, 59
 - Random row selection, 148

- SELECT INTO statement, 50
- Syntax diagram, 29, 30
- UPDATE usage, 63
- Semi-colon
 - SQL Statement usage, 39
 - Statement delimiter, 20
- Sequence
 - Create, 265
 - Multi table usage, 267
 - NEXTVAL expression, 265
 - PREVVAL expression, 265
- Sequence numbers. *See* Identity column
- SET Intelligent comment, 19
- SET statement, 52
- SIGN function, 151
- SIGNAL statement
 - Definition, 79
 - Trigger usage, 324, 325
- SIN function, 151
- SMALLINT function, 152
- SOME, sub-query, 226, 235
- Sort string, 394
- SOUNDEX function, 152
- Sourced function, 165
- SPACE function, 153
- Special register
 - Current maintained types, 247
 - Current query optimization, 247
 - Current refresh age, 246
- Special Registers, 25
- SQL Comment, 19
- SQRT function, 153
- Staging tables, 255
- Statement delimiter, 20, 75
- STDDEV function, 91
- STRIP function, 153
- Sub-query
 - Correlated, 232
 - DELETE usage, 65
 - Error prone, 226
 - EXISTS usage, 228, 233
 - IN usage, 231, 233
 - Multi-field, 233
 - Nested, 233
- Subselect, 28
- SUBSTR function
 - Definition, 154
- SUBTRACT function. *See* MINUS function
- SUM function, 91, 110
- Summary tables
 - Recursive hierarchies, 305

T

- Table. *See* Create Table
- TABLE function
 - Definition, 172
 - Fullselect, 281
 - Numbered list generate example, 176
- TABLE_NAME function, 155
- TABLE_SCHEMA function, 156
- TABLESAMPLE feature, 372
- Temporary Table
 - Common table expression, 278
 - Full select, 280

- Global Declared, 276, 284
 - Nest table expression, 275
 - TABLE function, 281
 - Terminator, statement, 20, 75
 - Test Data. *See* Sample Data
 - TIME
 - Convert to CHAR, 121
 - Duration, 25
 - Function, 156
 - Manipulation, 23
 - Time Series data, 369
 - TIMESTAMP
 - 24-hour notation, 403
 - Convert to CHAR, 121
 - Function, 156
 - Generate unique, 134
 - Manipulation, 403, 408
 - Unique generation, 134
 - TIMESTAMP_FORMAT function, 157
 - TIMESTAMP_ISO function, 157
 - TIMESTAMPDIFF function, 158
 - TO_CHAR function. *See* VARCHAR_FORMAT
 - TO_DATE function. *See* TIMETAMP_FORMAT
 - TRANSLATE function, 159, 177
 - Transpose
 - Data. *See* Denormalize data
 - User-defined function, 352
 - Triggers
 - BEFORE vs. AFTER triggers, 311, 322
 - Definition, 311
 - Delete counting, 269
 - History tables, 328, 333
 - Identity column, 270
 - INSTEAD OF triggers, 311, 333
 - Propagate changes, 325
 - Recursive hierarchies, 306, 308
 - ROW vs. STATEMENT triggers, 312
 - Sequence, 267
 - SIGNAL statement, 314, 324, 325
 - Syntax diagram, 311
 - Validate input, 314, 324, 325
 - TRIM. *See* LTRIM or RTRIM
 - TRIM function. *See* STRIP function
 - TRUNCATE function, 160
 - Truncation, numeric, 410
- U**
- UCASE function, 160
 - Unbalanced hierarchy, 297
 - Uncommitted Read, 415
 - Uncorrelated sub-query, 232
 - Nested, 233
 - UNION
 - Definition, 240
 - Outer join usage, 208, 210
 - Precedence Rules, 241
 - UNION ALL
 - Definition, 240
 - INSERT usage, 59, 60, 242
 - Recursion, 288
 - View usage, 242
 - Unique value generation
 - GENERATE_UNIQUE function, 133
 - Identity column, 257
 - Sequence, 265
 - Timestamp column, 134
- UPDATE**
- CASE usage, 46
 - Definition, 61
 - Fetch first n rows, 62
 - Fullselect, 62, 63, 283
 - MERGE usage, 71
 - Meta-data to real data, 359
 - Multiple tables usage, 242
 - Nested table expression, 62
 - OLAP functions, 63
 - Select results, 68
 - Stop after n rows, 62
- User defined function
- Data-type conversion example, 374, 376
 - Denormalize example, 385
 - Fibonacci Series, 391
 - Locate Block example, 300
 - Month number example, 175
 - Nullable output, 168
 - Numbered list generate example, 176
 - Pause query example, 394
 - Recursion usage, 300
 - Reverse example, 390
 - Scalar function, 167
 - Sort string example, 394
 - Sourced function, 165
 - Table function, 172
 - Week number example, 175
- V**
- VALUE function, 160
 - VALUES expression
 - Definition, 42
 - View usage, 44
 - VARCHAR function, 161
 - VARCHAR_FORMAT function, 161
 - VARIANCE function, 92
 - Versions (history tables), 330
 - View
 - Data in definition, 21
 - DDL example, 21, 44
 - History tables, 329, 332
 - UNION ALL usage, 242
- W**
- Wait. *See* PAUSE function
 - WEEK
 - Function, 161, 410
 - Get prior, 175
 - User defined function example, 175
 - WEEK_ISO function, 162
 - WHEN. *See* CASE expression
 - WHERE vs. ON, joins, 205, 207, 208, 210, 211
 - WHILE statement, 80
 - WITH statement
 - Cursor Stability, 414
 - Defintion, 278
 - Insert usage, 60, 280
 - Isolation level, 413
 - MAX values, getting, 104
 - Multiple tables, 279

Read Stability, 414
Recursion, 288
Repeatable Read, 414
Uncommitted Read, 415
VALUES expression, 42

X

X (hex) notation, 39

Y

YEAR function, 162
You are lost. See Lousy Index

Z

Zero divide (avoid), 46
Zero rows match, 404