Fun with DB2/2 SQL

G. Birchall

Preface

NOTES

- DISCLAIMER: This course is a best effort on my part. However, given the nature of the subject matter covered, it is highly probable that I have got some things wrong. Life is tough.

- TRADEMARKS: DB2/2 is a trade mark of IBM Corporation.

- CONTENTS: This document describes how to use DB2/2 SQL to answer common business questions. Most, but not all, of the examples given here are equally applicable to all flavours of SQL, some are specific to DB2/2 SQL. Basic SQL knowledge and some prior experience are assumed.

- HARDWARE & SOFTWARE USED: This course was written on a Gateway 2000 66Mhz 486 PC. An HP IIID printer was used to do the printing. Microsoft Word for Windows version 2.0 was used to write the document. Windows True Type was used to create the fonts.

  The course SQL was tested using a new DB2/2 product written by Steve Call. What this cute little piece of work does is run lots of SQL statements in batch collecting all kinds of good information along the way. To find out more, contact Steve via his Compuserve id: 71042,1507.

COPYRIGHT

- This course is the property of Graeme Birchall. Distribution does not imply publication. This particular copy was printed on 16 January, 1994, 1:15 PM. Feel free to make copies of this course, see the next page for details.


```
+-------------------------------------------+
|Version:     DB2.22.01.11-FU               |
|Date:        Jan-94                        |
|Author:      Graeme Birchall               |
|Address:     35 River Drive South #1214    |
|             Jersey City NJ 07310          |
|Ph/Fax:      (201)-963-0071                |
|Compuserve   73540,1566                    |
+-------------------------------------------+
```

Table of Contents

DISTRIBUTION

- USING THIS COURSE: This course is FREEWARE. It is for FREE public distribution. You may make and distribute as many copies as you so please (for free, you must not charge). You may also teach it "in house", or at a free public seminar without charge. You may not teach this course to a paying audience without prior permission. Nor may you include parts of this course in your own documents without prior permission.

- COMMENTS WELCOME: Any suggestions or comments that you may have about this material would be most appreciated. Also, let me know if there is something that you would like added.

- OTHER VERSIONS: An MVS/DB2/2 version of the course is also available (for free), just ask.

- A4 VERSION: This course is distributed as a Postscript print file for use printing with US letter size paper. An A4 (European) page size version is also available. If you can't find it on your local BBS, contact the author for a copy.

Table of Contents

Introduction to SQL 7

Advanced SQL 43

Fun with SQL 81

Notes 119

Index 145

Introduction to SQL

Table of Contents

NOTES

- SUBJECTS COVERED: The chapter will provide a simple introduction to DB2/2 SQL. It will also discuss the use of scalar functions, column functions, date fields, basic date addition, and various other kinds of DB2/2 expression.

- EXERCISE: There will be an exercise to do at the end of this chapter.

Table of Contents

Sample Database	10
Syntax Diagram Conventions	12
SQL Components	14
SQL Expressions	16
SQL Operators	18
SQL - Precedence Rules	20
Date/Time Fields - Issues	22
Date/Time Arithmetic Rules	24
Scalar Functions - Date/Time	28
Scalar Functions - Other	30
Column Functions	34
Functions - Data Truncation & Alteration	36
Varchar & Functions - Notes	38
Exercise 1	40

Sample Database

NOTES

- SAMPLE APPLICATION: The sample tables describe a simple inventory application. There is a CITY table (with one row per city), a WAREHOUSE table (with one row per city/warehouse), a PRODUCT table (with one row per product), and junction table (WP: with one row per city/warehouse/product). The SALE table records sales of products. Note that not every product has had a sale and those that have may have had multiple sales. Each table is suitably indexed.

- PRODUCT NUMBER VALUES: Note that the Product number values, unlike the other keys values, are not in simple continual ascending sequence. Instead the values go up in sequence for thirty numbers and then skip seventy numbers before going up in sequence for another thirty values, etc. The following table illustrates how they work. The reason why this was done was to allow me to fool the DB2/2 optimizer into choosing sub-optimal access paths (for illustration purposes).

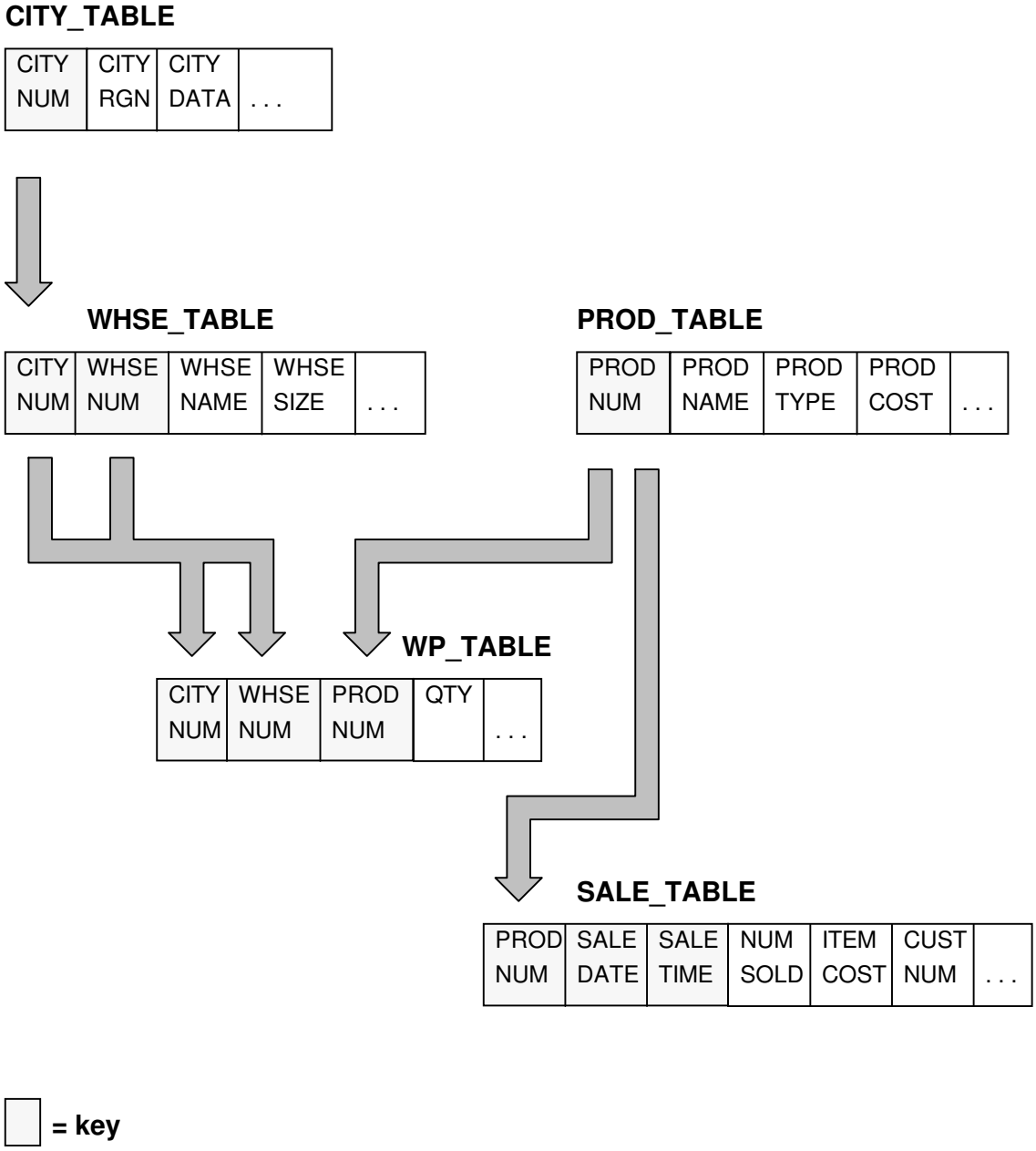
PRODUCT NUMBER VALUES

2000,	2001,	2002,	2003, 2004, 2005, ... 2026, 2027, 2028, 2029
2100,	2101,	2102,	2103, 2104, 2105, ... 2126, 2127, 2128, 2129
2200,	2201,	2202,	2203, 2204, 2205, ... 2226, 2227, 2228, 2229
...			
2900,	2901,	2902,	2903, 2904, 2905, ... 2926, 2927, 2928, 2929

INDEXES

TABLE-NAME	INDEX-NAME	INDEX-TYPE	FULLKEYCARD	INDEX-COLS	COLUMN CARD
CITY_TABLE	ICITYNUM	Unique	10	CITY_NUM	10
	ICITYRGN	Non-unique	2	CITY_RGN	2
WHSE_TABLE	IWHSENUM	Unique	40	CITY_NUM	10
				WHSE_NUM	4
	IWHSENME	Unique	40	WHSE_NAME	40
PROD_TABLE	IPRODNUM	Unique	300	PROD_NUM	300
	IPRODNME	Unique	300	PROD_NAME	300
	IPRODTYP	Non-unique	30	PROD_TYPE	30
WP_TABLE	IWPWHSE	Unique	12,000	CITY_NUM	10
				WHSE_NUM	4
				PROD_NUM	300
	IWPPROD	Unique	12,000	PROD_NUM	300
				CITY_NUM	10
				WHSE_NUM	4
SALE_TABLE	ISALENUM	Unique	2,400	PROD_NUM	240
				SALE_DATE	30
				SALE_TIME	80
	ICUSTNUM	Non-unique	21	CUST_NUM	21

Sample Database



Syntax Diagram Conventions

NOTES

- INTRODUCTION: This course uses RAILROAD diagrams to describe the DB2/2 SQL statements. This page discusses the conventions used in this course.

- START: The start of the series of statements.

- CONTINUE: A marker to indicate that the statement text continues below.

- RESUME: A marker that relates back to the prior CONTINUE point.

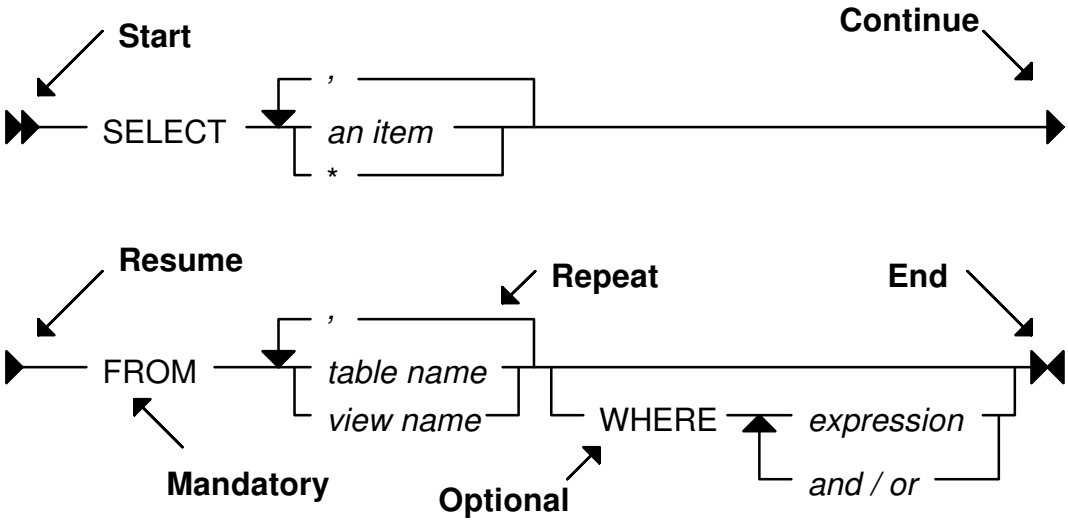
- MANDATORY: A statement that is in the main path and which must be used every time.

- REPEAT: A indicator that the path just used can be repeated.

- OPTIONAL: Those statements which do not have to be used in a request.

- END: The end of the series of statements.

Syntax Diagram Conventions



UPPER-CASE **=> SQL Key word**

Italic **=> Explained Elsewhere**

Underlined **=> Default Option**

SQL Components

NOTES

- SQL COMPONENTS: In a SQL statement (as opposed to a DML or DDL statement) one must select 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. Where conditions are optional.

- SELECT INDIVIDUAL ITEMS: The following items can be included in the select list:
  - A column name e.g: PROD_NUM.
  - A literal value e.g: 'ABC'.
  - An expression e.g: 12 * 9.
  - A function result e.g: MAX(PROD_NUM).

- CORRELATION NAME: A correlation name is defined in a FROM clause and relates to the preceding object name. In same cases it is used to provide a short form of the related object name. In other cases it is needed in order to uniquely identify logical tables when a physical table is referred to twice in the same query.

- SELECTING SETS OF COLUMNS: To select all of the available columns use "*", to select all of the available columns in a particular table use "A.*" where "A" is the table reference (not the name).

ANSWERS, SQL ON NEXT PAGE

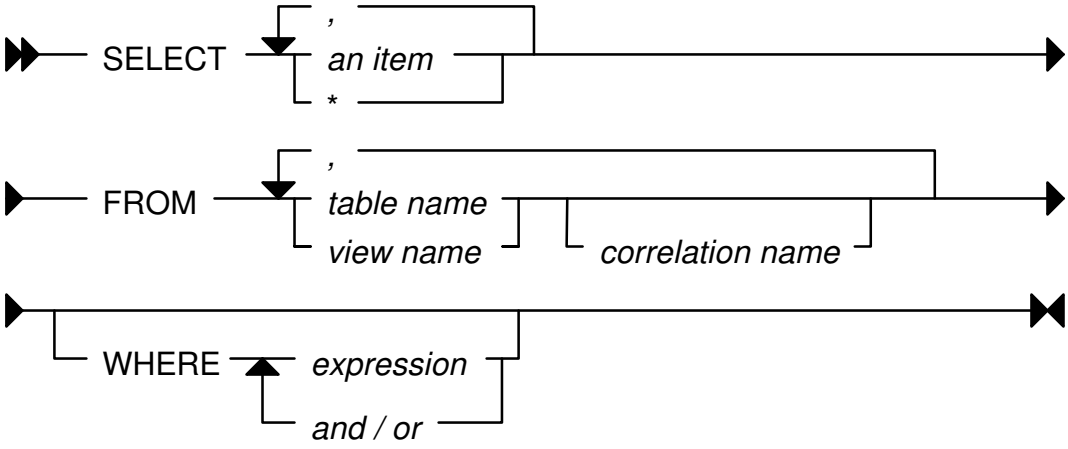
1ST
===

```
PROD_NUM  PROD_NAME  PROD_TYPE  PROD_COST  PROD_PADDING
--------  ---------  ---------  ---------  ------------
    2001  0002       J               2000  PROD
```

2ND
===

```
PROD  PROD  PROD              SALE      NUM   ITEM  CUST  SALE
NUM   TYPE  NUM   SALE-DATE   TIME      SOLD  COST  NUM   PADDING
----  ----  ----  ----------  --------  ----  ----  ----  -------  --------
2001  J     2001  1994-01-01  03.01.00    90   .00  9100  SALE     21:56:54
```

SQL Components



Examples

```
SELECT *
FROM   PROD_TABLE
WHERE  PROD_NUM   =   2001
  AND  PROD_TYPE  >   'A';
```

```
SELECT A.PROD_NUM
      ,A.PROD_TYPE
      ,B.*
      ,CURRENT TIME
FROM   PROD_TABLE A
      ,SALE_TABLE B
WHERE  A.PROD_NUM   =   2001
  AND  B.PROD_NUM   =   A.PROD_NUM
  AND  B.SALE_TIME  =  '03.01.00';
```

SQL Expressions

NOTES

- EXPRESSIONS: Any SQL statement is made up of one or more expressions. An expression is made of one or more components (see next page) and can include nested expressions.

- LABELED DURATION'S: A labeled duration is a represented followed by any one of seven keywords: YEARS, MONTHS, DAYS, HOURS, MINUTES, SECONDS, MICROSECONDS. Each keyword can be expressed is both the plural and singular sense.

- COLUMN HEADINGS FOR OUTPUT: When the SQLDA returned by DB2/2 is used. as it is here, to make the column headings, it is possible that many of the columns selected will have no name. Only those columns which are not modified by either an expression, or a function, will have a column heading.

SPECIAL REGISTERS

- USER: The authorization id of the user.

- CURRENT DATE: The current date.

- CURRENT TIME: The current time.

- CURRENT TIMESTAMP: The current timestamp.

- CURRENT SERVER: The name of the current server.

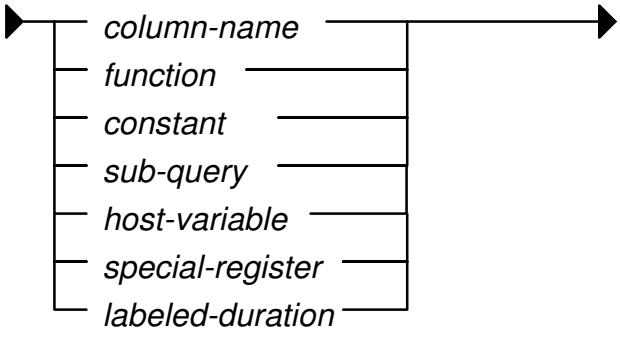
SELECTING QUOTES

- Two quotes in a line represent a string of one quote. Multiple pairs of quotes represent multiple one quote strings. A single quote, or an additional odd-numbered quote in a string, define the start and/or end of a text string.

```
SELECT 'abc'
      ,'abc''def'
      ,'abc'''
      ,'abc'''''
      ,'abc'''''''
      ,'''abc'''
FROM   CITY_TABLE
WHERE  CITY_NUM = 1;


---   -------   ----   -----   ------   -----
abc   abc'def   abc'   abc''   abc'''   'abc'
```

SQL Expressions



Example

```
SELECT PROD_NUM
      ,SUBSTR(PROD_NAME,1,3)
      ,'ABC'
      ,HOUR(CURRENT TIMESTAMP)
      ,CURRENT SERVER
FROM   PROD_TABLE
WHERE  PROD_NUM  =   2001
  AND  PROD_NAME >   ' ';
```

```
PROD_NUM
--------  ---  ---  --  --------
    2001  000  ABC  21  TEST023A
```

SQL Operators

NOTES

- BETWEEN: Note that 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, BETWEEN 12 AND 10 never will.

- LIKE: The like statement is used to do partial checks on character strings. The percent and underscore characters have special meanings in this statement, the first means skip a string of any length (including zero), the latter means skip one byte. For example:
  LIKE 'AB_D%' Will find 'ABCD', and 'ABCDE', but not 'ABD', or 'ABCCD'.
  LIKE '_X' Will find 'XX' and 'DX', but not 'X', or 'ABX', or 'AXB'.
  LIKE '%X' Will find 'AX', 'X' and 'AAX', but not 'XA'.

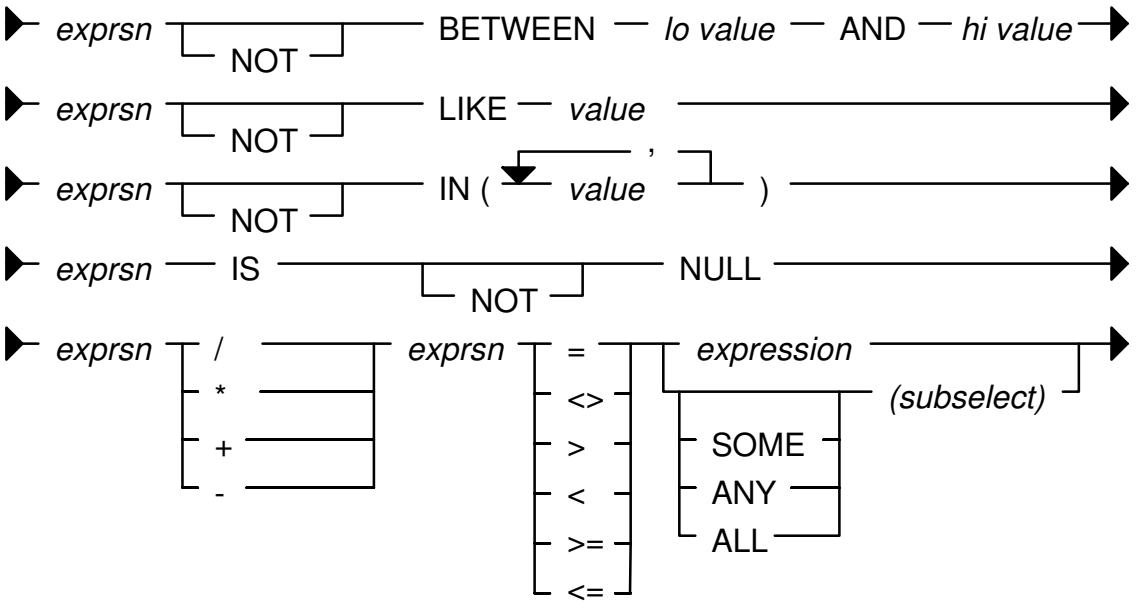
- PERCENT IN SEARCH: Use the TRANSLATE function to check for percent signs and/or underscores in the search string. See page 32 for details.

- IN: Check if the expression is equal to any one of the values in the IN list.

- IS/NOT NULL: One does not use the equal condition in a NULL check, instead one looks to see if a value IS, or IS NOT, null.

- PREDICATE: To compare two values use the following:
  = Equal
  <> Not equal
  < Less than
  > Greater than
  <= Less than or equal to
  >= Greater than or equal to.

SQL Operators



Example

```
SELECT *
FROM   PROD_TABLE
WHERE  PROD_NUM         =  2000
  AND  PROD_NUM        <>  9999
  AND  PROD_NAME    LIKE  '_0%'
  AND  PROD_TYPE      IN ('J','K','L')
  AND  PROD_NUM  BETWEEN  2000 * 1 AND 2000 * 1.1;
```

```
PROD_NUM  PROD_NAME  PROD_TYPE  PROD_COST  PROD_PADDING
--------  ---------  ---------  ---------  ------------
    2000  0001       J               2000  ABC%DEF
```

SQL - Precedence Rules

OPERATIONS PRECEDENCE

- NOTE: DB2/2 uses standard mathematical rules of precedence when doing arithmetic operations.

- WHAT'S FIRST: 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. All encompassing parentheses (e.g: AVG(1 * 7)) are done last.

PREDICATES PRECEDENCE

- WHAT'S FIRST: Parentheses are done first, then AND predicates, then OR predicates.

- COMMON MISTAKE: A common mistake made when using OR statements involves leaving out brackets when the OR should only be between two predicates and not between one and all the rest. For example, the first of the following two SQL statements will do a Cartesian Product because the last two predicates are not enclosed in brackets.

```
SELECT  COUNT(*)                --- Ans: 48
FROM    CITY_TABLE A
       ,WHSE_TABLE B
WHERE   A.CITY_NUM = B.CITY_NUM
  AND   A.CITY_NUM < 2
   OR   A.CITY_NUM > 8;

SELECT  COUNT(*)                --- Ans: 12
FROM    CITY_TABLE A
       ,WHSE_TABLE B
WHERE   A.CITY_NUM = B.CITY_NUM
  AND  (A.CITY_NUM < 2
   OR   A.CITY_NUM > 8);
```

SQL - Precedence Rules

Operations Precedence

1st. Parentheses
2nd. Prefix operators
3rd. Multiplication, & Division
4th. Addition & Subtraction
5th. Left to Right
6th. All encompassing functions

Predicates Precedence

1st. Parentheses
2nd. AND
3rd. OR

Example

```
SELECT  100  *  5 + 2
       ,100  * (5 + 2)
FROM    SALE_TABLE
WHERE   PROD_NUM  = 2001
  AND   SALE_TIME < '04.00.00'
  AND
       ('A' = 'A'    --- AND before OR
   OR   'A' = 'C'
  AND   'A' = 'D')
  AND
      (('A' = 'A'    --- OR before AND
   OR   'A' = 'C')
  AND   'D' = 'D');


-----   -----
  502     700
```

Date/Time Fields - Issues

NOTES

- INTRODUCTION: Date/time fields are stored internally by DB2/2 as unsigned packed decimal numbers. These are illustrated at the top of the next page. If one intends to do arithmetic on a date/time field then one must use one of four flavours of operator:
  - One can use what is called a labeled duration, (eg: 10 years, 5 days, 6 minutes).
  - One can use an appropriate date/time field (e.g: add a date value to a timestamp).
  - One can use a character string which represents a date/time value, (eg: '1994-03-24').
  - One can use a special length and scale of decimal field. These are called date/time duration's.

DATE/TIME INTERNAL FORMATS

- DATE: A date field is stored in a format equivalent to unsigned 4 byte packed decimal.

- TIME: A time field is stored in a format equivalent to unsigned 3 byte packed decimal.

- TIMESTAMP: A timestamp field is stored in a format equivalent to unsigned 10 byte packed decimal.

DATE/TIME EXTERNAL FORMATS

- DATE: YYYY-MM-DD (10 bytes).

- TIME: HH.MM.SS (8 bytes).

- TIMESTAMP: YYYY-MM-DD-HH.MM.SS.ZZZZZZ (26 bytes).

DATE/TIME DURATION'S

- LABELED DURATION'S: A labeled duration is a represented followed by any one of seven keywords: YEARS, MONTHS, DAYS, HOURS, MINUTES, SECONDS, MICROSECONDS. Each keyword can be expressed is both the plural and singular sense.

- DATE DURATION: A value that represents a number of years, months, and days in the format YYYYMMDD where YYYY is the year, MM is the month, and DD is the day. The field will have a format of DECIMAL(8,0).

- TIME DURATION: A value that represents a number of hours, minutes, and seconds in the format HHMMSS where HH is hour, MM is the minute, and SS is the second. The field will have a format of DECIMAL(6,0).

- TIMESTAMP DURATION: A value that represents a number of years, months, days, hours, minutes, seconds, and microseconds in the format YYYYXXDDHHMMSSZZZZZZ where YYYY is the year, XX is the month, DD is the day, HH is hour, MM is the minute, SS is the second, and ZZZZZZ is the microsecond. The field will have a format of DECIMAL(20,6).

Date/Time Fields - Issues

Internal Formats

- Date X'19941231'
- Time X'125633'
- Timestamp X'19941231125633000000'

Date/Time Operands

- Labeled Duration: eg: 5 days, 2 months, 1 hour
- Date Duration: Decimal(8,0) number
- Time Duration: Decimal(6,0) number
- Timestamp Duration: Decimal(20,6) number

Example

```
SELECT SALE_DATE
      ,SALE_TIME
      ,DECIMAL(19940101,8,0)
      ,DECIMAL(030100,6,0)
FROM   SALE_TABLE
WHERE  PROD_NUM    = 2001
  AND  SALE_TIME   = '03.01.00';
```

```
SALE_DATE   SALE_TIME
----------  --------   ---------   ------
1994-01-01  03.01.00   19940101.   30100.
```

Date/Time Arithmetic Rules

NOTES

- WARNING: Only certain types of field types can be used in date/time arithmetic.
- ADDITION: In general, a date/time field, or a string representation of a date/time field can only be added to a related duration field, or an appropriate labeled duration.
- SUBTRACTION: In general, a date/time field, or a string representation of a date/time field can only be subtracted by a related duration field, string representation, or an appropriate labeled duration.
- MULTIPLICATION & DIVISION: These types of manipulation on date/time fields are not supported.

OUTPUT FORMATS

		1st Operand		
2nd Operand	Function	Date	Time	Timestamp
Labeled Duration	Add	Date	Time	Timestamp
	Subtract	Date	Time	Timestamp
Date	Subtract	Date Duration		
Date Duration	Add	Date		Timestamp
	Subtract	Date		Timestamp
Time	Subtract		Time Duration	
Time Duration	Add		Time	Timestamp
	Subtract		Time	Timestamp
T.stamp	Subtract			Timestamp Duration
T.stamp Duration	Add			Timestamp
	Subtract			Timestamp

SAMPLE SQL/OUTPUT

```
SELECT SALE_DATE
      ,SALE_DATE   + DECIMAL(00010112,8,0)
      ,SALE_DATE   - DECIMAL(00010112,8,0)
      ,SALE_DATE   - DATE('0001-01-12')
      ,SALE_DATE   - '0001-01-12'
      ,SALE_TIME
      ,SALE_TIME   - TIME('00.01.22')
      ,SALE_TIME   - '00.01.22'
      ,SALE_TIME   - 3 HOURS
      ,SALE_TIME   - 2 HOURS + 2 MINUTES
FROM   SALE_TABLE
WHERE  PROD_NUM    = 2001
  AND  SALE_TIME   = '03.01.00';

SALE_DATE
----------  ----------  ----------  ---------  ---------
1994-01-01  1995-02-13  1992-11-20  19921120.  19921120.

SALE_TIME
---------   --------    --------    --------    --------
03.01.00      25938.      25938.    00.01.00    01.03.00
```

Date/Time Arithmetic Rules

Addition

2nd Operand	1st Operand		
	Date	Time	Timestamp
Date			
Date Duration	OK		OK
Time			
Time Duration		OK	OK
Timestamp			
Timestamp Duration			OK
Labeled Duration	Yr, Mth, Day	Hr, Min, Sec	All

Subtraction

2nd Operand	1st Operand		
	Date	Time	Timestamp
Date	OK		
Date Duration	OK		OK
Date String	OK		
Time		OK	
Time Duration		OK	OK
Time String		OK	
Timestamp			OK
Timestamp Duration			OK
Timestamp String			OK
Labeled Duration	Yr, Mth, Day	Hr, Min, Sec	All

Output Formats

- See notes

Date/Time Arithmetic Rules

NOTES

- BOLD LINES IN TOP EXAMPLE: The lines in "bold" in the example at the top example on the next page illustrate SQL statements which will NOT work. The first line tries to add two dates (instead of a date and a date duration). The second creates a temporary value which is neither hours nor minutes.

- ADDING MONTHS: When a number months are added to a Date or Timestamp field the actual date that results is a function of the current day in the month of the original and final values. If the result month is shorter than the original month, and the original day was "over the end" of the new month then appropriate truncation will occur.

- FIRST AND LAST DAYS: DB2/2 treats all days in the month as equal. In particular, it does not do anything special with either the last or the first day of the month.

SELECTING THE LAST DAY OF THE MONTH

The best way to get a sequence of values which always returns the last day of the month is to start of getting the first of the next month and then subtract one day.

```
SELECT DATE('1994-01-01')
      ,DATE('1994-01-01') + 1 MONTH  - 1 DAY
      ,DATE('1994-01-01') + 2 MONTHS - 1 DAY
      ,DATE('1994-01-01') + 3 MONTHS - 1 DAY
      ,DATE('1994-01-01') + 4 MONTHS - 1 DAY
      ,DATE('1994-01-01') + 5 MONTHS - 1 DAY
FROM   SALE_TABLE
WHERE  PROD_NUM    = 2001
  AND  SALE_TIME   = '03.01.00';
```

```
 ORIGINAL    +1 Month    +2 Months   +3 Months   +4 Months   +5 Months
----------  ----------  ----------  ----------  ----------  ----------
1994-01-01  1994-01-31  1994-02-28  1994-03-31  1994-04-30  1994-05-31
```

Date/Time Arithmetic Rules

```
SELECT SALE_DATE
--    ,SALE_DATE   + '0001-01-12'
      ,SALE_DATE   + DECIMAL(00010112,8,0)
      ,SALE_DATE   - '0001-01-12'
      ,SALE_TIME   - 3 HOURS
      ,SALE_TIME   - 2 HOURS + 2 MINUTES
--    ,SALE_TIME   -(2 HOURS + 2 MINUTES)
FROM   SALE_TABLE
WHERE  PROD_NUM    = 2001
  AND  SALE_TIME   = '03.01.00';
```

```
SALE_DATE
----------  ----------  --------   --------  --------
1994-01-01  1995-02-13  19921120.  00.01.00  01.03.00
```

Adding Months

```
SELECT DATE('1994-01-31')
      ,DATE('1994-01-31') + 1 MONTH
      ,DATE('1994-01-31') + 3 MONTHS
      ,DATE('1994-01-27')
      ,DATE('1994-01-27') + 1 MONTH
      ,DATE('1994-01-27') + 3 MONTHS
      ,DATE('1994-02-28')
      ,DATE('1994-02-28') + 1 MONTH
      ,DATE('1994-02-28') + 3 MONTHS
FROM   SALE_TABLE
WHERE  PROD_NUM    = 2001
  AND  SALE_TIME   = '03.01.00';
```

```
ORIGINAL    + 1 MONTH   + 3 MONTHS
----------  ----------  ----------
1994-01-31  1994-02-28  1994-04-30
1994-01-27  1994-02-27  1994-04-27
1994-02-28  1994-03-28  1994-05-28
```

Scalar Functions - Date/Time

NOTES

- CHAR: Returns a string representation of a date, time, timestamp or decimal number.

- DATE: Returns a date. The input can be a timestamp, date, the number of days since January 1st, 0001, a character representation of a date in the format YYYYNNN where YYYY is the year and NNN is the Julian day number, or a character representation of a date (eg: '1994-12-31').

- DAY: If the argument is a date or timestamp returns the day of the month (i.e: from 1 to 31). If the argument is a timestamp duration or date duration the result is any value between -99 and 99.

- DAYS: The number of days since January 1st, 0001. This function assumes that the Gregorian calendar has been in effect since that date, which is not true. Consequently, the DAYS function should not be used to compare current dates with historical dates (e.g: how many days since Shakespeare's birthday?) which are before the late-Renaissance in most of Europe, before 1752 in English speaking countries, or before 1919 in Russia.

- HOUR: The hour part of a value.

- MICROSECOND: The microsecond part of a value.

- MINUTE: The minute part of a value.

- MONTH: The month part of a value.

- SECOND: The second part of a value.

- TIME: The time (i.e: hour, minute, second) part of a value.

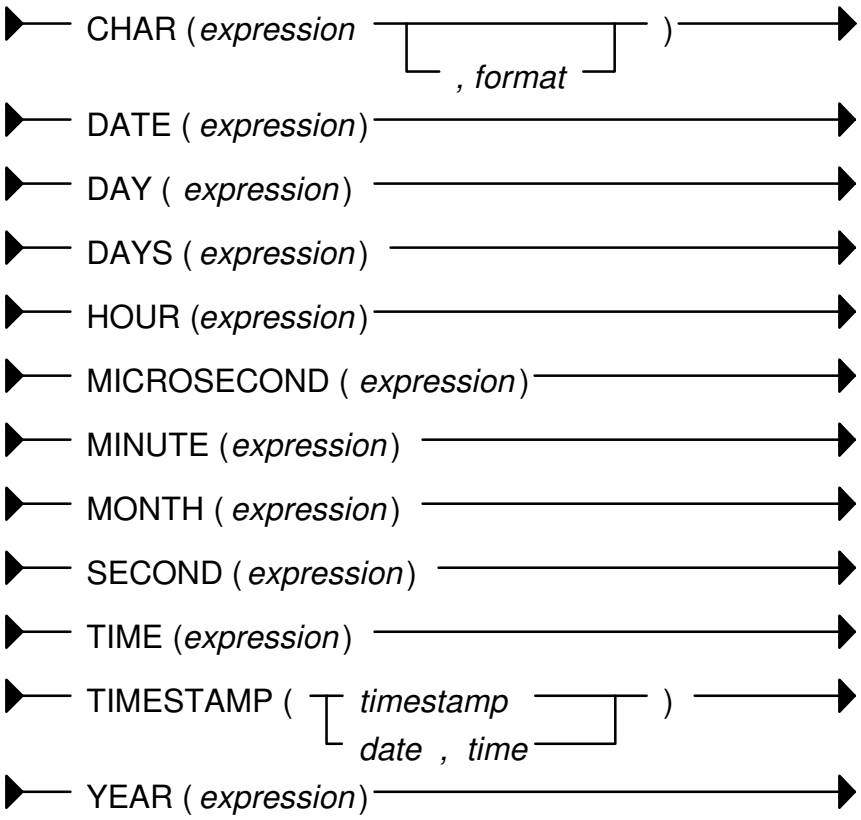
- TIMESTAMP: The timestamp representation of a timestamp value, or a combined date and time value.

- YEAR: The year part of a value.

CHAR FUNCTION EXAMPLES

```
SELECT CHAR(DATE('1994-12-31'))              ===>   1994-12-31
      ,CHAR(DATE('1994-12-31'),ISO)          ===>   1994-12-31
      ,CHAR(DATE('1994-12-31'),USA)          ===>   12/31/1994
      ,CHAR(DATE('1994-12-31'),EUR)          ===>   31.12.1994
      ,CHAR(DATE('1994-12-31'),JIS)          ===>   1994-12-31
      ,CHAR(TIME('12.56.33'))                ===>   12.56.33
      ,CHAR(TIME('12.56.33'),ISO)            ===>   12.56.33
      ,CHAR(TIME('12.56.33'),USA)            ===>   12:56 PM
      ,CHAR(TIME('12.56.33'),EUR)            ===>   12.56.33
      ,CHAR(TIME('12.56.33'),JIS)            ===>   12:56:33
FROM   SALE_TABLE
WHERE  PROD_NUM  = 2001
  AND  SALE_TIME = '03.01.00';
```

Scalar Functions - Date/Time



Example

```
SELECT TIMESTAMP(SALE_DATE,SALE_TIME)
      ,MONTH(SALE_DATE)
      ,DAYS(SALE_DATE)
      ,DAY(SALE_DATE)
      ,HOUR(SALE_TIME)
FROM   SALE_TABLE
WHERE  PROD_NUM    = 2001
  AND  SALE_TIME   = '03.01.00';


--------------------------  -  ------  -  -
1994-01-01-03.01.00.000000  1  727929  1  3
```

Scalar Functions - Other

NOTES

- DECIMAL: Returns a decimal representation of a numeric value. The second argument is the precision (from 1 to 31) and the third the scale (from 0 to the value of the precision).

- FLOAT: Returns a floating point representation of a number.

- INTEGER: Returns an integer representation of a number. Note, there is no SMALLINT function.

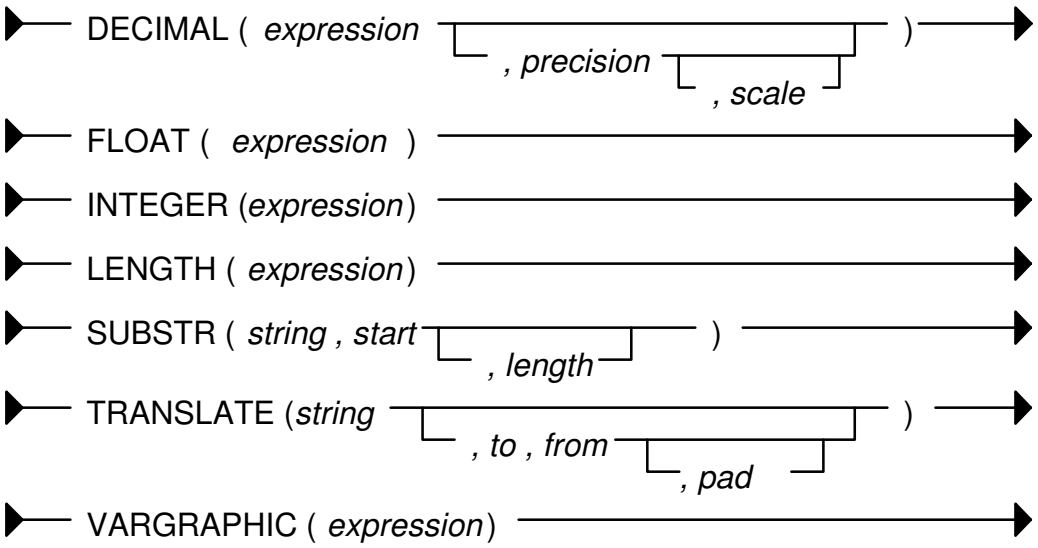
- LENGTH: Returns the length of any value (including numbers and date/time fields).

- SUBSTR: Returns the sub-string of any string.

- TRANSLATE: Converts individual values in either a character, or graphic string, from one value to another. Can also convert lower case data to upper case. See page 32.

- VARGRAPHIC: Returns a variable graphic representation of any string. Use of the VARGRAPHIC function requires that the underlying database have an appropriate code table.

Scalar Functions - Other



Example

```
SELECT DECIMAL(123.456,5,2)
      ,FLOAT(123.456)
      ,INTEGER(123.456)
      ,LENGTH('1234')
      ,SUBSTR('123456',1,2)
      ,SUBSTR('123456',3)
      ,VARGRAPHIC('12')
FROM   SALE_TABLE
WHERE  PROD_NUM    = 2001
  AND  SALE_TIME   = '03.01.00';
```

```
------  ----------------------  ---  -  --  ----  -----
123.45  +0.1234560000000000E+03  123  4  12  3456  .â1â2
```

Scalar Functions - Translate

NOTES - 1ST SQL

- INTRODUCTION: The TRANSLATE function converts individual values in either a character, or graphic string, from one value to another. It can also convert lower case data to upper case.

- STRING ONLY: Converts input string from lower case to upper case.

- TO VALUE: A value must be provided if this option is used. A zero length string is converted to a blank. If there are less FROM than TO values, the trailing TO values are ignored.

- FROM VALUE: A value must be provided if this option is used. A zero length string is converted to a blank. If there are more FROM than TO values, and no PAD value is provided, spaces are assumed for the missing values.

- PAD VALUE: If there are more FROM than TO values, this acts as an substitute, else it is ignored.

- 1ST SQL: The 1st statement on the next page does the following:
  1. Select the text string 'abcd'.
  2. Convert the string to upper case.
  3. Change all 'a' items to 'A'.
  4. Change all 'a' items to blank. Note blank is the default if a null "to" string is provided.
  5. Change all 'ab' items to 'A ' (i.e: 'A' then blank).
  6. Change all 'a' items to 'A', the 'B' is ignored.
  7. Change all 'ab' items to 'A ' (i.e: 'A' then blank).
  8. Change all 'ab' items to 'Az'.

NOTES - 2ND SQL

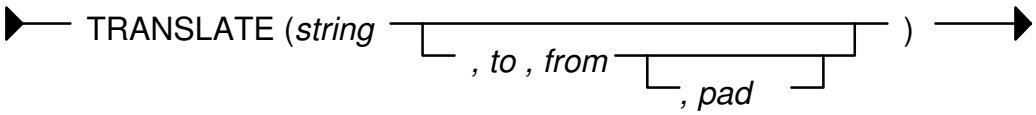
- LIKE PREDICATES: Use the TRANSLATE function to look for strings containing underscores, or percent signs, with the LIKE predicate. To do this, convert the "%" or "_" to some other character (e.g: "$" and "*") and - at the same time - convert the target characters to something else which is not in the LIKE string (e.g: "@" and "!").

  For example, if one is looking for "%" values in the string "AB%C%$", one can TRANSLATE the "%" values to "$" and - at the same time - convert any original "$" values to "*". One can then use the LIKE to search using LIKE '%$%'.

- 2ND SQL LIST: The 2nd statement on the next page does the following:
  1. Select the text string 'AB%$C%$'.
  2. Translate all "%" to "$".
  3. Translate all "%" to "$" and "$" to "*" - at the same time!
  4. Translate all "%" to "$" - then - translate all "$" to "*".
  5. Translate all "$" to "*" - then - translate all "%" to "$".

  The following LIKE checks are done:
  1. Find all strings beginning "AB".
  2. Find all strings containing a "%".
  3. Find all strings beginning with a "%".
  4. Find all strings containing "%%%".
  5. Find all strings containing "A%C".

Scalar Functions - Translate




```
SELECT   'abcd'
        ,TRANSLATE('abcd')
        ,TRANSLATE('abcd','A','a')
        ,TRANSLATE('abcd','','a')
        ,TRANSLATE('abcd','A','A')
        ,TRANSLATE('abcd','A','ab')
        ,TRANSLATE('abcd','AB','a')
        ,TRANSLATE('abcd','A','ab',' ')
        ,TRANSLATE('abcd','A','ab','z')
FROM     CITY_TABLE
WHERE    CITY_NUM    = 1;
```

```
----  ----  ----  ----  ----  ----  ----  ----  ----
abcd  ABCD  Abcd   bcd  abcd  A cd  Abcd  A cd  Azcd
```

```
SELECT   'AB%$C%$'
        ,TRANSLATE('AB%$C%$','$','%')
        ,TRANSLATE('AB%$C%$','*','$')
        ,TRANSLATE('AB%$C%$','$*','%$')
        ,TRANSLATE(TRANSLATE('AB%$C%$','$','%'),'*','$')
        ,TRANSLATE(TRANSLATE('AB%$C%$','*','$'),'$','%')
FROM     CITY_TABLE
WHERE    CITY_NUM    = 1
  AND    'ABC%$D'                        LIKE 'AB%'
  AND    TRANSLATE('ABC%$D','$*','%$') LIKE '%$%'
  AND    TRANSLATE('%ABC$D','$*','%$') LIKE '$%'
  AND    TRANSLATE('A%%%$D','$*','%$') LIKE '%$$$%'
  AND    TRANSLATE('A%C%$D','$*','%$') LIKE 'A$C%';
```

```
-------   -------   -------    -------   -------   -------
AB%$C%$   AB$$C$$   AB%*C%*    AB$*C$*   AB**C**   AB$*C$*
```

Column Functions

NOTES

- AVG: Get the average value in a set of rows. The column(s) must be numeric. ALL is the default. If DISTINCT is used duplicate values are ignored.

- COUNT: Get the number of rows or DISTINCT values in a set of rows. The resultant data type is an INTEGER. If the DISTINCT option is used both duplicate and null values are eliminated.

- MAX: Get the maximum value of a set of values. The use of the DISTINCT clause has no affect on the answer.

- MIN: Get the minimum value of a set of values. The use of the DISTINCT clause has no affect on the answer.

- SUM: Get the SUM of a set of numeric values. If DISTINCT is used duplicate values are ignored. Null values are always ignored.

- DATE/TIME FIELDS: The AVG and SUM functions can NOT be used on date/time fields. This is because these fields are stored internally as decimal numbers and the average, or sum, of two such values may not turn out to be a valid date/time value: For example, the average of 19910101 (i.e: the date: 1991-01-01) and 19920101 is 19915101 which is not a valid date.

RELATED ITEMS

- HAVING: The HAVING clause is a form of WHERE condition which acts upon derived summary rows instead of individual rows. A "single row" predicate (eg: COL1 = 'ABC') would normally go in the WHERE part of the SQL statement. A "multi-row" predicate (eg: SUM(COL2) > 123) has to go in the HAVING part of the statement.

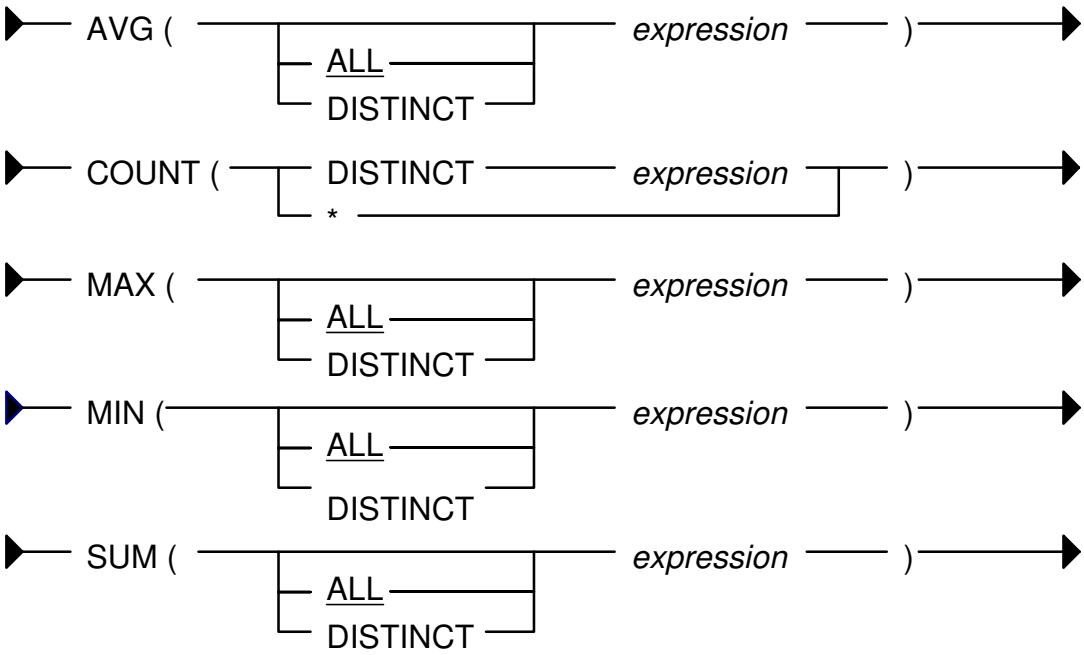
- DISTINCT: A distinct statement is used to eliminated duplicate values from the answer set.

- DISTINCT, ONE ONLY ALLOWED: Each SQL statement (UNION statements excepted) can only have a single distinct statement. This limit exists because DB2/2 resolves a distinct by reading all of the matching rows in distinct-key order, making a count as it goes.

- DISTINCT, WARNING: In DB2/2 it is possible to include an arithmetic expression in a distinct statement (e.g: SELECT DISTINCT(PROD_NUM * 2). In ANSI SQL this construct is not supported and it should not be used if the program is to be run on a non-IBM DBMS.

- ALL: Retrieve all rows, this is the default.

Column Functions



Example

```
SELECT AVG(NUM_SOLD)
      ,AVG(ALL NUM_SOLD)
      ,AVG(DISTINCT NUM_SOLD)
      ,COUNT(*)
      ,SUM(NUM_SOLD) /12
      ,SUM(NUM_SOLD  /12)
FROM   SALE_TABLE
WHERE  PROD_NUM < 2004
  AND (PROD_NUM < 2002
   OR  NUM_SOLD > 10)
HAVING COUNT(*) < 100;


--  --  --  --  ---  --
51  51  45  26  110  99
```

Functions - Data Truncation & Alteration

INTRODUCTION

- DB2/2 TRUNCATES, DOES NOT ROUND: When doing data truncation DB2/2 never rounds, it always truncates. Consequently, it is possible to get quite different answers depending upon how a question is expressed. For example, the two statements; Select SUM(TID/12) and SUM(TID)/12 from SYSTABLES will almost always differ in their results. The second one is probably what the user wanted.

SAMPLE SQL NOTES

- PLAN AVG: Select the average.
- AVG * 1: This will give the same answer as above.
- AVG * 1.1: This answer will presented as a decimal value.
- AVG * 1.0: This result is more accurate that the plain average.
- INTEGER(AVG * 1.0): This result is more accurate that the plain average.
- DECIMAL(AVG * 1.0): This result is more accurate that the plain average.

TRUNCATION - MORE EXAMPLES

```
SELECT DECIMAL(123.45,5,2)
      ,DECIMAL(123.456,5,2)
      ,INTEGER(123.678)
      ,-23.456
      ,DECIMAL(-23.456,5,2)
FROM   SALE_TABLE
WHERE  PROD_NUM    = 2001
  AND  SALE_TIME   = '03.01.00';


------  ------  ---  -------  ------
123.45  123.45  123  -23.456  -23.45
```

Example

```
SELECT AVG(PROD_NUM)
      ,AVG(PROD_NUM * 1)
      ,AVG(PROD_NUM * 1.0)
      ,INTEGER(AVG(PROD_NUM * 1.0))
      ,DECIMAL(AVG(PROD_NUM * 1.0),7,2)
FROM   SALE_TABLE
WHERE  SALE_TIME < '06.00.00'
  AND  PROD_NUM = (PROD_NUM/7) * 7;
```

```
ACTION                  RESULT
----------------------  ----------
AVG                           2467
AVG * 1                       2467
AVG * 1.0                     2467.70588235294117647
INTEGER,AVG * 1.0             2467
DECIMAL,AVG * 1.0             2467.70
```

Varchar & Functions - Notes

VARCHAR

- SELECT VARCHAR NEVER INDEX-ONLY: Selecting from a VARCHAR field which is part of an index key will always require access to the underlying table in order to get the data. This is because a VARCHAR field is stored in the index as fixed length (with padding zeros). DB2/2 has to go to the table to get the length of the field.

- CHECK VARCHAR MAY BE INDEX-ONLY: If the VARCHAR field is not selected, but only matched (in the predicate list), the query can be index-only, as long as it is the data and not the length that is being checked.

- USE SUBSTR TO MAKE FIXED LENGTH FIELD: The output of a SUBSTR is always fixed length. Use this function to convert either a VARCHAR field, or a text string (which is assumed to be VARCHAR), to fixed length.

- USE SUBSTR ON LONG VARCHAR: A long VARCHAR field is one longer than 254 bytes. Certain SQL restrictions apply to this field type: The LIKE is the only valid predicate, SUBSTR and LENGTH are the only valid functions. To use other predicate types (e.g: =, <, <=) on a long VARCHAR field, first apply the SUBSTR function to create an intermediate result that is shorter than 254 bytes.

FUNCTIONS

- FUNCTIONS/EXPRESSIONS WORK OUTWARD: Both functions and expressions are processed as the data is retrieved from DB2/2 and passed to the user's program, not the other way round. To illustrate this, the first of the two following INSERT statements does not work, whereas the second one does. Both attempt to do the same thing.

```
INSERT INTO FRED VALUES
(CURRENT DATE - 10 DAYS);
```

```
INSERT INTO FRED
SELECT CURRENT DATE -  10 DAYS
FROM   SYSIBM.SYSTABLES
WHERE  NAME = 'FRED';
```

- BEWARE SUM AND DISTINCT: It makes almost no sense to mix these two functions.

- BEWARE DISTINCT AND JOIN: If the join does a Cartesian product the SUM value returned will almost certainly be incorrect but there may be no other indication of a problem.

- SELECT AVG(DIVISION): This statement will give an answer but, for most users, it will not be what they intended. Instead use a SELECT(SUM) divided by a SELECT(SUM).

```
COL1   COL2          Divsision 1st            Addition 1st
====   ====          =============            ============
  10      1           10/ 1  =  10               10    1
 500     10          500/10  =  50              500   10
                                ===              ===  ===
                                 30 AVG          510   11  500/11 = 46 AVG
```

Varchar & Functions - Notes

Varchar

- Select VARCHAR is never index-only
- Check VARCHAR may be index-only
- Use SUBSTR to make VARCHAR or TEXT fixed length
- Use SUBSTR on LONG VARCHAR Fields

```
SELECT *
FROM   SYSIBM.SYSVIEWS
WHERE  SUBSTR(TEXT,1,3) < 'ABC';
```

Functions

- Functions/Expressions work outward
- Beware SUM and DISTINCT
- Beware SUM and JOIN
- AVG (Division) must be wrong

COL1	COL2
10	1
500	10

Exercise 1

Suggestions

- Work in a team of between 2 and 4, this will speed things up.
- Don't let one member of the team hog the computer for the whole time, rotate tasks.
- If you are having trouble with a question, skip it, and come back if you have time later.
- Use the index at the back to find the information that you need from the course.

DDL

See page 122 for a description of the tables used.

Q1

Select a COUNT of those rows in the SALE table where the CUST_NUM field is null. See page 34 for information on using the COUNT expression.

Q2

Select a COUNT of those rows in the SALE table where the DAY of the SALE_DATE is 1, 11, 21, or 31. Use the DAY function to get the day of the month (see page 28).

Q3

Select a COUNT of the DISTINCT WHSE_NUM values in the WP table where the CITY_NUM is less than 2. See page 34 for help.

Q4

Select the average of all PROD_NUM values in the SALE table. Also select the average of the distinct PROD_NUM values in the SALE table. Use the AVG function, see page 34 for assistance.

Q5

Select the following three fields from the SALE table:

- The SUM of the NUM_SOLD field, present in INTEGER format.
- The SUM of the NUM_SOLD field, present in DECIMAL format.
- The SUM of the NUM_SOLD field, present in FLOAT format.

Refer to page 30 for information on using the numeric functions.

Q6

Select a COUNT of those rows in SALE table where all of the following are true:

- The sale MONTH was December.
- The sale HOUR was 9 in the morning.
- The NUM_SOLD field is NOT NULL.

See page 28 for information on using date/time functions.

Q7

Select a COUNT of those rows in PROD table where all of the following are true:

- The PROD_TYPE field contains an "J" (use the LIKE predicate).
- The PROD_PADDING field STARTS with a "P" (use the LIKE predicate).
- The length of the PROD_PADDING field is more than 3 (use the LENGTH function).
- The second byte of the PROD_NAME field is less than "A" (use the SUBSTR function).

Refer to page 30 for information on using text functions.

Q8

Select the average CITY_NUM from the CITY table. Also select the same field, but this time multiply it by 1.0 before doing the AVG. Note the difference between the two results. Which is the more correct?

Exercise 1

Q9

Select the following fields from the SALE table where the PROD_NUM is 2001 and the HOUR of the SALE_TIME is 3 in the morning:

- The DATE of the sale in USA format (use the CHAR function).
- The DATE of the sale in European format.
- The day of the month when the sale occurred (use the DAY function).
- The day of the sale in decimal (2,0) format (use the DAY & DECIMAL functions).
- A timestamp field which combines the DATE and TIME fields (use the TIMESTAMP function).

Q10

Select the following fields from the SALE table where the PROD_NUM is 2001 and the HOUR of the SALE_TIME is 3 in the morning:

- The DATE of the sale.
- The DATE of the sale plus 10 years, 2 months, and 5 days.
- The DATE of the sale plus the date duration: 00100205.
- The number of days between the sale DATE and 1995-12-31 (use the DAYS function).
- A value which represents the DAY (of month) of the sale, times the HOUR of the sale time.

Q11

Select the following fields from the SALE table where the PROD_NUM is 2001 and the HOUR of the SALE_TIME is 3 in the morning:

- The DATE of the sale.
- The DATE of the sale in ISO format (use the CHAR function).
- The first seven characters of the sale DATE in ISO format (use the CHAR & SUBSTR functions).
- The customer number in floating point representation (use the FLOAT function).

Q12

Select a COUNT of those rows in the SALE table where all of the following are true:

- The PROD_NUM is between 2000 and 3000.
- The year is greater than 1991.
- The time is between 3am and 6am
- The second byte of the SALE_PADDING column is an "A" (use the LIKE predicate).

Q13

Select the following fields from the PROD table:

- The PROD_NUM times the PROD_COST.
- The first four bytes on the PROD_PADDING.
- The first four bytes on the PROD_PADDING, convert all "P"s to "D".
- The first four bytes on the PROD_PADDING, convert all "P"s to "D"s and "R"s to "Q".
- The first four bytes on the PROD_PADDING, convert all "P"s to "D"s and "D"s to "P".
- The first four bytes on the PROD_PADDING, convert all "P"s to "D"s and all else to "Z".

Get those rows where the following is true:

- The PROD_NUM is less than 2009.
- The PROD_TYPE is one of the following: "J", "K", "L", or "M".
- The PROD_NAME does not equal a single quote.
- The PROD_PADDING contains at least one blank.
- The PROD_PADDING does NOT contain a "%".

Refer to page 32 for assistance in using the TRANSLATE function.

Answers

Data: See page 135.
SQL: See page 138.

Advanced SQL

Table of Contents

NOTES

- SUBJECTS COVERED: The chapter will address the use of the GROUP BY, UNION clauses. Joins and sub-queries will also be addressed.

- EXERCISE: There will be an exercise to do at the end of this chapter.

Table of Contents

Sample Database	46
Group By	48
Joins, Unions, & Subqueries	50
Equi Join	52
Outer Join	56
Cartesian Product	58
Union & Union All	60
Except & Intersect	64
Sub-query - Syntax	68
Sub-query - Uncorrelated	70
Sub-query - Correlated	72
Subquery - Flavours	74
Exercise 2	78

Sample Database

NOTES

- SAMPLE APPLICATION: The sample tables describe a simple inventory application. There is a CITY table (with one row per city), a WAREHOUSE table (with one row per city/warehouse), a PRODUCT table (with one row per product), and junction table (WP: with one row per city/warehouse/product). The SALE table records sales of products. Note that not every product has had a sale and those that have may have had multiple sales. Each table is suitably indexed.

- PRODUCT NUMBER VALUES: Note that the Product number values, unlike the other keys values, are not in simple continual ascending sequence. Instead the values go up in sequence for thirty numbers and then skip seventy numbers before going up in sequence for another thirty values, etc. The following table illustrates how they work. The reason why this was done was to allow me to fool the DB2/2 optimizer into choosing sub-optimal access paths (for illustration purposes).

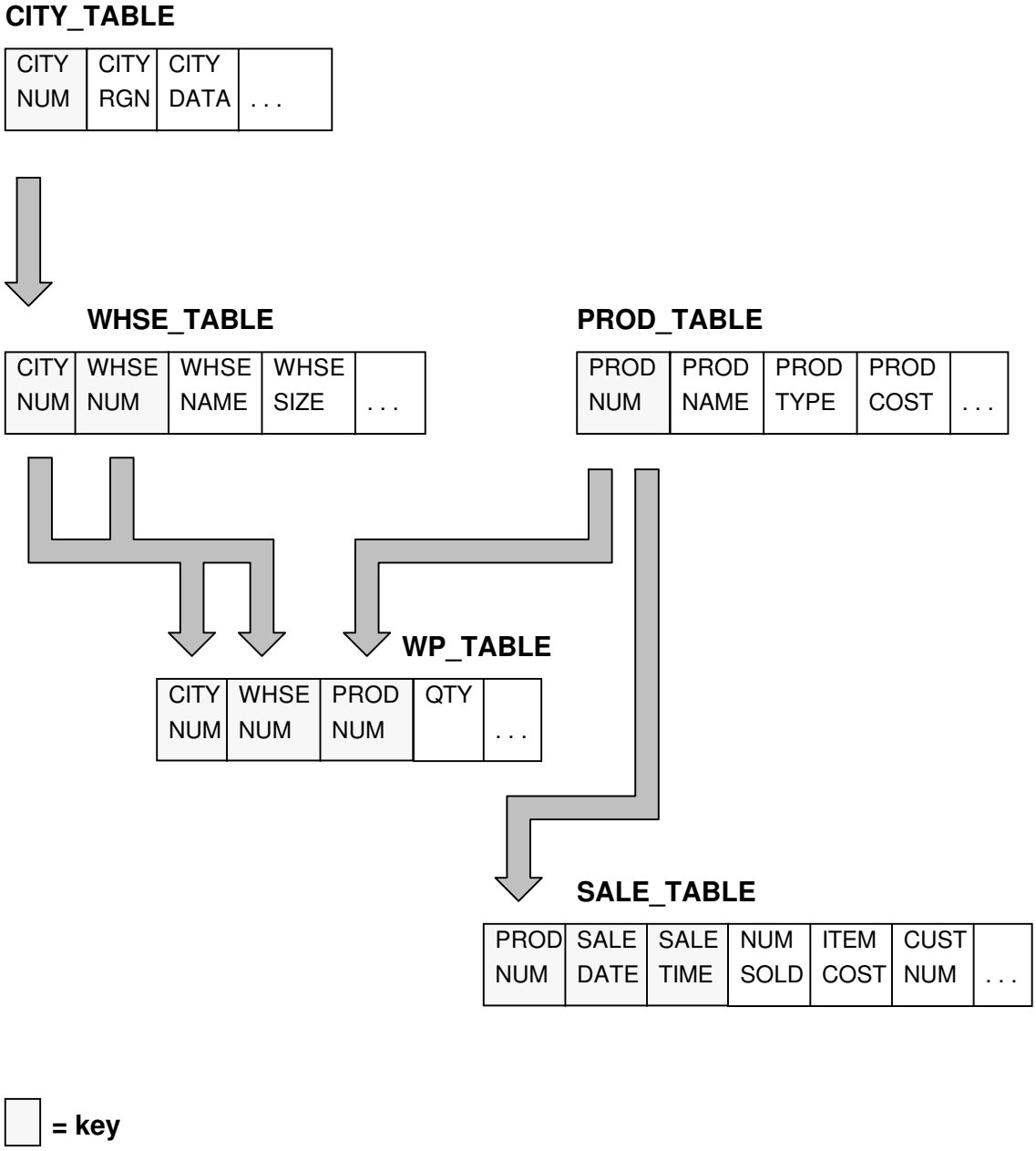
PRODUCT NUMBER VALUES

2000, 2001, 2002, 2003, 2004, 2005,	...	2026, 2027, 2028, 2029
2100, 2101, 2102, 2103, 2104, 2105,	...	2126, 2127, 2128, 2129
2200, 2201, 2202, 2203, 2204, 2205,	...	2226, 2227, 2228, 2229
...		
2900, 2901, 2902, 2903, 2904, 2905,	...	2926, 2927, 2928, 2929

INDEXES

TABLE-NAME	INDEX-NAME	INDEX-TYPE	FULLKEYCARD	INDEX-COLS	COLUMN CARD
CITY_TABLE	ICITYNUM	Unique	10	CITY_NUM	10
	ICITYRGN	Non-unique	2	CITY_RGN	2
WHSE_TABLE	IWHSENUM	Unique	40	CITY_NUM	10
				WHSE_NUM	4
	IWHSENME	Unique	40	WHSE_NAME	40
PROD_TABLE	IPRODNUM	Unique	300	PROD_NUM	300
	IPRODNME	Unique	300	PROD_NAME	300
	IPRODTYP	Non-unique	30	PROD_TYPE	30
WP_TABLE	IWPWHSE	Unique	12,000	CITY_NUM	10
				WHSE_NUM	4
				PROD_NUM	300
	IWPPROD	Unique	12,000	PROD_NUM	300
				CITY_NUM	10
				WHSE_NUM	4
SALE_TABLE	ISALENUM	Unique	2,400	PROD_NUM	240
				SALE_DATE	30
				SALE_TIME	80
	ICUSTNUM	Non-unique	21	CUST_NUM	21

Sample Database



Group By

NOTES

- ENGLISH MEANING: Find various data items for each PROD_NUM value in the SALE table where the PADDING field value is larger than 'A' and there is (for the particular PROD_NUM value) more than one underlying row, exactly one PADDING value, and an average sale HOUR greater than four. Finally, the PROD_NUM must either be less than 2004 or between 2224 and 2226.

- WARNING: Asking for those PROD_NUM values with an average sale hour greater than four is not the same as saying; an average time after 4 a.m. It actually means an average time equal to, or more than, 5 a.m.

- GROUP BY: The GROUP BY clause is used to group individual rows into combined sets based on the value in one, or more, columns. Note that explicit column names must be used, DB2/2 will not accept expressions or column numbers.

- HAVING: The HAVING clause is where predicates on the result of the GROUP BY are applied. Also, any field which is in the GROUP BY can also be processed in the HAVING part of the statement. A HAVING clause can also exist without a corresponding GROUP BY.

- ORDER BY: The ORDER BY clause can be used to order the output.

HAVING WITHOUT GROUP BY

It is possible to use a HAVING statement without a GROUP BY. In this case, the HAVING applies to all the rows in the table that match the WHERE predicates. Note that in the following statement all of fields in the SELECT list must contain a column function.

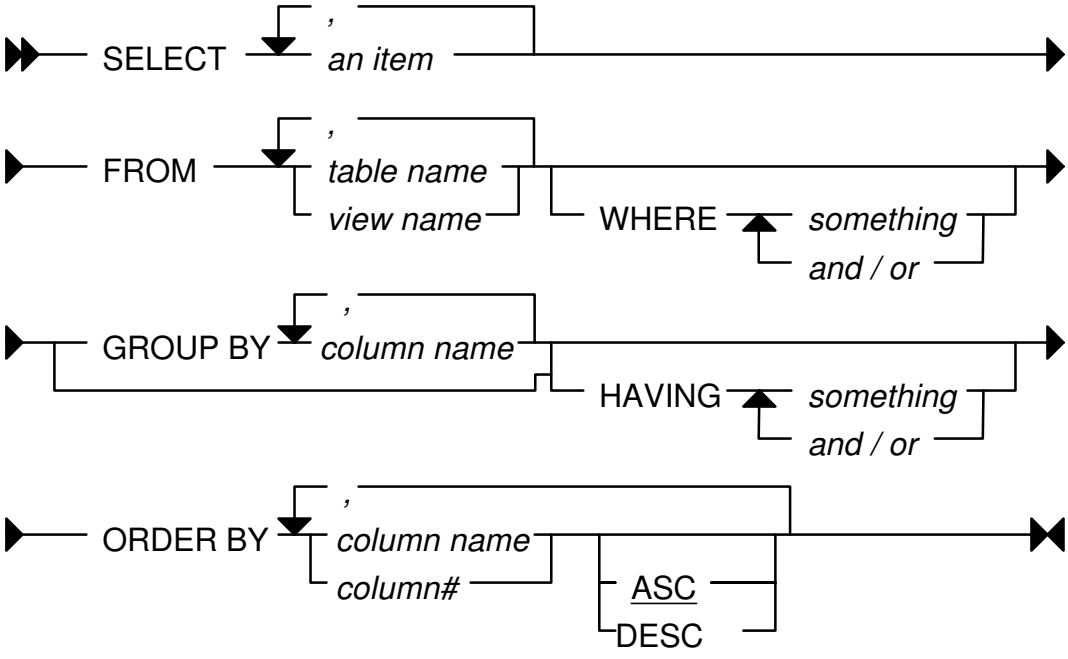
```
SELECT  AVG(PROD_NUM)
       ,SUM(CUST_NUM)
       ,COUNT(*)
FROM    SALE_TABLE
WHERE   PROD_NUM       < 2205
HAVING  AVG(NUM_SOLD) > 5
   AND (MAX(NUM_SOLD) < 123456
    OR  COUNT(*)       > 22)
   AND  COUNT(*)       > COUNT(DISTINCT CUST_NUM);
```

```
   Avg       Sum     Count
======    ======    ======
  2075    651000       520
```

ANSWER, SQL ON NEXT PAGE

```
PROD_NUM     SUM       AVG     COUNT
--------  -------   -------   -----
    2001    46000      4600      10
    2002    47000      4700      10
    2003        ?         ?      10
    2224        ?         ?      10
    2225        ?         ?      10
    2226        ?         ?      10
```

Group By



Example

```
SELECT   PROD_NUM
        ,SUM(CUST_NUM)
        ,AVG(CUST_NUM)
        ,COUNT(*)
FROM     SALE_TABLE
WHERE    SALE_PADDING > 'A'
GROUP BY PROD_NUM
HAVING   COUNT(*) > 1
   AND   COUNT(DISTINCT SALE_PADDING) = 1
   AND   AVG(HOUR(SALE_TIME)) >  4
   AND  (PROD_NUM < 2004
    OR   PROD_NUM BETWEEN 2224 AND  2226)
ORDER BY PROD_NUM ASC
        ,2 DESC;
```

Joins, Unions, & Subqueries

INTRODUCTION

- TYPES OF JOIN: This page describes the various kinds of logical join (between tables) which SQL can be used to effect. Some of these are very easy to write, others take more work.

- SAMPLE TABLES NOT USED IN SQL: The two sample tables shown on the next page (and used throughout this chapter) are not used in any SQL statements. They simply illustrate the kind of answer one would get from a particular type of join.

JOIN TYPES

- EQUI-JOIN: Get only those rows that exists in both tables and which have matching key values.

- OUTER-JOIN: Get all of the rows in one (or both) tables and, for those that match by key value, get the related data from the other table.

- CARTESIAN PRODUCT: Each row in one table is matched against all of the rows in the other. Observe that any common columns have to be selected once for each table involved.

- UNION: Get a combined table which has rows from both tables.

- INTERSECT: Get all of the rows in the 1st table, which have related rows in the 2nd.

- EXCEPT: Get all of the rows in the 1st table, which do NOT have related rows in the 2nd.

- SUB-QUERY: Rows are retrieved only from one table but, before retrieval, they are matched against the contents of a second table.

Joins, Unions, & Subqueries

Table #1

Col1	Col2
A	AA
B	BB
C	CC

Table #2

Col1	Col3
A	AAA
B	BBB

EQUI-JOIN on Col1

Col1	Col2	Col3
A	AA	AAA
B	BB	BBB

OUTER-JOIN on Col1

Col1	Col2	Col3
A	AA	AAA
B	BB	BBB
C	CC	

CARTESIAN-PRODUCT

Col1	Col2	Col1	Col3
A	AA	A	AAA
B	BB	A	AAA
C	CC	A	AAA
A	AA	B	BBB
B	BB	B	BBB
C	CC	B	BBB

UNION

Col1	Col2	Col3
A	AA	
B	BB	
C	CC	
A		AAA
B		BBB

INTERSECT

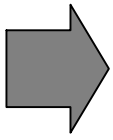
Col1
A
B

EXCEPT

Col1
C

SUB-QUERY

Col1	Col2
A	AA
B	BB
C	CC



Col2	Col3
AA	AAA
BB	BBB

Equi Join

NOTES

- ENGLISH MEANING: Find all those matching rows in the PROD and SALE tables where the Product number is less than 2004, the hour of the sale is before 5, and the Customer number is not null.

- MATCHING ROWS ONLY GOT: Only those rows that exists in both tables will be returned.

- MOST JOINS THIS WAY: In the real world must joins are of this kind.

- EQUAL MATCH ON ALL TABLES BUT ONE: 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.

MATCHING DATA, PROD-TABLE

PROD_NUM	PROD_NAME	PROD_TYPE	PROD_COST	PROD_PADDING
2000	P000	J	2000	ABC%DEF
2001	P001	J	2000	PROD
2002	P002	J	2000	PROD
2003	P003	J	2000	PROD

MATCHING DATA, SALE-TABLE

PROD_NUM	SALE_DATE	SALE_TIME	NUM_SOLD	ITEM_COST	CUST_NUM	SALE_PAD
2001	1994-01-01	03.01.00	90	.00	9100	SALE
2001	1994-01-01	04.01.00	80	.00	8100	SALE
2002	1994-01-01	03.02.00	90	.00	9200	SALE
2002	1994-01-01	04.02.00	80	.00	8200	SALE

Equi Join

Table #1

Col1	Col2
A	AA
B	BB
C	CC

Table #2

Col1	Col3
A	AAA
B	BBB

EQUI-JOIN on Col1

Col1	Col2	Col3
A	AA	AAA
B	BB	BBB

Example

```
SELECT   A.PROD_NUM
        ,B.SALE_TIME
        ,A.PROD_TYPE
FROM     PROD_TABLE A
        ,SALE_TABLE B
WHERE    A.PROD_NUM  < 2004
  AND    A.PROD_NUM  = B.PROD_NUM
  AND    HOUR(B.SALE_TIME) < 5
  AND    B.CUST_NUM   IS NOT NULL
ORDER BY 1 ASC
        ,2 DESC;
```

```
PROD_NUM  SALE_TIME  PROD_TYPE
--------  ---------  ---------
    2001  04.01.00   J
    2001  03.01.00   J
    2002  04.02.00   J
    2002  03.02.00   J
```

Equi Join

NOTES

- ENGLISH MEANING: Get various fields from the CITY, WHSE, WP, and PROD table where the tables match by common key. Apply other non-key criteria to limit the number of rows returned. Also, only get those rows where there is a corresponding Product number in the SALE table for a sale that occurred before 5am to a known customer at a known price.

- BOLD LINES: These lines indicate the predicates which relate the key of one table to another.

Equi Join

```
SELECT   D.PROD_TYPE
        ,A.CITY_RGN
        ,D.*
FROM     CITY_TABLE A
        ,WHSE_TABLE B
        ,WP_TABLE   C
        ,PROD_TABLE D
WHERE    A.CITY_DATA   = 'E'
  AND    A.CITY_NUM    = B.CITY_NUM
  AND  ( B.WHSE_NAME   = '0051'
   OR    B.WHSE_NAME   = '0053'
   OR    B.WHSE_SIZE   <  999 )
  AND    B.CITY_NUM    = C.CITY_NUM
  AND    B.WHSE_NUM    = C.WHSE_NUM
  AND    C.WP_PADDING <> ' '
  AND    C.PROD_NUM    = D.PROD_NUM
  AND    D.PROD_COST   < 2011
  AND  ( D.PROD_TYPE   = 'I'
   OR    D.PROD_TYPE   = 'K' )
  AND    D.PROD_NAME  <> ' '
  AND    EXISTS
        (SELECT *
         FROM   SALE_TABLE E
         WHERE  D.PROD_NUM  = E.PROD_NUM
           AND  E.SALE_TIME <  '05.00.00'
           AND  E.ITEM_COST IS NOT NULL
           AND  E.CUST_NUM  IS NOT NULL)
ORDER BY 1, 2, 3 DESC;
```

PROD TYPE	CITY RGN	PROD NUM	PROD NAME	PROD TYPE	PROD COST	PROD PADDING
I	C1	2102	0103	I	2010	PROD
I	C1	2102	0103	I	2010	PROD
I	C1	2102	0103	I	2010	PROD
I	C1	2102	0103	I	2010	PROD
I	C1	2101	0102	I	2010	PROD
I	C1	2101	0102	I	2010	PROD
I	C1	2101	0102	I	2010	PROD
I	C1	2101	0102	I	2010	PROD

Outer Join

NOTES

- ENGLISH MEANING: Find all those matching rows in the PROD and SALE tables where the Product number is less than 2004, the hour of the sale is before 5, and the Customer number is not null. Also get all rows from the PROD table where the Product number is less than 2004 and there is no matching row in the SALE table.

- DESCRIPTION: Get all of the rows in one (or both) tables and, for those that match, get the related data from the other table.

- NEEDS UNION: The DB2/2 SQL language, unlike some other SQL languages, does not support the outer join construct. To do an outer join one must combine a equi-join and a "not exists" subquery using a union.

- USE VIEW: A view can be defined on the outer-join union SQL in order to hide its complexity from users. Unfortunately, processing this view will often be rather inefficient because all of the rows that match the view search criteria will be obtained and stored in a work file before any predicates that external to the view are applied.

MATCHING DATA, PROD-TABLE

PROD_NUM	PROD_NAME	PROD_TYPE	PROD_COST	PROD_PADDING
2000	P000	J	2000	ABC%DEF
2001	P001	J	2000	PROD
2002	P002	J	2000	PROD
2003	P003	J	2000	PROD

MATCHING DATA, SALE-TABLE

PROD_NUM	SALE_DATE	SALE_TIME	NUM_SOLD	ITEM_COST	CUST_NUM	SALE_PAD
2001	1994-01-01	03.01.00	90	.00	9100	SALE
2002	1994-01-01	03.02.00	90	.00	9200	SALE
2003	1994-01-01	03.03.00	90	.00	--------	SALE

Outer Join

Table #1

Col1	Col2
A	AA
B	BB
C	CC

Table #2

Col1	Col3
A	AAA
B	BBB

OUTER-JOIN on Col1

Col1	Col2	Col3
A	AA	AAA
B	BB	BBB
C	CC	

Example

```
SELECT   'FOUND'
        ,A.PROD_NUM
        ,B.SALE_TIME
        ,A.PROD_TYPE
FROM     PROD_TABLE A
        ,SALE_TABLE B
WHERE    A.PROD_NUM  < 2004
  AND    A.PROD_NUM  = B.PROD_NUM
  AND    HOUR(B.SALE_TIME) < 4
  AND    B.CUST_NUM IS NOT NULL
UNION ALL
SELECT   'NOT FOUND'
        ,A.PROD_NUM
        ,TIME('00.00.00')
        ,A.PROD_TYPE
FROM     PROD_TABLE A
WHERE    A.PROD_NUM  < 2004
  AND    NOT EXISTS
        (SELECT *
         FROM   SALE_TABLE B
         WHERE  A.PROD_NUM  = B.PROD_NUM
         AND    HOUR(B.SALE_TIME) < 4
         AND    B.CUST_NUM IS NOT NULL)
ORDER BY 1, 2;
```

```
               PROD_NUM  SALE_TIME  PROD_TYPE
-----------    --------  ---------  ---------
FOUND              2001  03.01.00   J
FOUND              2002  03.02.00   J
NOT FOUND          2000  00.00.00   J
NOT FOUND          2003  00.00.00   J
```

Cartesian Product

NOTES

- ENGLISH MEANING: Find all rows in the PROD where the Product number is less than 2004. Also find every row in the SALE table where the Product number is less than 2004 and the time of the sale was before 5 a.m. Join each row found in one table to all the matching rows the other table.

- DESCRIPTION: Each row that matches in one table is joined to all of the rows that were obtained from the other table.

- WARNING: A Cartesian Product is almost always the wrong result. There are very few business cases where it makes sense to use this kind of SQL.

MATCHING DATA, PROD-TABLE

```
PROD_NUM  PROD_NAME  PROD_TYPE  PROD_COST  PROD_PADDING
========  =========  ========   =========  ============
    2000  0001       J               2000  ABC%DEF
    2001  0002       J               2000  PROD
    2002  0003       J               2000  PROD
```

MATCHING DATA, SALE-TABLE

```
PROD_NUM  SALE_DATE   SALE_TIME  NUM_SOLD  ITEM_COST  CUST_NUM  SALE_PAD
========  ==========  =========  ========  =========  ========  ========
    2001  1994-01-01  03.01.00         90        .00      9100  SALE
    2001  1994-01-01  04.01.00         80        .00      8100  SALE
    2002  1994-01-01  03.02.00         90        .00      9200  SALE
    2002  1994-01-01  04.02.00         80        .00      8200  SALE
```

CORRECT SQL (ALTERNATIVE)

```
SELECT   A.PROD_NUM
        ,B.PROD_NUM
        ,A.PROD_TYPE
        ,B.SALE_DATE
        ,B.SALE_TIME
FROM     PROD_TABLE A
        ,SALE_TABLE B
WHERE    A.PROD_NUM  = B.PROD_NUM
  AND    A.PROD_NUM  < 2003
  AND    B.PROD_NUM  < 2003
  AND    B.SALE_TIME < '05.00.00'
ORDER BY 1, 2;
```

```
PROD_NUM  PROD_NUM  PROD_TYPE  SALE_DATE   SALE_TIME
--------  --------  ---------  ----------  ---------
    2001      2001  J          1994-01-01  03.01.00
    2001      2001  J          1994-01-01  04.01.00
    2002      2002  J          1994-01-01  03.02.00
    2002      2002  J          1994-01-01  04.02.00
```

Cartesian Product

Table #1

Col1	Col2
A	AA
B	BB
C	CC

Table #2

Col1	Col3
A	AAA
B	BBB

CARTESIAN-PRODUCT

Col1	Col2	Col1	Col3
A	AA	A	AAA
B	BB	A	AAA
C	CC	A	AAA
A	AA	B	BBB
B	BB	B	BBB
C	CC	B	BBB

Example

```
SELECT   A.PROD_NUM
        ,B.PROD_NUM
        ,A.PROD_TYPE
        ,B.SALE_DATE
        ,B.SALE_TIME
FROM     PROD_TABLE A
        ,SALE_TABLE B
WHERE    A.PROD_NUM  < 2003
  AND    B.PROD_NUM  < 2003
  AND    B.SALE_TIME < '05.00.00'
ORDER BY 1, 2;
```

```
PROD_NUM  PROD_NUM  PROD_TYPE  SALE_DATE   SALE_TIME
--------  --------  ---------  ----------  ---------
    2000      2001  J          1994-01-01  03.01.00
    2000      2001  J          1994-01-01  04.01.00
    2000      2002  J          1994-01-01  03.02.00
    2000      2002  J          1994-01-01  04.02.00
    2001      2001  J          1994-01-01  03.01.00
    2001      2001  J          1994-01-01  04.01.00
    2001      2002  J          1994-01-01  03.02.00
    2001      2002  J          1994-01-01  04.02.00
    2002      2001  J          1994-01-01  03.01.00
    2002      2001  J          1994-01-01  04.01.00
    2002      2002  J          1994-01-01  03.02.00
    2002      2002  J          1994-01-01  04.02.00
```

Union & Union All

NOTES

- ENGLISH MEANING: Find all those rows in the PROD table where the Product number is less than 2004. Also find those rows in the SALE table Product number is less than 2004, the hour of the sale is less than 5, and the Customer number is not null.

- UNION: Gets the DISTINCT set of all matching rows.

- UNION ALL: Does not remove duplicate rows.

- UNION OUTPUT ORDER: If a UNION ALL is used in a query, and no ORDER BY is provided, the data will be returned as it is found (i.e: from the top part of the UNION first, then the lower part(s)). If a simple UNION is used, the data will be sorted (to remove duplicates) from left to right.

- UNION & GROUP BY: A union statement can refer to the result of a GROUP BY clause. However, the result of a union can never be used in a GROUP BY.

- UNION & VIEWS: A union statement can refer to the result of a VIEW. However, a view can never contain a union.

MATCHING DATA, PROD-TABLE

PROD_NUM	PROD_NAME	PROD_TYPE	PROD_COST	PROD_PADDING
2000	P000	J	2000	ABC%DEF
2001	P001	J	2000	PROD
2002	P002	J	2000	PROD
2003	P003	J	2000	PROD

MATCHING DATA, SALE-TABLE

PROD_NUM	SALE_DATE	SALE_TIME	NUM_SOLD	ITEM_COST	CUST_NUM	SALE_PAD
2001	1994-01-01	03.01.00	90	.00	9100	SALE
2001	1994-01-01	04.01.00	80	.00	8100	SALE
2002	1994-01-01	03.02.00	90	.00	9200	SALE
2002	1994-01-01	04.02.00	80	.00	8200	SALE

Union & Union All

Table #1

Col1	Col2
A	AA
B	BB
C	CC

Table #2

Col1	Col3
A	AAA
B	BBB

UNION

Col1	Col2	Col3
A	AA	
B	BB	
C	CC	
A		AAA
B		BBB

Example

```
SELECT   PROD_NUM
        ,PROD_TYPE
        ,TIME('00.00.00')
FROM     PROD_TABLE
WHERE    PROD_NUM  < 2004
UNION ALL
SELECT   PROD_NUM
        ,' '
        ,SALE_TIME
FROM     SALE_TABLE
WHERE    PROD_NUM        < 2004
  AND    HOUR(SALE_TIME) < 5
  AND    CUST_NUM   IS NOT NULL
ORDER BY 1 ASC
        ,2 DESC;
```

```
PROD_NUM  PROD_TYPE
--------  ---------  --------
    2000  J          00.00.00
    2001  J          00.00.00
    2001             03.01.00
    2001             04.01.00
    2002  J          00.00.00
    2002             03.02.00
    2002             04.02.00
    2003  J          00.00.00
```

Union & Union All

NOTES

- UNION: Append one set of rows to another, remove any duplicate values.

- UNION ALL: Append one set of rows to another, do NOT remove any duplicate values.

- PROCESSING SEQUENCE: In general (but there are no promises) DB2/2 will process a UNION, EXCEPT, or INTERSECT statement from top to bottom. If brackets exist, the code in brackets will be done first.

- 1ST SQL: Get all CITY_NUM values less than 5. Append to this result to the same data again - do not remove duplicates. Finally, append to the same result again, this time remove all duplicate values. This query is processed from top to bottom.

- 2ND SQL: Get all CITY_NUM values less than 5. Append to this result to the same data again - remove duplicates. Finally, append to the same result again, this time retain duplicate values. This query is processed from top to bottom.

- 3RD SQL: Do the SQL in brackets first: Then do the UNION ALL.

CITY-TABLE - SAMPLE ROWS

CITY_NUM	CITY_RGN	CITY_DATA	CITY_PADDING
0	C0	J	CITY
1	C1	I	CITY
2	C0	H	CITY
3	C1	G	CITY
4	C0	F	CITY

Union & Union All

Process Top to Bottom

```
SELECT CITY_NUM FROM CITY_TABLE WHERE CITY_NUM < 5
UNION ALL
SELECT CITY_NUM FROM CITY_TABLE WHERE CITY_NUM < 5
UNION
SELECT CITY_NUM FROM CITY_TABLE WHERE CITY_NUM < 5;

>> 0, 1, 2, 3, 4
```

```
SELECT CITY_NUM FROM CITY_TABLE WHERE CITY_NUM < 5
UNION
SELECT CITY_NUM FROM CITY_TABLE WHERE CITY_NUM < 5
UNION ALL
SELECT CITY_NUM FROM CITY_TABLE WHERE CITY_NUM < 5;

>> 0, 1, 2, 3, 4, 0, 1, 2, 3, 4
```

Process Brackets First

```
SELECT CITY_NUM FROM CITY_TABLE WHERE CITY_NUM < 5
UNION ALL
(SELECT CITY_NUM FROM CITY_TABLE WHERE CITY_NUM < 5
UNION
SELECT CITY_NUM FROM CITY_TABLE WHERE CITY_NUM < 5);

>> 0, 1, 2, 3, 4, 0, 1, 2, 3, 4
```

Except & Intersect

NOTES

- INTERSECT: Get all of the rows in the 1st table, which have related rows in the 2nd.
- EXCEPT: Get all of the rows in the 1st table, which do NOT have related rows in the 2nd.
- MATCH CRITERIA: An EXCEPT and/or INTERSECT process is done by matching ALL of the columns in the top and bottom result-sets. 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 (see page 68).

UNION, EQUIVALENT QUERIES

```
SELECT CITY_RGN FROM CITY_TABLE WHERE CITY_NUM < 5
UNION
SELECT CITY_RGN FROM CITY_TABLE WHERE CITY_NUM = 2;

>>  C0, C1


SELECT CITY_RGN FROM CITY_TABLE WHERE CITY_NUM < 5
UNION ALL
SELECT CITY_RGN FROM CITY_TABLE WHERE CITY_NUM = 2;

>>  C0, C1, C0, C1, C0, C0
```

CITY-TABLE - SAMPLE ROWS

CITY_NUM	CITY_RGN	CITY_DATA	CITY_PADDING
0	C0	J	CITY
1	C1	I	CITY
2	C0	H	CITY
3	C1	G	CITY
4	C0	F	CITY

Except & Intersect

Table #1

Col1	Col2
A	AA
B	BB
C	CC

Table #2

Col1	Col3
A	AAA
B	BBB

INTERSECT

Col1
A
B

EXCEPT

Col1
C

Intersect

```
SELECT CITY_RGN FROM CITY_TABLE WHERE CITY_NUM < 5
INTERSECT
SELECT CITY_RGN FROM CITY_TABLE WHERE CITY_NUM = 2;

>>  C0


SELECT CITY_RGN FROM CITY_TABLE WHERE CITY_NUM < 5
INTERSECT ALL
SELECT CITY_RGN FROM CITY_TABLE WHERE CITY_NUM = 2;

>>  C0, C0, C0
```

Except

```
SELECT CITY_RGN FROM CITY_TABLE WHERE CITY_NUM < 5
EXCEPT
SELECT CITY_RGN FROM CITY_TABLE WHERE CITY_NUM = 2;

>>  C1


SELECT CITY_RGN FROM CITY_TABLE WHERE CITY_NUM < 5
EXCEPT ALL
SELECT CITY_RGN FROM CITY_TABLE WHERE CITY_NUM = 2;

>>  C1, C1
```

Except & Intersect

NOTES

- UNION/INTERSECT/EXCEPT PRECEDENCE RULES: Predicates in brackets are done first. INTERSECT predicates are done before either UNION and/or EXCEPT predicates. In general, predicates of equal worth are done from top to bottom.

WITHOUT INTERSECT

- 1ST SQL: First the UNION, then the EXCEPT, is done. The two predicates have equal priority so are typically processed from top to bottom.

- 2ND SQL: First the EXCEPT, then the UNION, is done.

WITH INTERSECT

- 1ST SQL: First the INTERSECT, then the EXCEPT, is done. The former is processed first because it is defined as having a higher precedence.

- 2ND SQL: First the INTERSECT, then the EXCEPT, is done. The former is processed first because it is defined as having a higher precedence.

- 3RD SQL: First the EXCEPT, then the INTERSECT, is done. The former is processed first because it is in brackets.

CITY-TABLE - SAMPLE ROWS

CITY_NUM	CITY_RGN	CITY_DATA	CITY_PADDING
0	C0	J	CITY
1	C1	I	CITY
2	C0	H	CITY
3	C1	G	CITY
4	C0	F	CITY

Except & Intersect

Without Intersect

```
SELECT CITY_NUM FROM CITY_TABLE WHERE CITY_NUM < 1
UNION
SELECT CITY_NUM FROM CITY_TABLE WHERE CITY_NUM < 5
EXCEPT
SELECT CITY_NUM FROM CITY_TABLE WHERE CITY_NUM < 3;
>>  3, 4

SELECT CITY_NUM FROM CITY_TABLE WHERE CITY_NUM < 1
EXCEPT
SELECT CITY_NUM FROM CITY_TABLE WHERE CITY_NUM < 3
UNION
SELECT CITY_NUM FROM CITY_TABLE WHERE CITY_NUM < 5;
>>  0, 1, 2, 3, 4
```

With Intersect

```
SELECT CITY_NUM FROM CITY_TABLE WHERE CITY_NUM < 5
INTERSECT
SELECT CITY_NUM FROM CITY_TABLE WHERE CITY_NUM < 4
EXCEPT
SELECT CITY_NUM FROM CITY_TABLE WHERE CITY_NUM < 2;
>>  2, 3

SELECT CITY_NUM FROM CITY_TABLE WHERE CITY_NUM < 5
EXCEPT
SELECT CITY_NUM FROM CITY_TABLE WHERE CITY_NUM < 2
INTERSECT
SELECT CITY_NUM FROM CITY_TABLE WHERE CITY_NUM < 4;
>>  2, 3, 4

(SELECT CITY_NUM FROM CITY_TABLE WHERE CITY_NUM < 5
EXCEPT
SELECT CITY_NUM FROM CITY_TABLE WHERE CITY_NUM < 2)
INTERSECT
SELECT CITY_NUM FROM CITY_TABLE WHERE CITY_NUM < 4;
>>  2, 3
```

Sub-query - Syntax

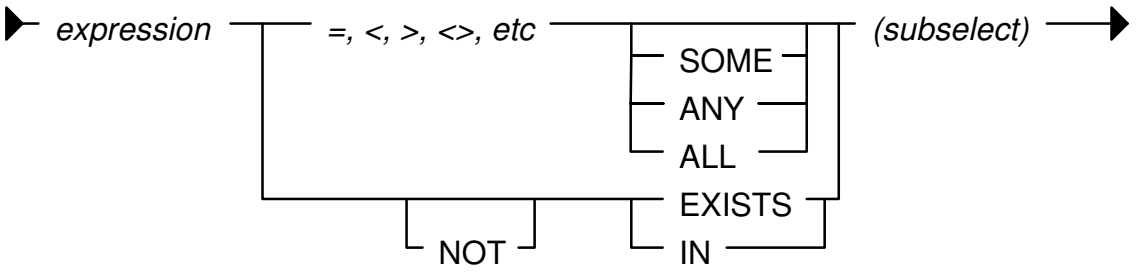
NOTES

- ALL UNCORRELATED: Note that all of the subqueries on the next page are uncorrelated.
- ANY: The ANY clause means that the subquery will be TRUE if there is any row that matches.
- SOME: The SOME clause works exactly the same way as the ANY. The SOME will be true even if only one value matches.
- ALL: The ALL clause is true only if all of the rows in the subquery match.
- EXISTS: If an EXISTS is used there is no need to specify a particular column as being selected in the subquery. For all other examples, it is field values that are being compared, not row existence.
- WARNING: If a subquery call does not have an IN, EXISTS, SOME, ANY or ALL clause and, instead, goes straight to the subquery, there is a chance that a SQL error will be encountered. When none of the above are used, DB2/2 assumes that only one value will be returned by the subquery. If more than one is returned, an error is generated.

ENGLISH MEANING, SUBQUERY CALLS

Product# IN	The current Product# is IN the list of Product numbers in the SALE table.
Product# =	The current Product# equals the MIN Product number in the SALE table.
Product# >	The current Product# is greater than the MIN Product number in the SALE table.
Product# = ANY	The current Product# equals ANY Product number in the SALE table.
Product# = SOME	The current Product# equals SOME Product number(s) in the SALE table.
Product# = ALL	The current Product# equals ALL of the Product number(s) in the SALE table.
EXISTS	There exists at least one row in the SALE table.
NOT EXISTS	There does not exist any rows in the SALE table.

Sub-query - Syntax



Example

```
SELECT   *
FROM     PROD_TABLE
WHERE    PROD_NUM  < 2003
  AND (  PROD_NUM IN
        (SELECT PROD_NUM FROM SALE_TABLE)
   OR    PROD_NUM =
        (SELECT MIN(PROD_NUM) FROM SALE_TABLE)
   OR    PROD_NUM >
        (SELECT MIN(PROD_NUM) FROM SALE_TABLE)
   OR    PROD_NUM = ANY
        (SELECT PROD_NUM FROM SALE_TABLE)
   OR    PROD_NUM = SOME
        (SELECT PROD_NUM FROM SALE_TABLE)
   OR    PROD_NUM = ALL
        (SELECT PROD_NUM FROM SALE_TABLE)
   OR    EXISTS
        (SELECT PROD_NUM FROM SALE_TABLE)
   OR    NOT EXISTS
        (SELECT * FROM SALE_TABLE) )
ORDER BY 1;
```

```
PROD_NUM  PROD_NAME  PROD_TYPE  PROD_COST  PROD_PADDING
--------  ---------  ---------  ---------  ------------
    2000  P000       J               2000  ABC%DEF
    2001  P001       J               2000  PROD
    2002  P002       J               2000  PROD
```

Sub-query - Uncorrelated

NOTES

- ENGLISH MEANING: Find all rows in the PROD table where this is at least one matching row in the SALE table with a Product number less than 2003 and a sale time before 5 a.m.

- DEFINITION: An uncorrelated subquery is one where the matching WHERE conditions on the subquery table have no direct relationship to the current row being processed in the top table (hence uncorrelated).

- DB2/2 LOGIC: In general, DB2/2 will resolve the subquery at the start of the SQL statement and then use the result throughout the remainder. Note that the subquery is only invoked once.

CORRELATED VS. UNCORRELATED VS. INTERSECT/EXCEPT - WHEN TO USE

- LOOK FOR RELATED ROW: Checks to look for related key values can be written either way. Ditto for the negative equivalent.

- LOOK FOR ANY ROW: Checks to find unrelated key values (e.g: check any/all rows) have to be written as an uncorrelated subquery. Ditto for the negative equivalent.

- SUB-QUERY vs. EXCEPT/INTERSECT: An EXCEPT and/or INTERSECT process is done by matching ALL of the columns in the top and bottom result-sets. 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.

MATCHING DATA, PROD-TABLE

PROD_NUM	PROD_NAME	PROD_TYPE	PROD_COST	PROD_PADDING
2000	P000	J	2000	ABC%DEF
2001	P001	J	2000	PROD
2002	P002	J	2000	PROD

MATCHING DATA, SALE-TABLE

PROD_NUM	SALE_DATE	SALE_TIME	NUM_SOLD	ITEM_COST	CUST_NUM	SALE_PAD
2001	1994-01-01	03.01.00	90	.00	9100	SALE
2001	1994-01-01	04.01.00	80	.00	8100	SALE
2002	1994-01-01	03.02.00	90	.00	9200	SALE
2002	1994-01-01	04.02.00	80	.00	8200	SALE

Sub-query - Uncorrelated

Table #1

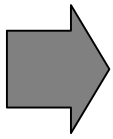
Col1	Col2
A	AA
B	BB
C	CC

Table #2

Col1	Col3
A	AAA
B	BBB

SUB-QUERY

Col1	Col2
A	AA
B	BB
C	CC



Col2	Col3
AA	AAA
BB	BBB

Example

```
SELECT   *
FROM     PROD_TABLE A
WHERE    A.PROD_NUM  < 2003
  AND    A.PROD_NUM IN
        (SELECT B.PROD_NUM
         FROM   SALE_TABLE B
         WHERE  B.PROD_NUM  < 2003
           AND  B.SALE_TIME < '05.00.00');
```

```
PROD_NUM  PROD_NAME  PROD_TYPE  PROD_COST  PROD_PADDING
--------  ---------  ---------  ---------  ------------
    2001  P001       J               2000  PROD
    2002  P002       J               2000  PROD
```

Sub-query - Correlated

NOTES

- ENGLISH MEANING: Find all rows in the PROD table where this is at least one matching row in the SALE table with a Product number less than 2003 and a sale time before 5 a.m.

- DEFINITION: An correlated subquery is one where the matching WHERE conditions on the subquery table can not be resolved without reference to the row currently being processed in the top table (hence correlated).

- DB2/2 LOGIC: In general, DB2/2 will resolve the subquery for each row that matches in the top table. This means that the subquery will probably be invoked multiple times.

MATCHING DATA, PROD-TABLE

PROD_NUM	PROD_NAME	PROD_TYPE	PROD_COST	PROD_PADDING
2000	P000	J	2000	ABC%DEF
2001	P001	J	2000	PROD
2002	P002	J	2000	PROD

MATCHING DATA, SALE-TABLE

PROD_NUM	SALE_DATE	SALE_TIME	NUM_SOLD	ITEM_COST	CUST_NUM	SALE_PAD
2001	1994-01-01	03.01.00	90	.00	9100	SALE
2001	1994-01-01	04.01.00	80	.00	8100	SALE
2002	1994-01-01	03.02.00	90	.00	9200	SALE
2002	1994-01-01	04.02.00	80	.00	8200	SALE

Sub-query - Correlated

Table #1

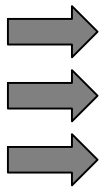
Col1	Col2
A	AA
B	BB
C	CC

Table #2

Col1	Col3
A	AAA
B	BBB

SUB-QUERY

Col1	Col2
A	AA
B	BB
C	CC



Col2	Col3
AA	AAA
BB	BBB

Example

```
SELECT   *
FROM     PROD_TABLE A
WHERE    A.PROD_NUM  < 2003
  AND    A.PROD_NUM IN
        (SELECT B.PROD_NUM
         FROM   SALE_TABLE B
         WHERE  A.PROD_NUM  = B.PROD_NUM
           AND  B.SALE_TIME < '05.00.00');
```

```
PROD_NUM  PROD_NAME  PROD_TYPE  PROD_COST  PROD_PADDING
--------  ---------  ---------  ---------  ------------
    2001  P001       J               2000  PROD
    2002  P002       J               2000  PROD
```

Subquery - Flavours

NOTES

- INTRODUCTION: Subqueries can be written to check for the presence the partial-presence (i.e: they are only true if exactly 10 rows are found), the non-presence, or the complete presence (i.e: they are only true if all rows found match) of related rows in the lower table.

- TRUE IF NONE MATCH: In this statement, the subquery is true only when NONE of the rows in SALE table match the current row in the PROD table.

- TRUE IF 10 MATCH: In these statements, the subquery is true only when exactly 10 of the rows in the SALE table match the current row in the PROD table. In the 2nd example we have assumed that the PROD_NUM is never less than zero, and that the COUNT will never be more than 999. If either is not true, the result may be incorrect.

  MOST EFFICIENT VERSION: Of the two uncorrelated queries shown here, the third is probably the more efficient because the work file it uses will be smaller (i.e: only containing values with more than 10 rows).

Subquery - Flavours

True if NONE match (Correlated & Uncorrelated)

```
SELECT COUNT(*)
FROM   PROD_TABLE A
WHERE  NOT EXISTS
      (SELECT *
       FROM   SALE_TABLE B
       WHERE  A.PROD_NUM = B.PROD_NUM);

SELECT COUNT(*)
FROM   PROD_TABLE
WHERE  PROD_NUM NOT IN
      (SELECT PROD_NUM
       FROM   SALE_TABLE);
```

True if TEN match (Correlated & Uncorrelated)

```
SELECT COUNT(*)
FROM   PROD_TABLE A
WHERE  10 =
      (SELECT COUNT(*)
       FROM   SALE_TABLE B
       WHERE  A.PROD_NUM = B.PROD_NUM);

SELECT COUNT(*)
FROM   PROD_TABLE
WHERE  PROD_NUM + (10*0.001) IN
      (SELECT PROD_NUM + (COUNT(*)*0.001)
       FROM   SALE_TABLE  GROUP BY PROD_NUM);

SELECT COUNT(*)
FROM   PROD_TABLE
WHERE  PROD_NUM IN
      (SELECT   PROD_NUM
       FROM     SALE_TABLE
       GROUP BY PROD_NUM
       HAVING   COUNT(*) = 10);
```

Subquery - Flavours

NOTES

- TRUE IF ANY MATCH: In these statements, the subquery is true if any one (or more) of the rows the SALE table has the same value as the current row in the PROD table.

- TRUE IF ALL MATCH: In this statement, the subquery is true only when ALL of the rows in SALE table match the current row in the PROD table. It is not possible to write a correlated equivalent of this statement.

Subquery - Flavours

True if ANY match (Correlated & Uncorrelated)

```
SELECT COUNT(*)
FROM   PROD_TABLE A
WHERE  EXISTS
      (SELECT *
       FROM   SALE_TABLE B
       WHERE  A.PROD_NUM = B.PROD_NUM);

SELECT COUNT(*)
FROM   PROD_TABLE
WHERE  PROD_NUM = ANY
      (SELECT PROD_NUM
       FROM   SALE_TABLE);

SELECT COUNT(*)
FROM   PROD_TABLE
WHERE  PROD_NUM = SOME
      (SELECT PROD_NUM
       FROM   SALE_TABLE);

SELECT COUNT(*)
FROM   PROD_TABLE
WHERE  PROD_NUM IN
      (SELECT PROD_NUM
       FROM   SALE_TABLE);
```

True if ALL match (Uncorrelated only)

```
SELECT COUNT(*)
FROM   PROD_TABLE
WHERE  PROD_NUM = ALL
      (SELECT PROD_NUM
       FROM   SALE_TABLE);
```

Exercise 2

DDL

See page 122 for a description of the tables used.

Q1

Select from the SALE table the following two fields:
- The PROD_NUM.
- A COUNT of rows (per individual product number).

Get for those rows where all of the following is true:
- The Product number is less than 2010.
- The Customer number is NOT NULL.
- The number of DISTINCT sale dates (per individual product number) is 1.

Write the query using a GROUP BY and HAVING clause, see page 48 for help.

Q2

Select from the SALE table the following two fields:
- The PROD_NUM.
- A COUNT of rows.

Do a GROUP BY on the PROD_NUM field selecting where the following are true:
- The Product number is less than 2010.
- The Customer number is NULL.
- The number or rows per individual product number is more than 5.
- The number of distinct SALE dates per individual product number is 1.
- The average hour of the sale (per individual product number) is greater than 4.

Hint: Use a HAVING in this statement (see page 48).

Q3

Select ALL of the fields from the SALE table where the following are true:
- The Product number is less than 2005.
- The item cost is less than 10.
- The sale time was before 5am.
- The number of DISTINCT not null customers (for each product number) more than 1.

Note: You will need to write a subquery to answer the last part of this problem. This is because you need to match each row found against all of the other rows with the same product number to see how many not null customers (per product number) there are. Do this subquery last, first write the rest of the SQL statement and test it.

Hint: Write a correlated subquery (matching on Product number). Count the number of distinct not null customer numbers that belong to matching rows in the subquery. See page 72 for help.

Q4

Select from the SALE and PROD tables the following two fields:
- A count of all the matching rows.
- A count of the DISTINCT PROD_TYPE values.

Select those rows where the following are true:
- The two tables have matching PROD_NUM fields.
- The PROD_TYPE begins with a "J".
- The SALE_TIME was before 11 am.

Exercise 2

Q5

Select a COUNT of the rows in the PROD table where the following is true:

- The Product number is between 2000 and 2100
- There are more than 5 rows in the SALE table with the same Product number.
- There are more than 10 rows in the WP table with the same Product number.

Q6

Get a COUNT of all those Products in the PROD table which have not had a sale. Use an uncorrelated subquery. See page 70 for help.

Q7

Get a COUNT of all those Products in the PROD table which have not had a sale. Use a correlated subquery. See page 72 for help.

Q8

Select two counts from the SALE table. One will be a count of those rows where the customer number is NULL, the other will be where the customer number is NOT NULL. Use a UNION statement to get one row per value. Add a text field to the output so we can tell which value is which. See page 61 for help.

Q9

Select all PROD_NUM from the PROD table where the PROD_TYPE is 'J' and there exists a matching PROD_NUM in the SALE table. Use an INTERSECT statement. See page 64 for help.

Q10

Select all CITY_NUM from the CITY table where there is a matching CITY_NUM in the WHSE_TABLE with a WHSE_SIZE less than 104. Use an INTERSECT statement. Next, exclude those rows where there is a matching CITY_NUM in the WHSE_TABLE with a WHSE_NAME greater than "0033". Use an EXCEPT statement. Finally, UNION the above result with those CITY_NUM values in the WHSE_TABLE where the WHSE_SIZE is greater than 108. Do NOT remove duplicates. Present the output in descending order.

Answers

Data: See page 136.
SQL: See page 140.

Fun with SQL

Table of Contents

NOTES

- SUBJECTS COVERED: This section will go over some of the wonderful things you can, and possibly shouldn't, do with DB2/2 SQL.

- EXERCISE: There will be an exercise to do at the end of this chapter.

Table of Contents

Sample Database	84
Making Charts using SQL	86
Number and Character Manipulation	88
Date Manipulation	90
Julian Date Conversion	92
Week Processing in SQL	94
Boolean Logic in SQL	96
Doing Multiple Counts in one Pass	98
Char & Varchar Processing Differences	100
Get data for the MIN / MAX row	102
Get the 10 MIN / MAX rows	104
Selecting Most Recent Data	106
Complicated Subqueries	108
Using 2 Distinct Stmts in a Query	110
Creating test data using SQL	112
Using SQL to generate DDL / DML / SQL	114
Exercise 3	116

Sample Database

NOTES

- SAMPLE APPLICATION: The sample tables describe a simple inventory application. There is a CITY table (with one row per city), a WAREHOUSE table (with one row per city/warehouse), a PRODUCT table (with one row per product), and junction table (WP: with one row per city/warehouse/product). The SALE table records sales of products. Note that not every product has had a sale and those that have may have had multiple sales. Each table is suitably indexed.

- PRODUCT NUMBER VALUES: Note that the Product number values, unlike the other keys values, are not in simple continual ascending sequence. Instead the values go up in sequence for thirty numbers and then skip seventy numbers before going up in sequence for another thirty values, etc. The following table illustrates how they work. The reason why this was done was to allow me to fool the DB2/2 optimizer into choosing sub-optimal access paths (for illustration purposes).

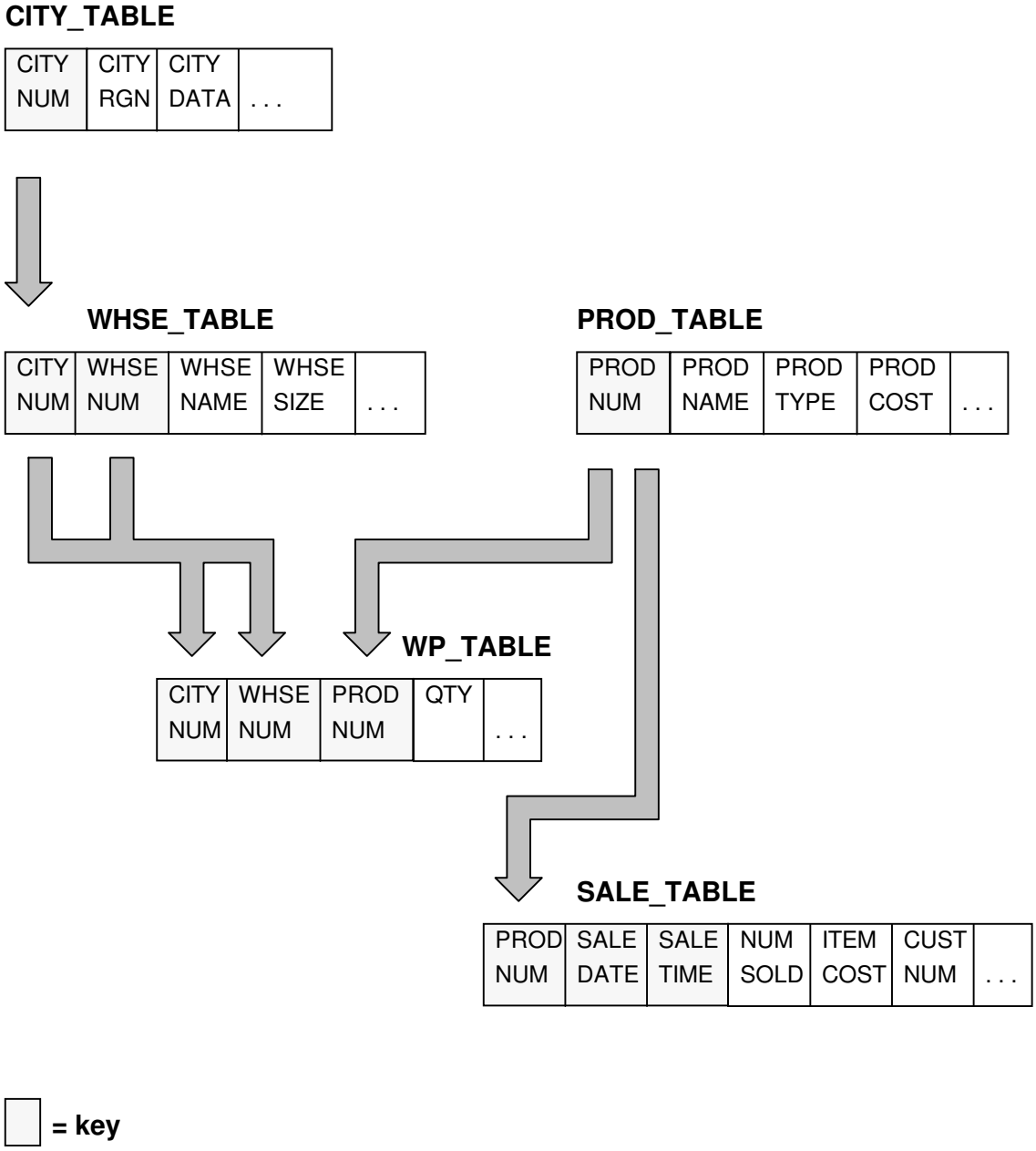
PRODUCT NUMBER VALUES

2000, 2001, 2002, 2003, 2004, 2005,	...	2026, 2027, 2028, 2029
2100, 2101, 2102, 2103, 2104, 2105,	...	2126, 2127, 2128, 2129
2200, 2201, 2202, 2203, 2204, 2205,	...	2226, 2227, 2228, 2229
...		
2900, 2901, 2902, 2903, 2904, 2905,	...	2926, 2927, 2928, 2929

INDEXES

TABLE-NAME	INDEX-NAME	INDEX-TYPE	FULLKEYCARD	INDEX-COLS	COLUMN CARD
CITY_TABLE	ICITYNUM	Unique	10	CITY_NUM	10
	ICITYRGN	Non-unique	2	CITY_RGN	2
WHSE_TABLE	IWHSENUM	Unique	40	CITY_NUM	10
				WHSE_NUM	4
	IWHSENME	Unique	40	WHSE_NAME	40
PROD_TABLE	IPRODNUM	Unique	300	PROD_NUM	300
	IPRODNME	Unique	300	PROD_NAME	300
	IPRODTYP	Non-unique	30	PROD_TYPE	30
WP_TABLE	IWPWHSE	Unique	12,000	CITY_NUM	10
				WHSE_NUM	4
				PROD_NUM	300
	IWPPROD	Unique	12,000	PROD_NUM	300
				CITY_NUM	10
				WHSE_NUM	4
SALE_TABLE	ISALENUM	Unique	2,400	PROD_NUM	240
				SALE_DATE	30
				SALE_TIME	80
	ICUSTNUM	Non-unique	21	CUST_NUM	21

Sample Database



Making Charts using SQL

NOTES

- ENGLISH MEANING: Get the warehouse size where the warehouse number is 100. Display the result as a line-bar chart.

- HOW IT WORKS: A dummy field full of X's is defined. The number of X's that show up in each line of output is a function of the warehouse size. The SUBSTR function is used to truncate the dummy field to size.

- WARNING: A SQL error will be generated if the warehouse size (after processing) used in the SUBSTR function exceeds the length of the dummy field.

Making Charts using SQL

```
SELECT   CITY_NUM
        ,WHSE_SIZE
        ,(WHSE_SIZE - 90) * 2
        ,SUBSTR('XXXXXXXXX*XXXXXXXXX*XXXXXXXXX*XXXXXXXXX*'
                ,1,(WHSE_SIZE - 90)*2)
FROM     WHSE_TABLE
WHERE    WHSE_NUM = 100
ORDER BY CITY_NUM;
```

```
CITY  WHSE
NUM   SIZE
----  ----   ----  ----------------------------------------
   0    100    20  XXXXXXXXX*XXXXXXXXX*
   1    101    22  XXXXXXXXX*XXXXXXXXX*XX
   2    102    24  XXXXXXXXX*XXXXXXXXX*XXXX
   3    103    26  XXXXXXXXX*XXXXXXXXX*XXXXXX
   4    104    28  XXXXXXXXX*XXXXXXXXX*XXXXXXXX
   5    105    30  XXXXXXXXX*XXXXXXXXX*XXXXXXXXX*
   6    106    32  XXXXXXXXX*XXXXXXXXX*XXXXXXXXX*XX
   7    107    34  XXXXXXXXX*XXXXXXXXX*XXXXXXXXX*XXXX
   8    108    36  XXXXXXXXX*XXXXXXXXX*XXXXXXXXX*XXXXXX
   9    109    38  XXXXXXXXX*XXXXXXXXX*XXXXXXXXX*XXXXXXXX
```

Number and Character Manipulation

NUMBER DISSECTION

- ENGLISH MEANING: Get the second and third right-most digits of the Product number in both character and numeric format from the SALE table where the Product number is between 2118 and 2121 and the sale was made before 4 a.m.

- PROCESSING SEQUENCE: Subtract the number from itself (first divided, then multiplied by 1,000) then divide what's left by 10. Because the field being processed is an INTEGER value the two division statements remove right-most values. If the field were of Decimal or Float type we would have to use the INTEGER function to convert it to integer before starting.

CHAR & VARCHAR DISSECTION

- LEFT-MOST CHARACTERS: Use the plain SUBSTR function to get the left most characters in a text field or string.

- RIGHT-MOST CHARACTERS: Use the SUBSTR and LENGTH functions together to get the right most characters from a field or text string. If the input is variable length, note that the start of the sub-string will differ from one line to the next depending upon the length of the field being processed.

- WARNING, DATA TOO SHORT: If the input field is shorter than what the SUBSTR expects to find a SQL error will result.

- WARNING, NULL DATA: A field which is defined as allowing nulls will report the same length as one defined as not null which has the same data (e.g: if both contain "ABC" both will be of length 3). However, if the null field contains a null value, the length of that field will also be null.

  OUTPUT FIELDS:
  1. Whole field.
  2. 1st five bytes.
  3. Whole field, starting at byte five.
  4. Length of field.
  5. Right most byte.
  6. Right most two bytes.
  7. Right most three bytes.

Number and Character Manipulation

Number Dissection

```
SELECT   PROD_NUM
        ,PROD_NUM/ 10
        ,PROD_NUM - (PROD_NUM/1000*1000)
        ,(PROD_NUM - (PROD_NUM/1000*1000) ) / 10
FROM     SALE_TABLE
WHERE    PROD_NUM  BETWEEN 2118 AND 2121
  AND    SALE_TIME < '04.00.00'
ORDER BY 1;
```

```
          <== TRUNCATION ==>
PROD_NUM  Right   Left  Both
--------  -----  -----  ----
    2118    211    118    11
    2121    212    121    12
```

Char & Varchar Dissection

```
SELECT   PROD_PADDING
        ,SUBSTR(PROD_PADDING,1,5)
        ,SUBSTR(PROD_PADDING,5)
        ,LENGTH(PROD_PADDING)
        ,SUBSTR(PROD_PADDING,LENGTH(PROD_PADDING),1)
        ,SUBSTR(PROD_PADDING,LENGTH(PROD_PADDING)-1,2)
        ,SUBSTR(PROD_PADDING,LENGTH(PROD_PADDING)-2,3)
FROM     PROD_TABLE
WHERE    PROD_NUM < 2005;
```

Date Manipulation

NOTES

- ENGLISH MEANING: Demonstrate those kinds of date addition which do and/or do not work.

- USING DAYS FUNCTION (NOT WORK): One can NOT use the DAYS function to reliably add dates and then derive a new date. It is not that it will not work, it is that the answer will be unintuitive, and probably not what the user wanted. There are two causes:

  First, the DAYS function does not count the days from day zero, but rather from the date '0001-01-01'. To illustrate, adding the first day in year one to the itself (using the DAYS function) will result in the second day in year one, not the second day of the second month in year two.

  Second, the DAYS function takes no note of the differing length of various months. To illustrate, adding the 1st of February in year one to itself (using the DAYS function) will result in the day part being off because January has more days than February.

- MULTIPLYING DATES (NOT WORK): This also can NOT be done using the DAYS function.

- USING A DATE DURATION: DB2/2 does not support the adding of DATE field types. To do the addition one of the fields can to be converted into a DATE DURATION equivalent. This is done by separating out the years, months, and days and then combining them into a DECIMAL number.

- USING LABELED DURATIONS: The date to be added can be converted into so many years, months, and days, which can then be added in turn.

- NOTE, DATA IS NOT IN DATABASE: The date values shown here do not actually exist in the sample database. They were put in, and then rolled out, just to illustrate the problems shown here.

START & END OF CURRENT MONTH

- The following SQL statement illustrates how to get the first and last days of the current month using date manipulation. To get the START of the month we subtract the number of days passed in the month from the current date (plus one day). To get the END of the month we first add a month, and then subtract the number of days passed in the (next) month.

```
SELECT CURRENT DATE
      ,CURRENT DATE + 1 DAY   - DAY(CURRENT DATE)           DAYS
      ,CURRENT DATE + 1 MONTH - DAY(CURRENT DATE + 1 MONTH) DAYS
FROM   CITY_TABLE
WHERE  CITY_NUM = 9;


CURRENT-DT  MNTH-START   MNTH-END
----------  ----------  ----------
1993-05-13  1993-05-01  1993-05-31
```

Date Addition

```
SELECT SALE_DATE
      ,DAYS(SALE_DATE)
      ,DAYS(SALE_DATE) + DAYS(SALE_DATE)
      ,DATE(DAYS(SALE_DATE) + DAYS(SALE_DATE))
      ,SALE_DATE
         + YEAR (SALE_DATE) YEARS
         + MONTH(SALE_DATE) MONTHS
         + DAY  (SALE_DATE) DAYS
      ,SALE_DATE + DECIMAL (
           YEAR (SALE_DATE) * 10000
         + MONTH(SALE_DATE) * 100
         + DAY  (SALE_DATE) * 1  ,8,0)
FROM   SALE_TABLE
WHERE  PROD_NUM = 2101;
```

SALE_DATE	Days*1	Days*2	Add DAYS	Add Labels	Add Duratn
0001-01-01	1	2	0001-01-02	0002-02-02	0002-02-02
0001-01-02	2	4	0001-01-04	0002-02-04	0002-02-04
0001-02-01	32	64	0001-03-05	0002-04-02	0002-04-02
0001-03-01	60	120	0001-04-30	0002-06-02	0002-06-02
0002-01-01	366	732	0003-01-02	0004-02-02	0004-02-02
1000-01-01	364878	729756	1999-01-02	2000-02-02	2000-02-02
1900-01-01	693596	1387192	3799-01-01	3800-02-02	3800-02-02
1994-01-01	727929	1455858	3987-01-02	3988-02-02	3988-02-02
1994-02-28	727987	1455974	3987-04-28	3988-05-26	3988-05-26
2000-01-01	730120	1460240	3999-01-01	4000-02-02	4000-02-02

Julian Date Conversion

SELECTING

- JULIAN DATE: This is a date format of type YYYYNNN where YYYY is the year and NNN is the day number in the year starting from 1 on January 1st. In some situations the year is only two bytes long.

- SELECTING A JULIAN DATE: There is no JULIAN DATE output format in DB2/2. What one can do is convert a date value into a number which represents a Julian Date using the following algorithm: First get the date for the related "start of year", Next, subtract from the DAYS of the current date - the DAYS of the start of the year. This will obtain the number of days that have passed since the start of the year. Add this value to the number of years (times 1,000) to get the Julian Date.

PREDICATE USAGE

- JULIAN INPUT: The DATE function can be used to convert a character representation of a date in the format YYYYNNN where YYYY is the year and NNN is the Julian day number into a valid DB2/2 date. This can then be used in predicates as shown on the next page.

HISTORICAL TRIVIA

- REAL JULIAN DATE: Most computer programmers think of the "Julian Date" as a date format. A more correct use of the term "Julian Date" refers to the current date according to the calendar as defined by Julius Caesar. In the twentieth century this means that one must subtract 15 days from the relevant Gregorian date (e.g: 1994-01-22 becomes 1994-01-07).

- JULIAN DAY: A Julian day is used in Astronomy to relate ancient Babylonian and Alexandrian observations with their modern equivalents. Each day belongs to a great cycle of 7,980 years which began on midday (Greenwich Mean Time), January the 1st, 4,713 B.C. For example, the evening of January 3rd, 1993 (Gregorian Calendar, using G.M.T.) equals the Julian day: 2,448,991.

SQL OUTPUT FIELDS

1. Date field.
2. Start of year (for given date).
3. DAYS value, for date field.
4. DAYS value, for start of year.
5. Difference, 1st DAYS value minus second (i.e: day of year).
6. Julian Date, for start of year value.
7. Julian Date, for date field.

Julian Date Conversion

```
SELECT SALE_DATE
      ,SALE_DATE + 1 DAY   - DAY(SALE_DATE)DAYS
                 + 1 MONTH - MONTH(SALE_DATE)MONTHS
      ,DAYS(SALE_DATE)
      ,DAYS(SALE_DATE + 1 DAY   - DAY(SALE_DATE)DAYS
                      + 1 MONTH - MONTH(SALE_DATE)MONTHS)
      ,DAYS(SALE_DATE)
      -DAYS(SALE_DATE + 1 DAY   - DAY(SALE_DATE)DAYS
                      + 1 MONTH - MONTH(SALE_DATE)MONTHS)
      ,YEAR(SALE_DATE) * 1000
      ,YEAR(SALE_DATE) * 1000
      +DAYS(SALE_DATE)
      -DAYS(SALE_DATE + 1 DAY   - DAY(SALE_DATE)DAYS
                      + 1 MONTH - MONTH(SALE_DATE)MONTHS)
      +1
FROM   SALE_TABLE
WHERE  PROD_NUM < 2200
  AND  CUST_NUM = 9100
  AND  SALE_DATE BETWEEN DATE('1993065') AND DATE('1995077')
ORDER BY 1;
```

```
SALE_DATE   Start-Year  DAYS#1  DAYS#2  Dif  Julian   Julian
----------  ----------  ------  ------  ---  -------  -------
1993-12-31  1993-01-01  727928  727564  364  1993000  1993365
1994-01-01  1994-01-01  727929  727929    0  1994000  1994001
1994-01-31  1994-01-01  727959  727929   30  1994000  1994031
1994-02-01  1994-01-01  727960  727929   31  1994000  1994032
1994-02-28  1994-01-01  727987  727929   58  1994000  1994059
1994-03-01  1994-01-01  727988  727929   59  1994000  1994060
```

Week Processing in SQL

GET FRIDAY DATA

- CHOOSE A VALUE: First, choose a date value that is on the target day (eg: 1993-04-02 is a Friday).

- USING IN SQL: Next, in the SQL statement WHERE clause, subtract the DAYS of the target date from the DAYS of the current row and compare the result to the same, first divided, then multiplied, by seven. If the result matches, the current row is a Friday.

GROUP BY DAY OF WEEK

1. CHOOSE A STANDARD: The SQL described here will return a number which represents the day of the week. Before beginning, one must decide on a standard way to number days, in this example we have defined Sunday as day 1, Monday as day 2, etc.

2. CHOOSE A WEEK START: Next one must choose a date value which is on the first day of the week (i.e: Sunday), and which is below ALL of the matching date values in the table (e.g: 1993-01-17). This must be done to ensure that one does not get a mixture of negative and positive week days.

3. CREATE A VIEW: Next, create a VIEW which contains all of the columns in the table, plus a new column which has the day-of-week value. The view is needed because DB2/2 will not do a GROUP BY on a derived field.

4. USE IN SQL: Select from the view as you would the original table.

SELECT WEEK NUMBERS

- The following SQL will assign week numbers starting at 1 on '1993-12-26', which is a Sunday. As above, to avoid negative week numbers, choose a literal value which is less than the lowest data value likely to be encountered.

```
SELECT ((DAYS(SALE_DATE) - DAYS('1993-12-26')) / 7 ) + 1
       ,SALE_DATE, PROD_NUM, SALE_TIME
FROM    SALE_TABLE A
WHERE   PROD_NUM  <  2220
  AND   SALE_TIME < '04.00.00'
ORDER BY 1,2,3;
```

ANSWER, LAST SQL STMT ON NEXT PAGE

WEEK_DAY	SUM	AVG	AVG	COUNT
1	10120	50	99	400
2	9920	49	98	400
3	9720	48	97	400
4	6000	50	96	240
5	8040	50	99	320
6	7880	49	98	320
7	7720	48	97	320

Week Processing in SQL

Get Friday Data

```
SELECT COUNT(*)
FROM   SALE_TABLE
WHERE  DAYS(SALE_DATE) - DAYS('1993-04-02') =
     ((DAYS(SALE_DATE) - DAYS('1993-04-02'))/7) * 7;
```

Group by Day of Week

```
CREATE VIEW SALE_WEEK
  (PROD_NUM       ,SALE_DATE   ,SALE_TIME
  ,NUM_SOLD       ,ITEM_COST   ,CUST_NUM
  ,SALE_PADDING   ,WEEK_DAY)
AS SELECT
   PROD_NUM       ,SALE_DATE   ,SALE_TIME
  ,NUM_SOLD       ,ITEM_COST   ,CUST_NUM
  ,SALE_PADDING
  ,((DAYS(SALE_DATE) - DAYS('1993-01-17')) -
  (((DAYS(SALE_DATE) - DAYS('1993-01-17'))/7) * 7)) + 1
FROM SALE_TABLE;
COMMIT;
```

```
SELECT   WEEK_DAY
        ,SUM(NUM_SOLD)
        ,AVG(NUM_SOLD)
        ,MAX(NUM_SOLD)
        ,COUNT(*)
FROM     SALE_WEEK
GROUP BY WEEK_DAY;
```

Boolean Logic in SQL

NOTES

- INTRODUCTION: It is possible to include inside SQL statements predicates which cause the logic of the statement (and consequently the answer that it gets) to change depending upon outside constraints.

  On this page this capability is illustrated by checking the current time. Depending on whether it is the morning or the afternoon, a different answer is obtained. One could equally well check the contents of a program variable.

- SUBQUERY: The subquery part of the statement is only invoked before midday.

- JOIN: After midday the join (i.e: the first part of the UNION) is done, before midday it is not.

- EFFICIENCY: When used with SUBQUERIES this way of writing SQL is usually cost free. This is because the literal value is checked before the subquery is invoked. In fact, if the only alternative is to have two different statements, this may be the slightly faster method because special DB2/2 code like sequential detect will kick in faster when only one statement is being repetitively invoked.

  When used with JOINS this method usually incurs a cost. At the very least, DB2/2 has to look at both sides of the UNION. More importantly, if the data is required in a certain order (which is almost always the case), then the use of a UNION will invariably induce a row sort which may not otherwise have happened.

Boolean Logic in SQL

Subquery

```
SELECT   COUNT(*)
FROM     PROD_TABLE A
WHERE    CURRENT TIME > '12.00.00'
   OR    EXISTS
        (SELECT *
         FROM   SALE_TABLE B
         WHERE  A.PROD_NUM = B.PROD_NUM);
```

Join

```
SELECT   A.*
        ,B.*
FROM     CITY_TABLE A
        ,CITY_TABLE B
WHERE    CURRENT TIME >  '12.00.00'
  AND    A.CITY_NUM = B.CITY_NUM
UNION ALL
SELECT   A.*
        ,A.*
FROM     CITY_TABLE A
WHERE    CURRENT TIME <= '12.00.00';
```

Doing Multiple Counts in one Pass

NOTES

- INTRODUCTION: The second SQL statement on the next page obtains multiple counts of the same field in a single pass. It does this by reducing the target row value to either a one, or a zero, depending upon whether it is above or below the break point (ie: 2222). The SUM function is then used to count the number of ones and zeros.

- ONLY FOR NUMERIC FIELDS: This trick only works on numeric fields.

- 1ST SQL: This statement illustrates how the INTEGER function can be used to get either a one or a zero depending upon the whether the input value is above or below a given break point, in this case the value 2222.

2ND SQL- FIELDS

- COUNT: Count of all matching rows in table.

- COUNT, ROWS < 2222: Count of all matching rows with a PROD_NUM < 2222. The two INTEGER functions convert any value less than 2222 to a one and any value greater than to a zero. The SUM function is then used to obtain a count.

- COUNT, ROWS >= 2222: Count of all matching rows with a PROD_NUM >= 2222. This is the same as the prior column except that a "minus one" is used to convert ones to zero and vice versa. The SUM is then used to get a count.

- PERCENT, ROWS < 2222: Percentage of all matching rows with a PROD_NUM < 2222. The result of the SUM multiplied by 100 and then divided by the number of total matching rows (i.e: the COUNT(*)) to get the percentage.

- PERCENT, ROWS >= 2222: Percentage of all matching rows with a PROD_NUM >= 2222.

Doing Multiple Counts in one Pass

Examples - Integer Function

```
SELECT INTEGER(1 / (INTEGER(0000/2222) +1))
      ,INTEGER(1 / (INTEGER(1111/2222) +1))
      ,INTEGER(1 / (INTEGER(2221/2222) +1))
      ,INTEGER(1 / (INTEGER(2222/2222) +1))
      ,INTEGER(1 / (INTEGER(2223/2222) +1))
      ,INTEGER(1 / (INTEGER(3333/2222) +1))
FROM   WP_TABLE
WHERE  CITY_NUM = 3
  AND  WHSE_NUM = 100
  AND  PROD_NUM = 2000;
```

```
INTEGER(0000/2222)  =>  0       1/(0+1)  =>  1
INTEGER(1111/2222)  =>  0       1/(0+1)  =>  1
INTEGER(2221/2222)  =>  0       1/(0+1)  =>  1
INTEGER(2222/2222)  =>  1       1/(1+1)  =>  0
INTEGER(2223/2222)  =>  1       1/(1+1)  =>  0
INTEGER(3333/2222)  =>  1       1/(1+1)  =>  0
```

Multiple Counts

```
SELECT COUNT(*)
      ,SUM(     INTEGER(1 / (INTEGER(PROD_NUM/2222)+1)))
      ,SUM(1 -  INTEGER(1 / (INTEGER(PROD_NUM/2222)+1)))
      ,    (SUM(INTEGER(1 / (INTEGER(PROD_NUM/2222)+1)))*100) / COUNT(*)
      ,100-(SUM(INTEGER(1 / (INTEGER(PROD_NUM/2222)+1)))*100) / COUNT(*)
FROM   WP_TABLE
WHERE  CITY_NUM < 3;
```

```
COUNT(*)          =>   3600
COUNT   <  2222   =>    984
COUNT   >= 2222   =>   2616
PERCENT <  2222   =>     27%
PERCENT >= 2222   =>     73%
```

Char & Varchar Processing Differences

NOTES

- WARNING: Sometimes a pair of VARCHAR and CHAR columns may seem to have that same value when one is compared to the other, but may return different results when accessed separately using the same LIKE predicate. This is NOT a bug, and is (fortunately) very rare.

- WHEN EQUAL VALUES ARE NOT EQUAL: Sometimes, DB2/2 will say that two fields are EQUAL when, internally, they are not. In particular, when doing an EQUAL check on a "less-than-full" VARCHAR field, DB2/2 will assume that it has trailing blanks (as if it were a CHAR field).

- WHEN THE PROBLEM OCCURS: The different answers shown on the next page occur because DB2/2 does NOT use implicit blanks in a LIKE predicate (unlike in a EQUAL predicate).

- RECOMMENDATIONS: In general, never compare a VARCHAR field against a CHAR field. If you do, try to avoid using a LIKE predicate on either of the fields. Also, be careful when using a LIKE " %" check on a VARCHAR field. If the field actually has a leading blank, a row will be found, if not, it wont.

- DEFAULT COLUMNS, SAME RESULT: Defining the two data columns as being not null with default does not affect the result.

SAMPLE SQL NOTES

- DATA: The next page has some DDL and DML that creates a table with one row of data. Internally, the first column/row contains five blank bytes; the second contains no data and has a length of zero. Neither value is null.

- EQUAL PREDICATE: When an EQUAL check is done on either column the result is always the same. DB2/2 implicitly uses blanks to pad out the VARCHAR field. Note that this occurs whenever an EQUAL predicate is used; to DB2/2 the check: "A" = "A " is always true.

- LIKE PREDICATE: When a LIKE check is done the result differs depending upon the column type being processed. In a LIKE predicate DB2/2 does NOT implicitly pad out the VARCHAR field with blanks. Because there is no actual blank value in the second column, no rows are found.

Char & Varchar Processing Differences

```
CREATE TABLE FRED
(COL1     CHAR(5) NOT NULL
,COL2  VARCHAR(5) NOT NULL);
COMMIT;

INSERT INTO FRED VALUES ('','');
COMMIT;
```

--- 1 Row Returned ---

```
SELECT *
FROM   FRED
WHERE  COL1 = COL2;
```

--- 1 Row Returned ---

```
SELECT *
FROM   FRED
WHERE  COL1 = ' ';
```

--- 1 Row Returned ---

```
SELECT *
FROM   FRED
WHERE  COL1 LIKE ' %';
```

--- 1 Row Returned ---

```
SELECT *
FROM   FRED
WHERE  COL2 = ' ';
```

--- No Rows Found! ---

```
SELECT *
FROM   FRED
WHERE  COL2 LIKE ' %';
```

Get data for the MIN / MAX row

NOTES

- ENGLISH MEANING: Select the row(s) in the SALE table which have the minimum sale time for each particular Product number less than 2005.

- If only the TIME field had been wanted a simple MAX query would have sufficed. Because all of the columns (in the row) are to be retrieved, two logical tables need to be processed. One is used to get the row with the MAX time (for the current Product number), the other joins to it by common time.

- WARNING, #ROWS MATCHING: If there is more than one row with the same MAX time (for a particular Product number) all will be returned using this query.

- ALTERNATIVE, ORDERED CURSOR: One can DECLARE an ascending or descending CURSOR on the Product number field and then stop fetching after the first row. See page 104 for more information.

MIN SQL STATEMENT, ANSWER

PROD_NUM	SALE_DATE	SALE_TIME	NUM_SOLD	ITEM_COST	CUST_NUM	SALE_PAD
2001	1994-01-01	03.01.00	90	.00	9100	SALE
2002	1994-01-01	03.02.00	90	.00	9200	SALE
2003	1994-01-01	03.03.00	90	.00	--------	SALE
2004	1994-01-01	03.04.00	90	.00	--------	SALE

MAX SQL STATEMENT, ANSWER

PROD_NUM	SALE_DATE	SALE_TIME	NUM_SOLD	ITEM_COST	CUST_NUM	SALE_PAD
2001	1994-01-01	12.01.00	0	.00	100	SALE
2002	1994-01-01	12.02.00	0	.00	200	SALE
2003	1994-01-01	12.03.00	0	.00	--------	SALE
2004	1994-01-01	12.04.00	0	.00	--------	SALE

Get data for the MIN / MAX row

MIN

```
SELECT   *
FROM     SALE_TABLE A
WHERE    A.PROD_NUM < 2005
  AND    A.SALE_TIME =
        (SELECT MIN(B.SALE_TIME)
         FROM   SALE_TABLE B
         WHERE  A.PROD_NUM  = B.PROD_NUM);
```

MAX

```
SELECT   *
FROM     SALE_TABLE A
WHERE    A.PROD_NUM < 2005
  AND    A.SALE_TIME =
        (SELECT MAX(B.SALE_TIME)
         FROM   SALE_TABLE B
         WHERE  A.PROD_NUM  = B.PROD_NUM);
```

Alternative, Ordered Cursor

```
DECLARE FRED CURSOR FOR
SELECT   *
FROM     SALE_TABLE A
ORDER BY PROD_NUM DESC;
```

Get the 10 MIN / MAX rows

NOTES

- ENGLISH MEANING: Select all the columns for the 10 rows in the SALE table with the minimum Product number(s).

- TOP QUERY: Every row in the SALE table will be processed. For each row, a check will be made to see if there is less than 10 other rows with a lower Product number.

- BOTTOM QUERY: The UPPER part of the query selects all of the columns from the target table. The MIDDLE part of the query contains a set of 10 nested subqueries which gets one up to the tenth Product number value. The LOWER part of the query is used to eliminate unwanted rows in the cases where are more than 10 rows are encompassed by the first 10 Product numbers.

  Because there is a ascending index on Product number in the SALE table, this query will be very efficient. This would be especially true if the SALE table were large. For MAX queries one needs to have a corresponding descending index.

- WARNING, SAMPLE SQL: If there are multiple rows that share the same 10 minimum values it may occur that these two queries will return more than 10 rows.

- ALTERNATIVE, ORDERED CURSOR (WARNING): One can DECLARE an ascending CURSOR on the Product number field and then stop fetching after the first 10 rows. However, one should NEVER do this, unless the ORDER BY fields represent a unique row. You should always fetch at least 11 rows to make sure that the 11th does not have the same value as the 10th. If it does you should either inform the user, or keep fetching until the value changes.

ANSWER, BOTH QUERIES ON NEXT PAGE

PROD_NUM	SALE_DATE	SALE_TIME	NUM_SOLD	ITEM_COST	CUST_NUM	SALE_PAD
2001	1994-01-01	03.01.00	90	.00	9100	SALE
2001	1994-01-01	04.01.00	80	.00	8100	SALE
2001	1994-01-01	05.01.00	70	.00	7100	SALE
2001	1994-01-01	06.01.00	60	.00	6100	SALE
2001	1994-01-01	07.01.00	50	.00	5100	SALE
2001	1994-01-01	08.01.00	40	.00	4100	SALE
2001	1994-01-01	09.01.00	30	.00	3100	SALE
2001	1994-01-01	10.01.00	20	.00	2100	SALE
2001	1994-01-01	11.01.00	10	.00	1100	SALE
2001	1994-01-01	12.01.00	0	.00	100	SALE

Get the 10 MIN / MAX rows

MIN 10, Inefficient

```
SELECT *
FROM   SALE_TABLE A
WHERE  10 >
      (SELECT COUNT(*)
       FROM   SALE_TABLE B
       WHERE  A.PROD_NUM > B.PROD_NUM);
```

MIN 10, Efficient

```
SELECT *
FROM   SALE_TABLE A
WHERE  A.PROD_NUM  <=
      (SELECT MIN(PROD_NUM) FROM SALE_TABLE WHERE PROD_NUM >
      (SELECT MIN(PROD_NUM) FROM SALE_TABLE WHERE PROD_NUM >
      (SELECT MIN(PROD_NUM) FROM SALE_TABLE WHERE PROD_NUM >
      (SELECT MIN(PROD_NUM) FROM SALE_TABLE WHERE PROD_NUM >
      (SELECT MIN(PROD_NUM) FROM SALE_TABLE WHERE PROD_NUM >
      (SELECT MIN(PROD_NUM) FROM SALE_TABLE WHERE PROD_NUM >
      (SELECT MIN(PROD_NUM) FROM SALE_TABLE WHERE PROD_NUM >
      (SELECT MIN(PROD_NUM) FROM SALE_TABLE WHERE PROD_NUM >
      (SELECT MIN(PROD_NUM) FROM SALE_TABLE WHERE PROD_NUM >
      (SELECT MIN(PROD_NUM) FROM SALE_TABLE))))))))))
   AND 10 >
      (SELECT COUNT(*)
       FROM   SALE_TABLE B
       WHERE  A.PROD_NUM > B.PROD_NUM);
```

Alternative, Ordered Cursor

```
DECLARE FRED CURSOR FOR
SELECT   *
FROM     SALE_TABLE A
ORDER BY PROD_NUM ASC;
```

Selecting Most Recent Data

NOTES

- ENGLISH MEANING: Select all the columns for those rows in the SALE table which refer to a sale which occurred within 10 minutes of the most recent sale.

- DESIGN: Convert the sale date and time to a timestamp. Get the max timestamp value. Subtract 10 minutes from the max value. Select all rows which have a timestamp greater than or equal to the sub-query result.

VALID TIMESTAMP FUNCTION FORMATS

```
1.  TIMESTAMP('YYYY-MM-DD-HH.MM.SS.NNNNNN')
2.  TIMESTAMP('YYYY-MM-DD-HH.MM.SS')
3.  TIMESTAMP('YYYY-MM-DD','HH.MM.SS')
4.  TIMESTAMP('YYYYMMDDHHMMSS')
5.  TIMESTAMP(DATE('YYYYNNN'),'HH.MM.SS')
6.  TIMESTAMP(CURRENT TIMESTAMP)
7.  TIMESTAMP(CURRENT DATE,CURRENT TIME)
```

Selecting Most Recent Data

```
SELECT  *
FROM    SALE_TABLE
WHERE   TIMESTAMP(SALE_DATE,SALE_TIME) >= ANY
        (SELECT MAX(TIMESTAMP(SALE_DATE,SALE_TIME))
                    - 10 MINUTES
         FROM   SALE_TABLE)
ORDER BY SALE_DATE DESC
        ,SALE_TIME DESC;
```

PROD#	SALE_DATE	SALE_TIM	#SOLD	COST	CUST#	PADDING
2028	03/01/1994	12:08:00	?	?	?	SALE
2027	03/01/1994	12:07:00	?	?	?	SALE
2026	03/01/1994	12:06:00	?	20.00	?	SALE
2025	03/01/1994	12:05:00	?	20.00	?	SALE
2024	03/01/1994	12:04:00	0	20.00	?	SALE
2023	03/01/1994	12:03:00	0	20.00	?	SALE
2022	03/01/1994	12:02:00	0	20.00	200	SALE
2021	03/01/1994	12:01:00	0	20.00	100	SALE

Complicated Subqueries

NOTES

- SAME QUERIES: Both queries are logically the same.

- ENGLISH MEANING: Get a list of distinct Product Types from the Product table where the Product number is less 2200, and there exists more than four sales or products of that type to customers with a non-null customer number, and there are more than 8 Product numbers.

- JOIN IN SUBQUERY: In the subquery the PROD table is first accessed to get a complete list of product numbers for the current product type. Next the PROD table in joined to the SALE table by common product number so the count can be obtained.

- GROUP BY IN SUBQUERY: This is the same as the first query except that count is checking the subquery (using the HAVING clause) instead of being passed back to the top part of the query for evaluation.

- WHICH IS FASTER: They are probably both about the same. Both suffer from a common problem; the subquery will not stop counting until all matching rows a processed. Ideally, it should stop after four rows have been found.

- LESS EFFICIENT VERSION: The following SQL statement is probably less efficient that either of the two given on the next page. This is because it will do the EXISTS check before having eliminated those rows where there are less than 8 rows per Product type.

```
SELECT   PROD_TYPE
FROM     PROD_TABLE A
WHERE    PROD_NUM < 2200
   AND   EXISTS
        (SELECT COUNT(*)
         FROM   PROD_TABLE B
               ,SALE_TABLE C
         WHERE  A.PROD_TYPE = B.PROD_TYPE
         AND    B.PROD_NUM  = C.PROD_NUM
         AND    C.CUST_NUM IS NOT NULL
         HAVING COUNT(*) > 4)
GROUP BY PROD_TYPE
HAVING   COUNT(*) > 8;
```

Complicated Subqueries

Join in Subquery

```
SELECT   PROD_TYPE
FROM     PROD_TABLE A
WHERE    PROD_NUM < 2200
GROUP BY PROD_TYPE
HAVING   COUNT(*) > 8
   AND   4 <
         (SELECT COUNT(*)
          FROM   PROD_TABLE B
                ,SALE_TABLE C
          WHERE  A.PROD_TYPE = B.PROD_TYPE
          AND    B.PROD_NUM  = C.PROD_NUM
          AND    C.CUST_NUM IS NOT NULL);
```

Group By in Subquery

```
SELECT   PROD_TYPE
FROM     PROD_TABLE A
WHERE    PROD_NUM < 2200
GROUP BY PROD_TYPE
HAVING   COUNT(*) > 8
   AND   EXISTS
         (SELECT COUNT(*)
          FROM   PROD_TABLE B
                ,SALE_TABLE C
          WHERE  A.PROD_TYPE = B.PROD_TYPE
          AND    B.PROD_NUM  = C.PROD_NUM
          AND    C.CUST_NUM IS NOT NULL
          HAVING COUNT(*) > 4);
```

Using 2 Distinct Stmts in a Query

NOTES

- ENGLISH MEANING: Get a list of those cities in the WP table where all of the warehouses in the city store more than 22 distinct Product numbers.

- DB2/2 LIMIT: There is a DB2/2 limit of one DISTINCT per query (unions aside). To get around this limit one can use a double-nested subquery to resolve one, or more, distinct questions.

- NOTE, ERROR: Due to a bug in DB2/2, the last statement gave the wrong result, it should work.

- WARNING: In the second query the "implied" DISTINCT statement (in bold) is logically the same as the true DISTINCT in the first query. This SQL is not easy to write correctly so be careful.

- TOP QUERY: Because there is a DISTINCT statement in the HAVING part of the query we are unable to have the same in the SELECT list. This means that we can select all the rows which match, but we do not know the value that matched.

- MIDDLE QUERY: The second DISTINCT has been replaced with a double-nested subquery call. The top half of the subquery gets a count of rows in the city. The lower part of the subquery qualifies this count to only include one row per Product number. This kind of query can be very difficult to write because, in order to get the correct answer, one must know exactly which fields make up the unique key of the table and how they relate to the question being asked.

- USING VIEW: A view was created (see below) which gets the number of distinct values per City (using a GROUP BY statement). This view was then referred to by another GROUP BY (in the statement shown). What DB2/2 did was first resolve the view into a work file, then apply second group by on that.

WP_VIEW DDL

```
CREATE VIEW WP_VIEW
(CITY_NUM ,NMBR_PROD ,SUM_QTY ,NMBR_ROWS)
AS SELECT  CITY_NUM  ,COUNT(DISTINCT PROD_NUM)
          ,SUM(QTY)  ,COUNT(*)
FROM       WP_TABLE
GROUP BY   CITY_NUM;
COMMIT;
```

ANSWER, 2ND & 3RD QUERIES

```
CITY_NUM  COUNT(*)  COUNT DIS
========  ========  =========
       0      1200        300
       1      1200        300
       2      1200        300
       3      1200        300
       4      1200        300
       5      1200        300
       6      1200        300
       7      1200        300
       8      1200        300
       9      1200        300
```

Using 2 Distinct Stmts in a Query

One Distinct

```
SELECT   CITY_NUM
        ,COUNT(*)
--      ,COUNT(DISTINCT PROD_NUM)
FROM     WP_TABLE
GROUP BY CITY_NUM
HAVING   COUNT(DISTINCT PROD_NUM) > 22;
```

Two Distincts

```
SELECT   CITY_NUM
        ,COUNT(*)
        ,COUNT(DISTINCT PROD_NUM)
FROM     WP_TABLE A
WHERE    22 <
        (SELECT COUNT(*)
         FROM   WP_TABLE B
         WHERE  A.CITY_NUM = B.CITY_NUM
           AND  NOT EXISTS
               (SELECT *
                FROM   WP_TABLE C
                WHERE  B.CITY_NUM = C.CITY_NUM
                  AND  B.PROD_NUM = C.PROD_NUM
                  AND  B.WHSE_NUM < C.WHSE_NUM))
GROUP BY CITY_NUM;
```

Using View

```
SELECT   CITY_NUM
        ,AVG(NMBR_ROWS)
        ,AVG(NMBR_PROD)
FROM     WP_VIEW
GROUP BY CITY_NUM
HAVING   AVG(NMBR_PROD) > 22;
```

Creating test data using SQL

NOTES

- DESIGN: By repetitively joining a table to itself (and doing a Cartesian product in the process) one can very quickly generate a large amount of somewhat generalized sample data. The example on this page was used to generate the data in the SALE table which is used in this document. For clarity, the update statements have been omitted.

- PERFORMANCE ENHANCEMENTS: Two things will make the select/insert process run a lot faster: Firstly, add a suitable index, preferably unique, to the small input data table. This will make the join a lot more efficient. Secondly, lock the output table in exclusive mode before doing each block of inserts.

- WARNING, LARGE UOW: Don't run large units of work, commit occasionally. DB2/2, like any good DBMS, works best with small units of work.

DDL, BASE TABLE

```
CREATE TABLE BASE_TABLE
  (U1     SMALLINT  NOT NULL,
   U2     CHAR(01)  NOT NULL,
   U3     CHAR(01)  NOT NULL,
   D1     SMALLINT  NOT NULL,
   D2     CHAR(01)  NOT NULL,
   D3     CHAR(01)  NOT NULL)
   IN DBGBTEST.TSGBBASE;
```

SOURCE DATA, BASE TABLE

```
U1 U2 U3 D1 D2 D3
-- -- -- -- -- --
0  0  A  9  9  J
1  1  B  8  8  I
2  2  C  7  7  H
3  3  D  6  6  G
4  4  E  5  5  F
5  5  F  4  4  E
6  6  G  3  3  D
7  7  H  2  2  C
8  8  I  1  1  B
9  9  J  0  0  A
```

Creating test data using SQL

DDL

```
CREATE TABLE SALE_TABLE
  (PROD_NUM         INTEGER       NOT NULL,
   SALE_DATE        DATE                  ,
   SALE_TIME        TIME                  ,
   NUM_SOLD         INTEGER               ,
   ITEM_COST        DECIMAL(7,2)          ,
   CUST_NUM         INTEGER               ,
   SALE_PADDING     CHAR(63)              )
   IN DBGBTEST.TSGBSALE;
```

Insert DML

```
INSERT INTO SALE_TABLE
SELECT 2000 + A.U1*100 + B.U1*10 + C.U1
      ,DATE('1994-01-01') + B.U1 MONTHS - A.U1 DAYS
      ,TIME('12.00.00')   - D.U1 HOURS  + C.U1 MINUTES
      ,D.U1*10 + A.U1
      ,A.U1*100 + B.U1*10 + D.U1/100
      ,D.U1*1000 + C.U1*100
      ,'SALE'
FROM  BASE_TABLE A
     ,BASE_TABLE B
     ,BASE_TABLE C
     ,BASE_TABLE D
WHERE B.U1 < 3
  AND C.U1 BETWEEN 1 AND 8;
```

Using SQL to generate DDL / DML / SQL

NOTES

- INTRODUCTION: SQL can be used to generate other SQL, DML, or DDL statements. This can be a fast way to get certain tasks done.

- NOTE, NO CONCAT: Because DB2/2 lacks a CONCAT function, it is not possible to get the output in a single field, instead one must print all of the output fields together in a "flat file".

- DDL EXAMPLE: The query will return a DROP TABLE statement for every table created by the user. Some sample output follows, note that the gap between the name and the dot will NOT be a problem when the statement is run:

```
DROP TABLE GRAEME . BASE_TABLE ;
DROP TABLE GRAEME . CITY_TABLE ;
DROP TABLE GRAEME . PROD_TABLE ;
DROP TABLE GRAEME . SALE_TABLE ;
DROP TABLE GRAEME . WHSE_TABLE ;
DROP TABLE GRAEME . WP_TABLE   ;
```

- DML EXAMPLE: This query return an UPDATE TABLE statement for each row in the Product table. After each update statement there will be a COMMIT. The right most field is used to control the order of the output lines, first by Product number, then by statement line number. Sample output follows:

```
UPDATE SALE_TABLE SET ITEM_COST =      4000 --     2000.1
WHERE PROD_NUM =                       2000;--     2000.2
COMMIT; ---                               0 --     2000.3
UPDATE SALE_TABLE SET ITEM_COST =      4002 --     2001.1
WHERE PROD_NUM =                       2001;--     2001.2
COMMIT; ---                               0 --     2001.3
UPDATE SALE_TABLE SET ITEM_COST =      4004 --     2002.1
WHERE PROD_NUM =                       2002;--     2002.2
COMMIT; ---                               0 --     2002.3
UPDATE SALE_TABLE SET ITEM_COST =      4006 --     2003.1
WHERE PROD_NUM =                       2003;--     2003.2
COMMIT; ---                               0 --     2003.3
UPDATE SALE_TABLE SET ITEM_COST =      4008 --     2004.1
WHERE PROD_NUM =                       2004;--     2004.2
COMMIT; ---                               0 --     2004.3
...
```

Using SQL to generate DDL / DML / SQL

DDL

```
SELECT  'DROP TABLE ', CREATOR, '.', NAME, ';'
FROM    SYSIBM.SYSTABLES
WHERE   CREATOR = USER
  AND   TYPE    = 'T';
```

DML

```
SELECT 'UPDATE SALE_TABLE SET ITEM_COST= ', PROD_NUM * 2,' '
      ,'-- ' , PROD_NUM + 0.1
FROM   PROD_TABLE
WHERE  PROD_NUM < 2010
UNION ALL
SELECT 'WHERE PROD_NUM = ',                  PROD_NUM,    ';'
      ,'-- ' , PROD_NUM + 0.2
FROM   PROD_TABLE
WHERE  PROD_NUM < 2010
UNION ALL
SELECT 'COMMIT; ---',                        0,           ' '
      ,'-- ' , PROD_NUM + 0.3
FROM   PROD_TABLE
WHERE  PROD_NUM < 2010
ORDER BY 4,5;
```

Exercise 3

DDL

See page 122 for a description of the tables used.

Q1

Select a list of individual Product numbers from the SALE table where all the following is true:

- The number of rows per PROD_NUM is more than 8.
- The number of distinct Customer numbers per PROD_NUM is > 3.
- Either the AVG(NUM_SOLD) is > 40 OR the MAX(CUST_NUM) is < 9999 (per PROD_NUM).
- The MIN(SALE_TIME) is less than 11pm (per PROD_NUM).
- The AVG(NUM_SOLD) is less than 47 (per PROD_NUM).

Order the output in descending PROD_NUM order.

Q2

Select all columns from the PROD table where the Product number is less than 2100 and EITHER of the following two conditions are true:

- The fourth character of the Product type field is a '0'.
- There is at least one MATCHING row in the SALE table with a NOT NULL Customer number.

Q3

Select the following fields:

- The Product Type from the PROD table.
- The City Region from the CITY table.
- All the fields in the WHSE table.

To get the desired data join the CITY, WHSE, WP and PROD tables by common key(s) and fetch all rows where the following are true:

- The City DATA field equals "A".
- The Warehouse NAME is "0093", or the Warehouse SIZE is less than 22.
- The Warehouse/Product QTY is less than 2222.
- The Product TYPE is "J"
- There exists at least one row in the SALE table with matching Product number for a not null Customer that happened after the JULIAN date: 1992246.

Refer to pages 122 and 123 for a list of the keys in each of the tables that have to be joined. See page 92 for Julian date usage information.

Hint: Do the four table join first, leave the subquery to last.

Q4

Select all columns from the SALE table where the following is true:

- The Product number is less than 2020.
- The Customer number is the MAX, or the MIN, for the current Product number (see page **Error! Bookmark not defined.**).

Order the output by Product number and Sale time.

Q5

Select all rows from the CITY table, getting the following fields:

- The City number.
- The City region.
- The thrid to last (from the right) character in the City region (see page 89 for help).

Exercise 3

Q6

Select the following columns from the SALE table:

- The Product number in Decimal (8,3) format.
- The Sale Date in USA format.
- The minute when the sale occurred in Decimal (2,0) format.
- The first four bytes of the Padding field.
- The Number sold times 16.
- The Customer number (assume -1 when NULL)

Only fetch rows where the following is true:

- The Product number is less than 2010.
- The Product number is less than the year of the sale (plus 100 years).
- The Sale hour is greater than the Sale minute.
- The Customer number is greater than 55 (assume it equals the number sold when NULL).
- The query is run sometime before the year 1997.
- The user who ran the query does NOT have a SQLID of FRED.

Q7

Which of the following two SQL statements gives the most accurate result?

```
SELECT AVG(WHSE_SIZE)
FROM   WHSE_TABLE;

SELECT AVG(WHSE_SIZE * 1.0)
FROM   WHSE_TABLE;
```

Q8

Select all the rows in the Warehouse table which have the second to highest city number.

Answers

Data: See page 137.
SQL: See page 142.

Notes

Table of Contents

NOTES

- INTRODUCTION: This section contains additional information which is not covered in the course but which may be useful for reference.

Table of Contents

Sample Database	122
Sample DDL/Inserts - Base Table	124
Sample DDL/Inserts - City Table	125
Sample DDL/Inserts - Warehouse Table	126
Sample DDL/Inserts - Product Table	127
Sample DDL/Inserts - Warehouse/Product Table	128
Sample DDL/Inserts - Sales Table	129
Sample Data - City & Warehouse Tables	130
Sample Data - Product Table	131
Sample Data - Warehouse/Product Table	132
Sample Data - Sale Table	133
Answers - Data, Exercise 1	135
Answers - Data, Exercise 2	136
Answers - Data, Exercise 3	137
Answers - SQL, Exercise 1	138
Answers - SQL, Exercise 2	140
Answers - SQL, Exercise 3	142

Sample Database

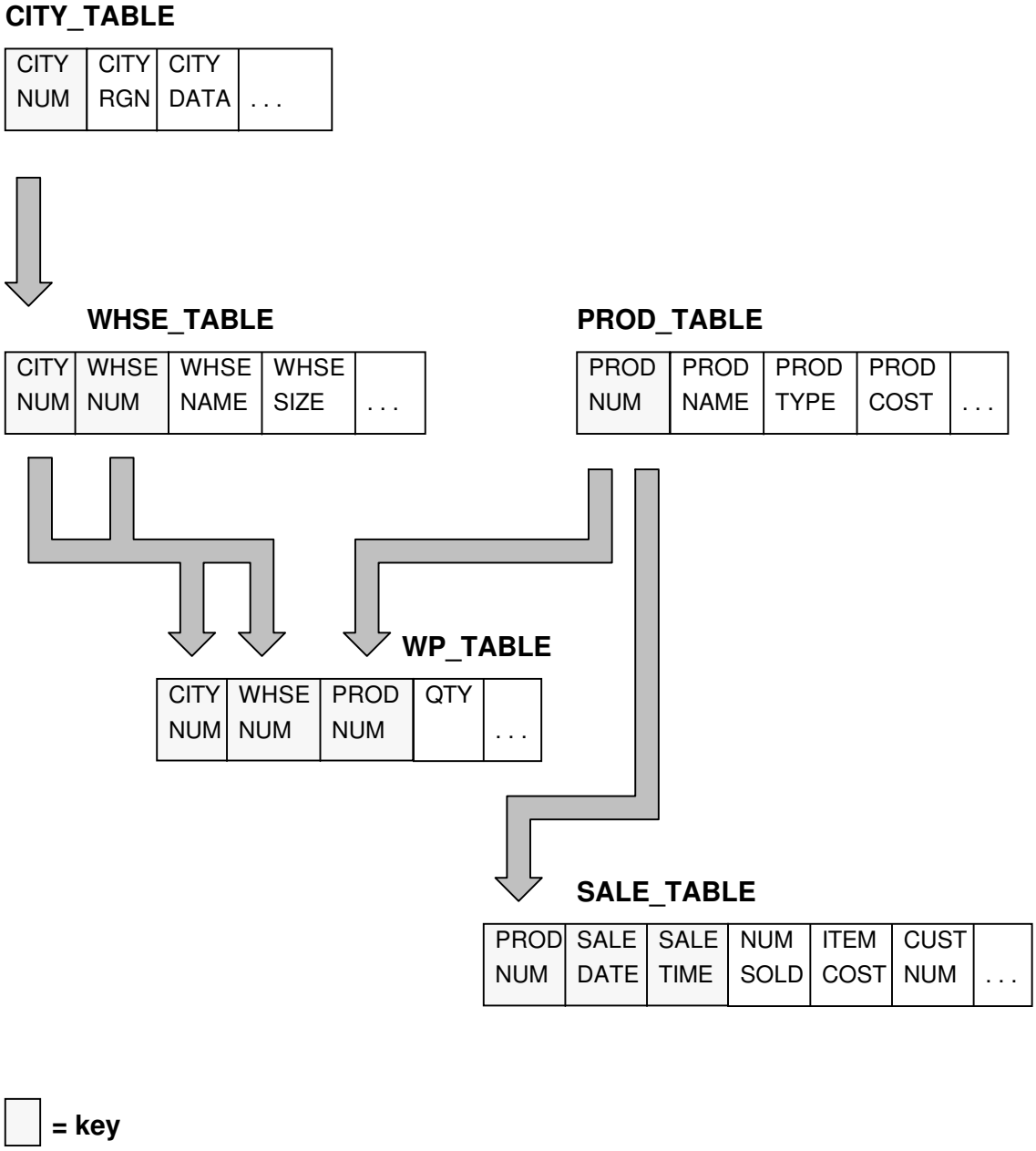
COLUMNS

CITY_TABLE			PROD_TABLE		
CITY_NUM	INTEGER	NOT NULL	PROD_NUM	INTEGER	NOT NULL
CITY_RGN	CHAR(04)	NOT NULL	PROD_NAME	CHAR(04)	NOT NULL
CITY_DATA	CHAR(04)	NOT NULL	PROD_TYPE	CHAR(04)	NOT NULL
CITY_PADDING	CHAR(80)	NOT NULL	PROD_COST	INTEGER	NOT NULL
WHSE_TABLE			PROD_PADDING	CHAR(76)	NOT NULL
CITY_NUM	INTEGER	NOT NULL	**SALE_TABLE**		
WHSE_NUM	INTEGER	NOT NULL	PROD_NUM	INTEGER	NOT NULL
WHSE_NAME	CHAR(04)	NOT NULL	SALE_DATE	DATE	
WHSE_SIZE	INTEGER	NOT NULL	SALE_TIME	TIME	
WHSE_PADDING	CHAR(76)	NOT NULL	NUM_SOLD	INTEGER	
WP_TABLE			ITEM_COST	DECIMAL(7,2)	
CITY_NUM	INTEGER	NOT NULL	CUST_NUM	INTEGER	
WHSE_NUM	INTEGER	NOT NULL	SALE_PADDING	CHAR(63)	
PROD_NUM	INTEGER	NOT NULL			
QTY	INTEGER	NOT NULL			
WP_PADDING	CHAR(76)	NOT NULL			

INDEXES

TABLE-NAME	INDEX-NAME	INDEX-TYPE	FULLKEYCARD	INDEX-COLS	COLUMN CARD
CITY_TABLE	ICITYNUM	Unique	10	CITY_NUM	10
	ICITYRGN	Non-unique	2	CITY_RGN	2
WHSE_TABLE	IWHSENUM	Unique	40	CITY_NUM	10
				WHSE_NUM	4
	IWHSENME	Unique	40	WHSE_NAME	40
PROD_TABLE	IPRODNUM	Unique	300	PROD_NUM	300
	IPRODNME	Unique	300	PROD_NAME	300
	IPRODTYP	Non-unique	30	PROD_TYPE	30
WP_TABLE	IWPWHSE	Unique	12,000	CITY_NUM	10
				WHSE_NUM	4
				PROD_NUM	300
	IWPPROD	Unique	12,000	PROD_NUM	300
				CITY_NUM	10
				WHSE_NUM	4
SALE_TABLE	ISALENUM	Unique	2,400	PROD_NUM	240
				SALE_DATE	30
				SALE_TIME	80
	ICUSTNUM	Non-unique	21	CUST_NUM	21

Sample Database



Sample DDL/Inserts - Base Table

```
CREATE TABLE BASE_TABLE
  (U1     SMALLINT  NOT NULL,
   U2     CHAR(02)  NOT NULL,
   U3     CHAR(02)  NOT NULL,
   D1     SMALLINT  NOT NULL,
   D2     CHAR(01)  NOT NULL,
   DATE1  DATE      NOT NULL);
COMMIT;

INSERT INTO BASE_TABLE VALUES (0,'C0','W0',9,'J','0001-01-01');
INSERT INTO BASE_TABLE VALUES (1,'C1','W1',8,'I','0001-01-01');
INSERT INTO BASE_TABLE VALUES (2,'C2','W2',7,'H','0001-01-01');
INSERT INTO BASE_TABLE VALUES (3,'C3','W3',6,'G','0001-01-01');
INSERT INTO BASE_TABLE VALUES (4,'C4','W4',5,'F','0001-01-01');
INSERT INTO BASE_TABLE VALUES (5,'C5','W5',4,'E','0001-01-01');
INSERT INTO BASE_TABLE VALUES (6,'C6','W6',3,'D','0001-01-01');
INSERT INTO BASE_TABLE VALUES (7,'C7','W7',2,'C','0001-01-01');
INSERT INTO BASE_TABLE VALUES (8,'C8','W8',1,'B','0001-01-01');
INSERT INTO BASE_TABLE VALUES (9,'C9','W9',0,'A','0001-01-01');
COMMIT;
```

Sample DDL/Inserts - City Table

```
CREATE TABLE CITY_TABLE
  (CITY_NUM            INTEGER   NOT NULL,
   CITY_RGN            CHAR(04)  NOT NULL,
   CITY_DATA           CHAR(04)  NOT NULL,
   CITY_PADDING        CHAR(80)  NOT NULL);
COMMIT;

CREATE UNIQUE INDEX ICITYNUM  ON CITY_TABLE
   (CITY_NUM);
COMMIT;

CREATE INDEX ICITYNME  ON CITY_TABLE
   (CITY_RGN);
COMMIT;

INSERT INTO CITY_TABLE
SELECT U1
      ,' '
      ,D2
      ,'CITY'
FROM BASE_TABLE;
COMMIT;

UPDATE CITY_TABLE
SET    CITY_RGN = 'C0'
WHERE  CITY_NUM = CITY_NUM/2 * 2;

UPDATE CITY_TABLE
SET    CITY_RGN = 'C1'
WHERE  CITY_NUM <> CITY_NUM/2 * 2;
COMMIT;
```

Sample DDL/Inserts - Warehouse Table

```
CREATE TABLE WHSE_TABLE
  (CITY_NUM            INTEGER   NOT NULL,
   WHSE_NUM            INTEGER   NOT NULL,
   WHSE_NAME           CHAR(04)  NOT NULL,
   WHSE_SIZE           INTEGER   NOT NULL,
   WHSE_PADDING        CHAR(76)  NOT NULL);
COMMIT;

CREATE UNIQUE INDEX IWHSENUM  ON WHSE_TABLE
   (CITY_NUM, WHSE_NUM);
COMMIT;

CREATE UNIQUE INDEX IWHSENME  ON WHSE_TABLE
   (WHSE_NAME);
COMMIT;

INSERT INTO WHSE_TABLE
SELECT A.U1
      ,100 + B.U1
      ,SUBSTR(CHAR(A.DATE1 + (A.U1*10 + B.U1) YEARS,ISO),1,4)
      ,100 + A.U1
      ,'WHSE'
FROM  BASE_TABLE A,
      BASE_TABLE B
WHERE B.U1 < 4;
COMMIT;
```

Sample DDL/Inserts - Product Table

```
CREATE TABLE PROD_TABLE
  (PROD_NUM            INTEGER   NOT NULL,
   PROD_NAME           CHAR(04)  NOT NULL,
   PROD_TYPE           CHAR(04)  NOT NULL,
   PROD_COST           INTEGER   NOT NULL,
   PROD_PADDING        CHAR(76)  NOT NULL);
COMMIT;

CREATE UNIQUE INDEX IPRODNUM  ON PROD_TABLE
   (PROD_NUM);
COMMIT;

CREATE UNIQUE INDEX IPRODNME  ON PROD_TABLE
   (PROD_NAME);
COMMIT;

CREATE INDEX IPRODTYP  ON PROD_TABLE
   (PROD_TYPE);
COMMIT;

INSERT INTO PROD_TABLE
SELECT  2000 + A.U1*100 + B.U1*10 + C.U1
       ,SUBSTR(CHAR(A.DATE1 + (A.U1*100 + B.U1*10 + C.U1) YEARS,ISO),1,4)
       ,A.D2
       ,2000 + A.U1*10+B.U1
       ,'PROD'
FROM  BASE_TABLE A,
      BASE_TABLE B,
      BASE_TABLE C
WHERE B.U1 < 3;
COMMIT;

UPDATE PROD_TABLE
SET    PROD_PADDING = 'ABC%DEF'
WHERE  PROD_NUM     =  2000;

CREATE VIEW PROD_VIEW  (PROD_COST)
    AS   SELECT  AVG(PROD_COST)
    FROM PROD_TABLE;
```

Sample DDL/Inserts - Warehouse/Product Table

```
CREATE TABLE WP_TABLE
  (CITY_NUM            INTEGER   NOT NULL,
   WHSE_NUM            INTEGER   NOT NULL,
   PROD_NUM            INTEGER   NOT NULL,
   QTY                 INTEGER   NOT NULL,
   WP_PADDING          CHAR(76)  NOT NULL);
COMMIT;

CREATE UNIQUE INDEX IWPWHSE ON WP_TABLE
   (CITY_NUM, WHSE_NUM, PROD_NUM);
COMMIT;

CREATE UNIQUE INDEX IWPPROD ON WP_TABLE
   (PROD_NUM, CITY_NUM, WHSE_NUM);
COMMIT;

CREATE VIEW WP_VIEW
 (CITY_NUM
 ,NMBR_PROD
 ,SUM_QTY
 ,NMBR_ROWS)
AS
SELECT   CITY_NUM
        ,COUNT(DISTINCT PROD_NUM)
        ,SUM(QTY)
        ,COUNT(*)
FROM     WP_TABLE
GROUP BY CITY_NUM;
COMMIT;

INSERT INTO WP_TABLE
SELECT CITY_NUM
      ,WHSE_NUM
      ,PROD_NUM
      ,WHSE_SIZE + PROD_COST
      ,'WP'
FROM   WHSE_TABLE,
       PROD_TABLE;
COMMIT;
```

Sample DDL/Inserts - Sales Table

```
CREATE TABLE SALE_TABLE
  (PROD_NUM        INTEGER       NOT NULL,
   SALE_DATE       DATE                  ,
   SALE_TIME       TIME                  ,
   NUM_SOLD        INTEGER               ,
   ITEM_COST       DECIMAL(7,2)          ,
   CUST_NUM        INTEGER               ,
   SALE_PADDING    CHAR(63)              );
COMMIT;

CREATE UNIQUE INDEX ISALENUM  ON SALE_TABLE
   (PROD_NUM
   ,SALE_DATE
   ,SALE_TIME);
COMMIT;

CREATE INDEX ISALCUST  ON SALE_TABLE
   (CUST_NUM);
COMMIT;

INSERT INTO SALE_TABLE
SELECT 2000 + A.U1*100 + B.U1*10 + C.U1
      ,DATE('1994-01-01') + B.U1 MONTHS - A.U1 DAYS
      ,TIME('12.00.00')   - D.U1 HOURS  + C.U1 MINUTES
      ,D.U1*10 + A.U1
      ,A.U1*100 + B.U1*10 + D.U1/100
      ,D.U1*1000 + C.U1*100
      ,'SALE'
FROM  BASE_TABLE A
     ,BASE_TABLE B
     ,BASE_TABLE C
     ,BASE_TABLE D
WHERE B.U1 IN (0,1,2)
  AND C.U1 BETWEEN 1 AND 8;
COMMIT;

UPDATE SALE_TABLE
SET    CUST_NUM = NULL
WHERE  PROD_NUM - (PROD_NUM/10*10) BETWEEN 3 AND 4;

UPDATE SALE_TABLE
SET    CUST_NUM  = NULL
      ,NUM_SOLD  = NULL
WHERE  PROD_NUM - (PROD_NUM/10*10) BETWEEN 5 AND 6;

UPDATE SALE_TABLE
SET    CUST_NUM  = NULL
      ,NUM_SOLD  = NULL
      ,ITEM_COST = NULL
WHERE  PROD_NUM - (PROD_NUM/10*10) BETWEEN 7 AND 8;
COMMIT;
```

Sample Data - City & Warehouse Tables

CITY_NUM	CITY_RGN	CITY_DATA	CITY_PADDING
0	C0	J	CITY
1	C1	I	CITY
2	C0	H	CITY
3	C1	G	CITY
4	C0	F	CITY
5	C1	E	CITY
6	C0	D	CITY
7	C1	C	CITY
8	C0	B	CITY
9	C1	A	CITY

CITY_NUM	WHSE_NUM	WHSE_NAME	WHSE_SIZE	WHSE_PADDING
0	100	0001	100	WHSE
0	101	0002	100	WHSE
0	102	0003	100	WHSE
0	103	0004	100	WHSE
1	100	0011	101	WHSE
1	101	0012	101	WHSE
1	102	0013	101	WHSE
1	103	0014	101	WHSE
2	100	0021	102	WHSE
2	101	0022	102	WHSE
2	102	0023	102	WHSE
2	103	0024	102	WHSE
3	100	0031	103	WHSE
3	101	0032	103	WHSE
3	102	0033	103	WHSE
3	103	0034	103	WHSE
4	100	0041	104	WHSE
4	101	0042	104	WHSE
4	102	0043	104	WHSE
4	103	0044	104	WHSE
5	100	0051	105	WHSE
5	101	0052	105	WHSE
5	102	0053	105	WHSE
5	103	0054	105	WHSE
6	100	0061	106	WHSE
6	101	0062	106	WHSE
6	102	0063	106	WHSE
6	103	0064	106	WHSE
7	100	0071	107	WHSE
7	101	0072	107	WHSE
7	102	0073	107	WHSE
7	103	0074	107	WHSE
8	100	0081	108	WHSE
8	101	0082	108	WHSE
8	102	0083	108	WHSE
8	103	0084	108	WHSE
9	100	0091	109	WHSE
9	101	0092	109	WHSE
9	102	0093	109	WHSE
9	103	0094	109	WHSE

Sample Data - Product Table

PROD_NUM	PROD_NAME	PROD_TYPE	PROD_COST	PROD_PADDING
2000	0001	J	2000	ABC%DEF
2001	0002	J	2000	PROD
2002	0003	J	2000	PROD
2003	0004	J	2000	PROD
2004	0005	J	2000	PROD
2005	0006	J	2000	PROD
2006	0007	J	2000	PROD
2007	0008	J	2000	PROD
2008	0009	J	2000	PROD
2009	0010	J	2000	PROD
2010	0011	J	2001	PROD
2011	0012	J	2001	PROD
2012	0013	J	2001	PROD
2013	0014	J	2001	PROD
2014	0015	J	2001	PROD
2015	0016	J	2001	PROD
2016	0017	J	2001	PROD
2017	0018	J	2001	PROD
2018	0019	J	2001	PROD
2019	0020	J	2001	PROD
2020	0021	J	2002	PROD
2021	0022	J	2002	PROD
2022	0023	J	2002	PROD
2023	0024	J	2002	PROD
2024	0025	J	2002	PROD
2025	0026	J	2002	PROD
2026	0027	J	2002	PROD
2027	0028	J	2002	PROD
2028	0029	J	2002	PROD
2029	0030	J	2002	PROD
2100	0101	I	2010	PROD
2101	0102	I	2010	PROD
2102	0103	I	2010	PROD
2103	0104	I	2010	PROD
2104	0105	I	2010	PROD
2105	0106	I	2010	PROD
2106	0107	I	2010	PROD
2107	0108	I	2010	PROD
2108	0109	I	2010	PROD
2109	0110	I	2010	PROD
2110	0111	I	2011	PROD
2111	0112	I	2011	PROD
.				
.				
.				
2920	0921	A	2092	PROD
2921	0922	A	2092	PROD
2922	0923	A	2092	PROD
2923	0924	A	2092	PROD
2924	0925	A	2092	PROD
2925	0926	A	2092	PROD
2926	0927	A	2092	PROD
2927	0928	A	2092	PROD
2928	0929	A	2092	PROD
2929	0930	A	2092	PROD

Sample Data - Warehouse/Product Table

CITY_NUM	WHSE_NUM	PROD_NUM	QTY	WP_PADDING
0	100	2000	2100	WP
0	100	2001	2100	WP
0	100	2002	2100	WP
0	100	2003	2100	WP
0	100	2004	2100	WP
0	100	2005	2100	WP
0	100	2006	2100	WP
0	100	2007	2100	WP
0	100	2008	2100	WP
0	100	2009	2100	WP
0	100	2010	2101	WP
0	100	2011	2101	WP
0	100	2012	2101	WP
0	100	2013	2101	WP
0	100	2014	2101	WP
0	100	2015	2101	WP
0	100	2016	2101	WP
0	100	2017	2101	WP
0	100	2018	2101	WP
0	100	2019	2101	WP
0	100	2020	2102	WP
0	100	2021	2102	WP
0	100	2022	2102	WP
0	100	2023	2102	WP
0	100	2024	2102	WP
0	100	2025	2102	WP
0	100	2026	2102	WP
0	100	2027	2102	WP
.				
.				
0	101	2000	2100	WP
0	101	2001	2100	WP
0	101	2002	2100	WP
0	101	2003	2100	WP
.				
.				
0	102	2000	2100	WP
0	102	2001	2100	WP
0	102	2002	2100	WP
0	102	2003	2100	WP
.				
.				
0	103	2000	2100	WP
0	103	2001	2100	WP
0	103	2002	2100	WP
0	103	2003	2100	WP
.				
.				
.				
9	103	2924	2201	WP
9	103	2925	2201	WP
9	103	2926	2201	WP
9	103	2927	2201	WP
9	103	2928	2201	WP
9	103	2929	2201	WP

Sample Data - Sale Table

PROD_NUM	SALE_DATE	SALE_TIME	NUM_SOLD	ITEM_COST	CUST_NUM	SALE_PADDING
2001	01/01/1994	03:01:00	90	.00	9100	SALE
2001	01/01/1994	04:01:00	80	.00	8100	SALE
2001	01/01/1994	05:01:00	70	.00	7100	SALE
2001	01/01/1994	06:01:00	60	.00	6100	SALE
2001	01/01/1994	07:01:00	50	.00	5100	SALE
2001	01/01/1994	08:01:00	40	.00	4100	SALE
2001	01/01/1994	09:01:00	30	.00	3100	SALE
2001	01/01/1994	10:01:00	20	.00	2100	SALE
2001	01/01/1994	11:01:00	10	.00	1100	SALE
2001	01/01/1994	12:01:00	0	.00	100	SALE
.						
.						
2003	01/01/1994	03:03:00	90	.00	?	SALE
2003	01/01/1994	04:03:00	80	.00	?	SALE
2003	01/01/1994	05:03:00	70	.00	?	SALE
2003	01/01/1994	06:03:00	60	.00	?	SALE
2003	01/01/1994	07:03:00	50	.00	?	SALE
2003	01/01/1994	08:03:00	40	.00	?	SALE
2003	01/01/1994	09:03:00	30	.00	?	SALE
2003	01/01/1994	10:03:00	20	.00	?	SALE
2003	01/01/1994	11:03:00	10	.00	?	SALE
.						
.						
2005	01/01/1994	03:05:00	?	.00	?	SALE
2005	01/01/1994	04:05:00	?	.00	?	SALE
2005	01/01/1994	05:05:00	?	.00	?	SALE
2005	01/01/1994	06:05:00	?	.00	?	SALE
2005	01/01/1994	07:05:00	?	.00	?	SALE
2005	01/01/1994	08:05:00	?	.00	?	SALE
2005	01/01/1994	09:05:00	?	.00	?	SALE
2005	01/01/1994	10:05:00	?	.00	?	SALE
2005	01/01/1994	11:05:00	?	.00	?	SALE
.						
.						
2007	01/01/1994	03:07:00	?	?	?	SALE
2007	01/01/1994	04:07:00	?	?	?	SALE
2007	01/01/1994	05:07:00	?	?	?	SALE
2007	01/01/1994	06:07:00	?	?	?	SALE
2007	01/01/1994	07:07:00	?	?	?	SALE
2007	01/01/1994	08:07:00	?	?	?	SALE
2007	01/01/1994	09:07:00	?	?	?	SALE
2007	01/01/1994	10:07:00	?	?	?	SALE
2007	01/01/1994	11:07:00	?	?	?	SALE
.						
.						
.						
2928	02/20/1994	03:08:00	?	?	?	SALE
2928	02/20/1994	04:08:00	?	?	?	SALE
2928	02/20/1994	05:08:00	?	?	?	SALE
2928	02/20/1994	06:08:00	?	?	?	SALE
2928	02/20/1994	07:08:00	?	?	?	SALE
2928	02/20/1994	08:08:00	?	?	?	SALE
2928	02/20/1994	09:08:00	?	?	?	SALE
2928	02/20/1994	10:08:00	?	?	?	SALE
2928	02/20/1994	11:08:00	?	?	?	SALE
2928	02/20/1994	12:08:00	?	?	?	SALE

Blank Page

Answers - Data, Exercise 1

Q1

```
1800
```

Q2

```
480
```

Q3

```
4
```

Q4

```
2464   2464
```

Q5

```
59400   58400   59400.
```

Q6

```
36
```

Q7

```
29
```

Q8

```
4   4.500
```

Q9

```
01/01/1994   01.01.1994   1   1.   1994-01-01-03.01.00.000000
```

Q10

```
1994-01-01   2004-03-06   2004-03-06   729   3
```

Q11

```
1994-01-01   1994-01-01   1994-01   9100.
```

Q12

```
720
```

Q13

```
PROD_NUM    #1     #2     #3     #4     #5
---------  ----   ----   ----   ----   ----
     2001   PROD   DROD   DQOD   DROP   DROD
     2002   PROD   DROD   DQOD   DROP   DROD
     2003   PROD   DROD   DQOD   DROP   DROD
     2004   PROD   DROD   DQOD   DROP   DROD
     2005   PROD   DROD   DQOD   DROP   DROD
     2006   PROD   DROD   DQOD   DROP   DROD
     2007   PROD   DROD   DQOD   DROP   DROD
     2008   PROD   DROD   DQOD   DROP   DROD
```

Answers - Data, Exercise 2

Q1

```
PROD_NUM  COUNT
--------  -----
    2001     10
    2002     10
```

Q2

```
PROD_NUM  COUNT
--------  -----
    2003     10
    2004     10
    2005     10
    2006     10
    2007     10
    2008     10
```

Q3

```
PROD_NUM  SALE_DATE   SALE_TIME  NUM_SOLD  ITEM_COST  CUST_NUM  SALE_PAD
--------  ----------  ---------  --------  ---------  --------  --------
    2001  1994-01-01  03.01.00         90        .00      9100  SALE
    2001  1994-01-01  04.01.00         80        .00      8100  SALE
    2002  1994-01-01  03.02.00         90        .00      9200  SALE
    2002  1994-01-01  04.02.00         80        .00      8200  SALE
```

Q4

```
192  1
```

Q5

```
24
```

Q6

```
60
```

Q7

```
60
```

Q8

```
NOT NULL:  600
NULL:     1800
```

Q9

```
2000  2009 2010  2019  2020  2029
```

Q10

```
9  9  9  9  2  1  0
```

Answers - Data, Exercise 3

Q1

12 rows

Q2

6 rows, PROD_NUM: 2021 2011 2001 2002 2012 2022

Q3

PROD_TYPE	CITY_RGN	CITY_NUM	WHSE_NUM	WHSE_NAME	WHSE_SIZE	WHSE_PAD
J	C1	9	102	0093	109	WHSE
J	C1	9	102	0093	109	WHSE
J	C1	9	102	0093	109	WHSE
J	C1	9	102	0093	109	WHSE
J	C1	9	102	0093	109	WHSE
J	C1	9	102	0093	109	WHSE

Q4

PROD_NUM	SALE_DATE	SALE_TIME	NUM_SOLD	ITEM_COST	CUST_NUM	SALE_PAD
2001	1994-01-01	03.01.00	90	.00	9100	SALE
2001	1994-01-01	12.01.00	0	.00	100	SALE
2002	1994-01-01	03.02.00	90	.00	9200	SALE
2002	1994-01-01	12.02.00	0	.00	200	SALE
2011	1994-02-01	03.01.00	90	10.00	9100	SALE
2011	1994-02-01	12.01.00	0	10.00	100	SALE
2012	1994-02-01	03.02.00	90	10.00	9200	SALE
2012	1994-02-01	12.02.00	0	10.00	200	SALE

Q5

CITY_NUM	CITY_RGN	_
0	C0	0
1	C1	1
2	C0	0
3	C1	1
4	C0	0
5	C1	1
6	C0	0
7	C1	1
8	C0	0
9	C1	1

Q6

59 rows

Q7

The second statement is more accurate.

Q8

CITY_NUM	WHSE_NUM	WHSE_NAME	WHSE_SIZE	WHSE_PADDING
8	100	0081	108	WHSE
8	101	0082	108	WHSE
8	102	0083	108	WHSE
8	103	0084	108	WHSE

Answers - SQL, Exercise 1

Q1

```
SELECT COUNT(*)
FROM   SALE_TABLE
WHERE  CUST_NUM IS NULL;
```

Q2

```
SELECT COUNT(*)
FROM   SALE_TABLE
WHERE  DAY(SALE_DATE) IN (1,11,21,31);
```

Q3

```
SELECT COUNT(DISTINCT WHSE_NUM)
FROM   WP_TABLE
WHERE  CITY_NUM < 2;
```

Q4

```
SELECT AVG(PROD_NUM)
      ,AVG(DISTINCT PROD_NUM)
FROM   SALE_TABLE;
```

Q5

```
SELECT INTEGER(SUM(NUM_SOLD))
      ,DECIMAL(SUM(NUM_SOLD))
      ,FLOAT(SUM(NUM_SOLD))
FROM   SALE_TABLE;
```

Q6

```
SELECT COUNT(*)
FROM   SALE_TABLE
WHERE  MONTH(SALE_DATE) = 12
  AND  HOUR(SALE_TIME)  =  9
  AND  NUM_SOLD IS NOT NULL;
```

Q7

```
SELECT COUNT(*)
FROM   PROD_TABLE
WHERE  PROD_TYPE LIKE '%J%'
  AND  PROD_PADDING LIKE 'P%'
  AND  LENGTH(PROD_PADDING)  >  3
  AND  SUBSTR(PROD_NAME,2,1) < 'A';
```

Q8

```
SELECT AVG(CITY_NUM)
      ,AVG(CITY_NUM * 1.0)
FROM   CITY_TABLE;
```

Q9

```
SELECT CHAR(SALE_DATE,USA)
      ,CHAR(SALE_DATE,EUR)
      ,DAY(SALE_DATE)
      ,DECIMAL(DAY(SALE_DATE),2,0)
      ,TIMESTAMP(SALE_DATE,SALE_TIME)
FROM   SALE_TABLE
WHERE  PROD_NUM = 2001
  AND  HOUR(SALE_TIME) = 3;
```

Answers - SQL, Exercise 1

Q10

```
SELECT SALE_DATE
      ,SALE_DATE + 10 YEARS + 2 MONTHS + 5 DAYS
      ,SALE_DATE + DECIMAL(00100205,8,0)
      ,DAYS('1995-12-31') - DAYS(SALE_DATE)
      ,DAY(SALE_DATE) * HOUR(SALE_TIME)
FROM   SALE_TABLE
WHERE  PROD_NUM = 2001
  AND  HOUR(SALE_TIME) = 3;
```

Q11

```
SELECT SALE_DATE
      ,CHAR(SALE_DATE,ISO)
      ,SUBSTR(CHAR(SALE_DATE,ISO),1,7)
      ,FLOAT(CUST_NUM)
FROM   SALE_TABLE
WHERE  PROD_NUM = 2001
  AND  HOUR(SALE_TIME) = 3;
```

Q12

```
SELECT COUNT(*)
FROM   SALE_TABLE
WHERE  PROD_NUM  BETWEEN 2000 AND 3000
  AND  SALE_DATE > '1991-12-31'
  AND  SALE_TIME BETWEEN '03.00.00' AND '06.00.00'
  AND  SALE_PADDING LIKE '_A%';
```

Q13

```
SELECT PROD_NUM
      ,SUBSTR(PROD_PADDING,1,4)
      ,TRANSLATE(SUBSTR(PROD_PADDING,1,4),'D','P')
      ,TRANSLATE(SUBSTR(PROD_PADDING,1,4),'DQ','PR')
      ,TRANSLATE(SUBSTR(PROD_PADDING,1,4),'DP','PD')
      ,TRANSLATE(SUBSTR(PROD_PADDING,1,4),'D','P','Z')
FROM   PROD_TABLE
WHERE  PROD_NUM  < 2009
  AND  PROD_TYPE IN ('J','K','L','M')
  AND  PROD_NAME <> ''''
  AND  PROD_PADDING LIKE '% %'
  AND  TRANSLATE(PROD_PADDING,'$@','%$') NOT LIKE '%$%';
```

Answers - SQL, Exercise 2

Q1

```
SELECT   PROD_NUM
        ,COUNT(*)
FROM     SALE_TABLE
WHERE    PROD_NUM < 2010
  AND    CUST_NUM IS NOT NULL
GROUP BY PROD_NUM
HAVING   COUNT(DISTINCT SALE_DATE) = 1;
```

Q2

```
SELECT   PROD_NUM
        ,COUNT(*)
FROM     SALE_TABLE
WHERE    PROD_NUM < 2010
  AND    CUST_NUM IS NULL
GROUP BY PROD_NUM
HAVING   COUNT(*) > 5
   AND   COUNT(DISTINCT SALE_DATE) = 1
   AND   AVG(HOUR(SALE_TIME)) > 4;
```

Q3

```
SELECT   *
FROM     SALE_TABLE A
WHERE    PROD_NUM  < 2005
  AND    ITEM_COST < 10
  AND    SALE_TIME < '05.00.00'
  AND    1 <
        (SELECT COUNT(DISTINCT CUST_NUM)
         FROM   SALE_TABLE B
         WHERE  A.PROD_NUM = B.PROD_NUM
           AND  B.CUST_NUM IS NOT NULL);
```

Q4

```
SELECT   COUNT(*)
        ,COUNT(DISTINCT A.PROD_TYPE)
FROM     PROD_TABLE A
        ,SALE_TABLE B
WHERE    A.PROD_NUM = B.PROD_NUM
  AND    A.PROD_TYPE LIKE 'J%'
  AND    B.SALE_TIME < '11.00.00';
```

Q5

```
SELECT   COUNT(*)
FROM     PROD_TABLE A
WHERE    PROD_NUM BETWEEN 2000 AND 2100
  AND    5 <
        (SELECT COUNT(*)
         FROM   SALE_TABLE B
         WHERE  A.PROD_NUM = B.PROD_NUM)
  AND    10 <
        (SELECT COUNT(*)
         FROM   WP_TABLE C
         WHERE  A.PROD_NUM = C.PROD_NUM);
```

Answers - SQL, Exercise 2

Q6

```
SELECT   COUNT(*)
FROM     PROD_TABLE
WHERE    PROD_NUM NOT IN
        (SELECT PROD_NUM
         FROM   SALE_TABLE);
```

Q7

```
SELECT   COUNT(*)
FROM     PROD_TABLE A
WHERE    PROD_NUM NOT IN
        (SELECT B.PROD_NUM
         FROM   SALE_TABLE B
         WHERE  A.PROD_NUM = B.PROD_NUM);
```

Q8

```
SELECT  'NULL'
        ,COUNT(*)
FROM     SALE_TABLE A
WHERE    CUST_NUM IS NULL
UNION
SELECT  'NOT NULL'
        ,COUNT(*)
FROM     SALE_TABLE A
WHERE    CUST_NUM IS NOT NULL;
```

Q9

```
SELECT   PROD_NUM
FROM     PROD_TABLE
WHERE    PROD_TYPE = 'J'
EXCEPT
SELECT   PROD_NUM
FROM     SALE_TABLE;
```

Q10

```
SELECT CITY_NUM
FROM   CITY_TABLE
INTERSECT
SELECT CITY_NUM
FROM   WHSE_TABLE
WHERE  WHSE_SIZE < 104
EXCEPT
SELECT CITY_NUM
FROM   WHSE_TABLE
WHERE  WHSE_NAME > '0033'
UNION ALL
SELECT CITY_NUM
FROM   WHSE_TABLE
WHERE  WHSE_SIZE > 108
ORDER BY 1 DESC;
```

Answers - SQL, Exercise 3

Q1

```
SELECT   PROD_NUM
FROM     SALE_TABLE
GROUP BY PROD_NUM
HAVING   COUNT(*) > 8
   AND   COUNT(DISTINCT CUST_NUM) > 3
   AND  (AVG(NUM_SOLD)  > 40
    OR   MAX(CUST_NUM)  < 9999)
   AND   MIN(SALE_TIME) < '23.00.00'
   AND   AVG(NUM_SOLD)  < 47
ORDER BY PROD_NUM DESC;
```

Q2

```
SELECT  *
FROM    PROD_TABLE A
WHERE   PROD_NUM < 2100
  AND ( SUBSTR(PROD_TYPE,4,1) = '0'
   OR   EXISTS
       (SELECT *
        FROM   SALE_TABLE B
        WHERE  A.PROD_NUM = B.PROD_NUM
          AND  B.CUST_NUM IS NOT NULL) );
```

Q3

```
SELECT   D.PROD_TYPE
        ,A.CITY_RGN
        ,B.*
FROM     CITY_TABLE A
        ,WHSE_TABLE B
        ,WP_TABLE   C
        ,PROD_TABLE D
WHERE    A.CITY_DATA = 'A'
  AND    A.CITY_NUM  = B.CITY_NUM
  AND  ( B.WHSE_NAME = '0093'
   OR    B.WHSE_SIZE < 22 )
  AND    B.CITY_NUM  = C.CITY_NUM
  AND    B.WHSE_NUM  = C.WHSE_NUM
  AND    C.QTY       < 2222
  AND    C.PROD_NUM  = D.PROD_NUM
  AND    D.PROD_TYPE = 'J'
  AND    EXISTS
        (SELECT *
         FROM   SALE_TABLE E
         WHERE  D.PROD_NUM  = E.PROD_NUM
           AND  E.SALE_DATE > DATE('1992246')
           AND  E.CUST_NUM IS NOT NULL);
```

Answers - SQL, Exercise 3

Q4

```
SELECT   *
FROM     SALE_TABLE A
WHERE    PROD_NUM < 2020
  AND (  CUST_NUM IN
        (SELECT MAX(CUST_NUM)
         FROM   SALE_TABLE B
         WHERE  A.PROD_NUM = B.PROD_NUM)
  OR     CUST_NUM IN
        (SELECT MIN(CUST_NUM)
         FROM   SALE_TABLE B
         WHERE  A.PROD_NUM = B.PROD_NUM) )
ORDER BY PROD_NUM
        ,SALE_TIME;
```

Q5

```
SELECT   CITY_NUM
        ,CITY_RGN
        ,SUBSTR(CITY_RGN,LENGTH(CITY_RGN)-2,1)
FROM     CITY_TABLE;
```

Q6

```
SELECT   DECIMAL(PROD_NUM,8,3)
        ,CHAR(SALE_DATE,USA)
        ,DECIMAL(MINUTE(SALE_TIME),2,0)
        ,SUBSTR(SALE_PADDING,1,4)
        ,NUM_SOLD * 16
FROM     SALE_TABLE
WHERE    PROD_NUM   < 2010
  AND    PROD_NUM   < YEAR(SALE_DATE + 100 YEARS)
  AND    HOUR(SALE_TIME)           > MINUTE(SALE_TIME)
  AND    YEAR(CURRENT DATE)        < 1997
  AND    CURRENT SERVER           <> 'FRED';
```

Q7

See page 137.

Q8

```
SELECT *
FROM   WHSE_TABLE
WHERE  CITY_NUM =
      (SELECT MAX(CITY_NUM)
       FROM   WHSE_TABLE
       WHERE  CITY_NUM <
             (SELECT MAX(CITY_NUM)
              FROM   WHSE_TABLE));
```

Index

Index

NOTES

- INTRODUCTION: Use this index to locate subjects referred to in the course contents.

Index

A
ALL, subquery, 68
AND operator, 20
ANY, subquery, 68
AVG function
  Description, 34
  Truncation, 36

B
BETWEEN operator, 18
Boolean Logic, 96

C
Cartesian Product, 58
CHAR function, 28
Charts, made using SQL, 86
CITY Table
  Data Listing, 130
  DDL Listing, 125
Correlated Subquery, 72
Correlation name, 14
COUNT function
  Description, 34
  Group By, 48
  Multiple in one pass, 98
CURRENT DATE, 16

D
Date
  Addition, 90
  Arithmetic, 24
  Duration, 22
  Field type, 22
  Function, 28
  Julian data, 92
  Multiplication, 90
  Week processing, 94
DAY function, 28, 90
DAYS function, 28, 90
DDL
  Creating using SQL, 114
  Sample Tables, 124-30
DECIMAL function
  Description, 30
  Truncation, 36
DISTINCT
  Function usage, 34
  Multiple in query, 110

E
EQUAL operator, 100
Equi Join, 52-55
Except, 64-68
EXISTS, subquery, 68

F
FLOAT function, 30
From Statement, 14
Functions, 28-40

G
Gregorian Date, 92
Group By
  Day of week, 94
  Description, 48
  In subquery, 108
  View usage, 94

H
HAVING, 34, 48
HOUR function, 28

I
IN operator
  Predicate, 18
  Subquery, 68
INTEGER function
  Description, 30
  Multiple Counts, 98
  Truncation, 36
Intersect, 64-68

J
Join
  Boolean Logic, 96
  Cartesian Product, 58
  Equi Join, 52
  In subquery, 108
  Outer Join, 56

Index

Types, 50
Julian Date
Conversion, 92
Date function, 28
Historical meaning, 92
Julian Day, 92

L
Labeled Durations, 16, 22
LENGTH function
Description, 30
Making charts, 86
Null varchar field, 88
LIKE operator, 18, 32, 100
Long Varchar, 38

M
MAX
Function, 34
Row found, 102
Ten rows, see MIN
Timestamp rows, 106
MICROSECOND function, 28
MIN
Function, 34
Row found, 102
Ten rows, 104
MINUTE function, 28
MONTH
Addition, 26, 90
Function, 28, 90
Selecting last of month, 26

N
NOT
BETWEEN, 18
EXISTS, 68
LIKE, 18
NULL, 18
NULL
Length function, 88
NOT operator, 18
Number
Dissection, 88
Functions, 30

O
OR operator, 20
Order By
Description, 48
Outer Join, 56

P
Percentage Calculation, 98
Precedence Rules, 20
PROD Table
Data Listing, 131
DDL Listing, 127

Q
Quotes, 16

R
Rounding errors, 36

S
SALE Table
Data Listing, 133
DDL Listing, 129
Sample Tables
Data Listings, 130-35
Diagram, 11, 47, 85
Scalar functions, 28-34
SECOND function, 28
SOME, subquery, 68
Special Registers, 16
Subquery
Boolean Logic, 96
Correlated, 72
Flavours, 74-77
Syntax, 68
Uncorrelated, 70
SUBSTR function
Character dissection, 88
Description, 30
Long varchar fields, 38
Making charts, 86
Making fixed output, 38
SUM function, 34, 98
Syntax Diagram Conventions, 12

Index

T
Test data, creating, 112
Time
  Arithmetic, 24
  Duration, 22
  Field type, 22
  Function, 28
Timestamp
  Duration, 22
  Field type, 22
  Function, 28
  Input formats, 106
  Max rows, 106
TRANSLATE function
  Description, 30, 32
  LIKE operator, 32

U
Uncorrelated Subquery, 70
Union
  All, 60
  Boolean Logic, 96
  Description, 60
  Outer Join, 56
  Output order, 60
USER, 16

V
VARCHAR
  Index-only access, 38
  Length function, 30
  LIKE processing, 100
  Long varchar, 38
VARGRAPHIC function, 30
View
  Distinct usage, 110
  Group By usage, 94
  Week processing, 94

W
Week
  Group by day, 94
  Select numbers, 94
  Selecting days, 94
WHSE Table
  Data Listing, 130
  DDL Listing, 126
WP Table
  Data Listing, 132
  DDL Listing, 128

Y
YEAR function, 28, 90