Introduction to SQL Using *Oracle*

Barry Dwyer © 2001

Contents

1 BASIC CONCEPTS 3

1.1 SCHEMAS AND INSTANCES 3
1.2 DATABASE SUB-LANGUAGES 3
1.3 AN EXAMPLE DATABASE 3
1.4 FOREIGN KEYS 5

2 CREATING SCHEMAS 5

2.1 CREATING TABLES 5
2.2 DATA TYPES 5
2.3 ENSURING UNIQUENESS 7
2.4 CHANGING SCHEMAS 8
2.5 DISPLAYING A SCHEMA 8
2.6 ENTITY-RELATIONSHIP DIAGRAMS 8

3 POPULATING A DATABASE 9

3.1 INSERTING ROWS 9
3.2 FORMATTING COLUMNS 9

4 PROJECTION & SELECTION 10

4.1 PROJECTION 10
4.2 ALL OR DISTINCT 11
4.3 ARITHMETIC OPERATORS 11
4.4 SELECTION 12
4.4.1 Range Testing 13
4.4.2 Pattern Matching 13
4.4.3 Testing Null Values 14
4.4.4 System Variables 15
4.4.5 Set Membership 15

5 GROUPING AND ORDERING 15

5.1 ORDERING 15
5.2 AGGREGATE FUNCTIONS 16
5.3 GROUPING 16
5.4 SELECTING AGGREGATES 17

6 SET OPERATORS 18

7 JOINS 20

7.1 CARTESIAN PRODUCT 20
7.2 TYPES OF JOIN 20
7.3 OUTER JOINS 23

8 NESTED QUERIES 23

8.1 SUB-QUERIES 24
8.2 ALL AND ANY OR SOME 25
8.3 CORRELATED SUB-QUERIES 25
8.4 EXISTS 26
8.5 A LOGIC TRAP 26

9 EXECUTION OF SELECT STATEMENTS 27

10 VIEWS 28

10.1 VIEWS AND TABLES 29
10.2 GROUPED VIEWS 30
10.3 DROPPING VIEWS 30

11 UPDATING 31

11.1 INSERT 31

11.2 DELETE 31
11.3 UPDATE 31
11.4 UPDATING VIEWS 32
11.5 WITH CHECK OPTION 32

12 DATABASE INTEGRITY 33

12.1 RANGE AND ROW CONSTRAINTS 33
12.2 UNIQUE 34
12.3 REFERENTIAL INTEGRITY 34
12.4 LOOK-UP TABLES 35
12.5 TRIGGERS 36

13 TRANSACTIONS 36

13.1 COMMIT AND ROLLBACK 36
13.2 RECOVERY 37

14 DATABASE SECURITY 37

14.1 GRANT 37
14.2 REVOKE 38

1 Basic Concepts

1.1 Schemas and Instances

A **relational database** is a *set* of named **tables**. In database management systems, we use the word 'table' rather than 'file', because there is sometimes a complex relationship between tables and the operating system files. In *Oracle*, for example, all tables share a **tablespace**, consisting of one or more files.

The *structure* of the tables is called the database **schema**; the information they contain is the database **instance**. Both the instance and the schema are **persistent** — they are recorded permanently, and changes made to them outlive the session that makes them.

1.2 Database Sub-Languages

A **database management system (DBMS)** must provide at least 3 **sub-languages**:

- A **data definition language (DDL)** to define the schema.
- A **data manipulation language (DML)** to populate, update or query the instance.
- A **data control language (DCL)** to control users' rights to access the database.

Structured Query Language (**SQL**, sometimes pronounced 'sequel') is an American National Standard (ANS) language that includes DDL, DML and DCL sub-languages. SQL is supported by almost every DBMS, of which *Oracle* is an example. DBMS's differ in the range of computer systems they support, how the database is implemented, and the additional tools and features that they offer. The version of SQL supported by *Oracle* differs in minor ways from the ANSI standard.![1]

1.3 An Example Database

As an example SQL database, we use four **tables**, which are direct equivalents of the 'Suppliers', 'Customers', 'Products' and 'Updates' files in the Cobol examples.

A **table** has a *fixed* set of **columns or attributes**, and a *variable* number of **rows**, or **tuples**.!

Suppliers

Account	Name	Street	Suburb	Balance
B007	Blue Waters Nominees	1 Francis Road	Paramatta NSW 2150	$12,0459.56
N001	Netherlands Electrical Supply	124 Burbridge Road	Hilton SA 5033	$127.99
N002	Nippon Electronics Importers	19 Ashford Rd	Redfern NSW 2016	$2,790.46
P004	Pacific Auto Electronics	PO Box 407	Sydney NSW 2000	$10,326.06
S002	South Aust. Import Trading	137 Burbridge Road	Hilton SA 5033	$0.00
U003	US Audio Imports	5 Penna Ave	Clayton VIC 3109	-$45.50

No two rows of a table can have the same value of the table's **primary key** (conventionally shown underlined), which is some subset of its columns. 'Account' is the key of the 'Suppliers' table. We may therefore think of the table being organised something like a Cobol indexed file with rows as records, and 'Account' as its record key.

Non-key columns of a table ('Name', 'Street', etc.) are called **non-key attributes**.

All columns must be simple variables — no structures or arrays. In the 'Suppliers' table, there is no notion of 'Address' comprising 'Name', 'Street', and 'Suburb'.

The 'Customers' table is similar to the 'Suppliers' table, but has additional columns for 'Credit_Limit' and 'Available_Credit'.[2]

---

[1] If some aspect of SQL is supported by *Oracle* but is not standard, it will be described here as '*Oracle* SQL.' If it is standard but is not supported by *Oracle* it will be described as 'ANS SQL.' If it is both standard and supported by *Oracle*, then it will be described simply as 'SQL.'

[2] SQL names use underscores rather than hyphens.

Customers

Account	Name	Street	Suburb	Balance	Credit _Limit	Available _Credit
A001	Autobarn Elizabeth	61 Elizabeth Way	Elizabeth SA 5158	$208.50	$5000	$4,791.50
B003	BCR Mobile Installations	25 Sydney Street	Ridghaven SA 5058	$0.00	$2000	$2,000.00
B007	Blaupunkt	Cnr Centre and McNaughton Rds	Clayton VIC 3109	$10,295.00	$25,000	$12,537.25
B012	Bobs Electronic Repairs	28 Limbert Avenue	Seacombe Gardens SA 5047	-$129.50	$1,000	$1,129.50
C002	Car Audio Designs	354 North East Road	Klemzig SA 5059	$24.00	$1,000	$976.00
C005	Car Audio Services	217, Main North Road	Sefton Park, SA 5014	$823.00	$2,000	$1,177.00
C007	Cargear Pty Ltd	453 Magill Road	St Morris SA 5059	$0.00	$1,000	$865.00
C010	Cartronics	10 Goodwood Road	Wayville SA 5014	$519.00	$1,000	$481.00

and so on ...

The 'Products' table has 9 columns. Its primary key is 'Item_No':

Products

Item_No	Description	Supplier	Reorder _Level	Reorder _Qty	On_ Order	Stock	Price	Valuation
ACLOTP	Alcatel One Touch Phone	P004	10	20	0	9	$69.50	$449.55
ALTCIU	Altec Caller ID Unit	P004	5	10	0	12	$12.00	$84.33
ALTPCD	Altec Portable CD Including Headphones	P004	10	30	30	3	$37.00	$75.00
AUBCRC	Audioboss Deluxe 60W Car Radio Cassette	U003	5	10	10	1	$34.00	$25.00
AUGBOX	Audio-Gods Box Speakers	U003	5	10	0	4	$35.00	$90.00
CANCBJ	Canon Colour Bubble Jet Printer	N002	10	20	0	14	$89.00	$1,034.77
CANFBS	Canon Flatbed Scanner	N002	7	15	0	6	$100.50	$450.00
FAXROL	Fax Rolls	S002	200	1000	0	243	$0.95	$197.36
FDM6SS	Freedom 6.5" Stereo Speakers	P004	50	100	0	124	$18.50	$1,409.67

and so on ...

A **composite key** comprises more than one column. In the next example, no two rows of the 'Updates' table can have the same *combination of* the **components** 'YY_MM_DD' and 'HH_MM_SS'. Again, since columns cannot be structured, there is no way to say that together these two columns make up the 'Time_Stamp'.

Updates

YY_MM_DD	HH_MM_SS	Kind	Item_No	Account	Qty_ Delivered	Cost	Amount
01/02/06	14:55:14	D	ALTCIU	P004	10	$100.00	
01/02/06	14:56:23	$		N002			2790.46
01/02/06	14:57:19	D	ALTPCD	P004	30	$900.00	
01/02/06	14:58:44	D	PLPDCD	N001	50	$14,950.00	
01/02/06	15:00:02	D	PLPRCM	N001	50	$4,550.00	
01/02/06	15:01:07	$		N002			50.00
01/02/06	15:01:40	D	ALTCIU	N001	10	$90.00	
01/02/14	15:54:15	D	ALTCIU	P004	10	$100.00	
01/02/14	15:55:50	D	ALTCIU	P004	10	$100.00	
01/02/14	15:56:17	$		N002			$2,790.46

Although the number of columns in a table is fixed, particular **attributes** may have **null** values. The 'Updates' table contains 7 cases where there is *no* 'Amount', and 3 cases where there is *no* 'Item_No', 'Qty_Delivered' or 'Cost'.

1.4 Foreign Keys

If an attribute of one table refers to the **primary key** of a second table, it is a **foreign key** in the first table. Foreign keys link tables to form potentially complex data structures. The tables above are related because each row of the 'Updates' table has an 'Account' that matches a row of the 'Suppliers' table, and some have an 'Item_no' that matches a row of the 'Products' table. 'Account' and 'Item_No' are foreign keys in the 'Updates' table. In addition, 'Supplier' in the 'Products' table is a foreign key to the 'Suppliers' table.

We can follow the foreign key links between tables. For example, the first row of the 'Updates' table has an 'Item_No' attribute of 'ALTCIU'. This refers to the second row of the 'Products' table, the 'Altec Caller ID Unit'. This, in turn, refers to the 4th row of the 'Suppliers' table, 'Pacific Auto Electronics'. As it happens, this is the same supplier indicated by the 'Account' attribute of the first row of the 'Updates' table. What it means in this case is that the delivery concerned *was* obtained from the approved supplier.

2 Creating Schemas

2.1 Creating Tables

A database schema is described using the data definition language (DDL). The **create table** statement is used to define a new table. It specifies the name of the table, the names and data types of its columns, and a set of table constraints:!

```
create table Suppliers (
    Account          char(4),
    Name             char(30),
    Street           char(30),
    Suburb           char(30),
    Balance          number(8,2),
    primary key      (Account)
    );
```

In this example, the only table constraint is that the primary key of the table is 'Account'. Every table should have a **primary key** constraint.![3]

When a **create table** statement is executed, the table definition is added to the database schema, and remains there until it is deliberately removed. Creating a table does not place any information in the table. It is therefore the *rough* equivalent of storing a Cobol record definition in a library file for later use.

2.2 Data types

Among others, *Oracle* SQL provides the following data types:

char(*w*)	A string of *w* characters,
vchar(*w*)	A string of not more than *w* characters,
decimal(*w*,*d*)	A number with (*w*-*d*) digits before and *d* digits after the decimal point,
number(*w*,*d*)	Same as decimal(*w*,*d*),
integer(*w*)	Same as decimal(*w*,0),
float	A floating point number,
date	The date and time to the nearest second.

Money may be represented either as **integer** cents, or as **number**(*w*,2) dollars.

---

[3] The relational database model logically demands that a primary key should be specified, because duplicate rows are not allowed. But the clause is actually optional, because early versions of SQL did not support it.

A number that would be described as '**pic** 9(***6***)v9(2)' in Cobol is described as '**number**(***8***,2)' in SQL, because the number before the comma is the *total* number of decimal places.

In SQL, if a column may never contain a null value, it should be given a **not null** constraint. Every row added to the table must then contain a value in that column. Primary keys are **not null** implicitly. In *Oracle* SQL the previous example may be written as follows:

```
create table Suppliers (
    Account          char(4),
    Name             char(30) not null,
    Street           char(30) not null,
    Suburb           char(30) not null,
    Balance          number(8,2) not null,
    primary key      (Account)
    );
```

We would create the 'Customers' table in similar style.

```
create table Customers (
    Account          char(4),
    Name             char(30) not null,
    Street           char(30) not null,
    Suburb           char(30) not null,
    Balance          number(8,2) not null,
    Credit_Limit     number(6) not null,
    Available_Credit number(8,2) not null,
    primary key      (Account)
    );
```

The 'Products' table introduces the idea of a **foreign key** constraint.

```
create table Products (
    Item_No          char(6),
    Description      char(40) not null,
    Supplier         char(4) not null,
    Reorder_Level    number(4) not null,
    Reorder_Qty      number(4) not null,
    On_Order         number(4) not null,
    Stock            number(4) not null,
    Price            number(6,2) not null,
    Valuation        number(8,2) not null,
    primary key      (Item_no),
    foreign key      (Supplier) references Suppliers(Account)
    );
```

The meaning of the foreign key constraint is that the value of 'Supplier' in a *child* row of the 'Products' table must equal the 'Account' of a *parent* row in the 'Suppliers' table. A parent may have many children, but a child has only one parent.

An *Oracle* **date** type *includes* the date and time of day. *Oracle* dates are stored in the database as fixed-point numbers. The integer part measures the number of days since a certain date in the distant past. The fractional part gives the time of day. However, this is not the ANSI standard, which has separate **date** and **time** types. For simplicity, we shall treat 'YY_MM_DD' and 'HH_MM_SS' as 8-character strings here, eg., '01/12/31'.

The 'Updates' table also illustrates how to define a composite primary key:

```
create table Updates (
    YY_MM_DD         char(8),
    HH_MM_SS         char(8),
    Kind             char(1) not null,
    Item-No          char(6),
    Account          char(4),
    Qty_Delivered    number(4),
    Cost             number(8,2),
    Amount           number(8,2),
    primary key      (YY_MM_DD, HH_MM_SS),
    foreign key      (Item_No) references Products(Item_No),
    foreign key      (Account) references Suppliers(Account)
    );
```

The 'Updates' table has *two* foreign key constraints. Both foreign keys can be null. Therefore 'Item_No' can either be null or the 'Item_No' of a parent row of the 'Products' table, and 'Account' can either be null or the key of a parent row of the 'Suppliers' table.

It is important to realise that the names of tables and the names and types of columns defined by DDL statements are *persistent*, ie., they are recorded permanently *in the database itself*.[4] The same names must be used whenever the database is queried or updated.![5]

There must be no comma immediately before a right parenthesis.

2.3 Ensuring Uniqueness

SQL always ensures that two rows cannot be given the same primary key. In addition, ANS SQL allows *any* column to be given the **unique** property. Here is how we could ensure that no two 'Products' rows have the same value of 'Description':

```
create table Products (
    Item_No            char(6),
    Supplier           char(4) not null,
    Description        char(40) not null unique,
    Reorder_Level      number(4) not null,
    Reorder_Qty        number(4) not null,
    On_Order           number(4) not null,
    Stock              number(4) not null,
    Price              number(6,2) not null,
    primary key        (Item_no),
    foreign key        (Supplier) references Suppliers(Account)
    );
```

Alternatively, it is possible to create an **index** on a column or group of columns.! Indices serve two purposes: First, given a value, they provide rapid access to the row or rows of the table that contain that value — specifically, in much less time that it takes to scan the whole table. This is important in real-life applications, where databases can be very large.

For example, we might want to make it easy to find all products with a given 'Supplier':

```
create index Products_Supplier on Products(Supplier);
```

Second, indexes can efficiently ensure that all values in a column are different.

For example, we can ensure that no two products have the same 'Description':

```
create unique index Products_Description on Products(Description);
```

However, the same effect is achieved more simply by specifying that the 'Description' is **unique** when creating the 'Products' table. A **unique** constraint effectively implies that an index will be constructed; that is the quickest way to check for duplicates.

Whenever *any* set of columns uniquely identifies a row, it is potentially a primary key. However, a table may have only *one* primary key (although the primary key may be composite), because, as in a Cobol indexed file, the primary key determines the order in which its rows are stored. Alternative primary keys, called **candidate keys**, have to be implemented by **unique** indices. Accessing a table using its primary key may be expected to be slightly more efficient than accessing it using a secondary index.

Contrast the difference between *candidate* keys and *composite* keys: Suppose X and Y are two columns of table T. If X is the primary key and Y is a unique candidate key, then a unique row of T may be retrieved given *either* X *or* Y. On the other hand, if (X,Y) is the composite primary key of T, then a unique row of T can only be retrieved given *both* X *and* Y. The choice is determined by the nature of the data: does either X or Y individually identify a unique row, or does only their combination identify a row uniquely?

---

4 In Oracle, the properties of tables may be retrieved by typing '**select** * **from** tabs;' and the properties of columns by typing '**select** * **from** cols;'

5 In practice, all SQL statements are made only after the user has selected a specific database — a bit like setting one's current directory. However, since this is usually done through operating system environment variables, the user may be unaware of it.

2.4 Changing Schemas

After a table schema has been defined, it is still possible to add a new column to it, using the **alter table** statement:!

```
alter table Suppliers add Total_Valuation number(10,2);
```

New columns are always added to the right of the existing columns. It is impossible to add a new column with the **not null** property unless the table is empty, because every row must contain a value in that column, and clearly, none of them do. Depending on the DBMS, it is usually possible to first add the new column, then insert data into it in every row, then finally to specify **not null**.

When the **alter** statement will not do, it is usually possible to **create** a new table with the desired properties, then to copy data from an existing table into the new one.

If necessary, a table — *and all the data it contains* — may be removed from the database using the **drop table** statement:!

```
drop table Updates;
```

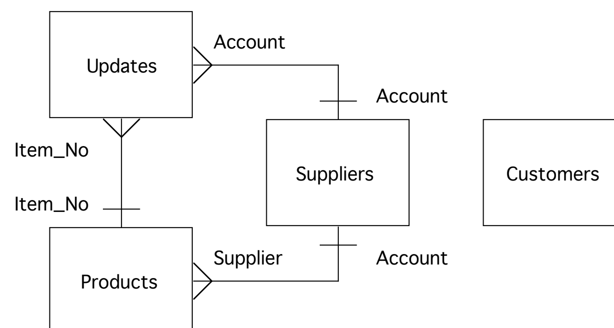
2.5 Displaying a Schema

Oracle SQL*Plus provides a **describe** statement, which lists the columns of a table:

```
describe Updates;
NAME               NULL?        TYPE
--------------     --------     -------------------
YY_MM_DD           NOT NULL     CHAR(8)
HH_MM_SS           NOT NULL     CHAR(8)
KIND               NOT NULL     CHAR(1)
ITEM-NO                         CHAR(6)
ACCOUNT                         CHAR(4)
QTY_DELIVERED                   NUMBER(4)
COST                            NUMBER(8,2)
AMOUNT                          NUMBER(8,2)
```

2.6 Entity-Relationship Diagrams

The relationships between tables can be summarised by an Entity-Relationship Diagram.



Boxes represent tables, and lines represent relations between them. A bar across the line indicates 'one' and a trident indicates 'many'. (So the absence of a bar or trident is an error.) Usually, the 'one' side of a relation is a primary key belonging to the *parent* table, and the 'many' side is a foreign key belonging to the *child* or *dependent* table.

It is obvious that there are two ways to relate an row of the 'Updates' table to a row of the 'Suppliers' table: One is directly on 'Account'. This is the link that expresses the idea of the supplier who actually delivers a product. The second is *via* the 'Products' table. This expresses the idea of the supplier who *normally* supplies the product that is delivered. The two suppliers involved are not always the same.

Unusually, in this schema there is no connection between the 'Customers' table and the other tables. This would change if we were to take more kinds of transactions into account. For example, in a complete system there would be additional kinds of transaction to deal with sales orders placed by customers for products, and payments from customers.

3 Populating a Database

3.1 Inserting Rows

Once a table has been declared, it needs to be **populated**, using the SQL data manipulation language (DML) **insert** statement:!

```
insert into Suppliers values ('B007','Blue Waters Nominees          ',
                                     '1 Francis Road                ',
                                     'Paramatta NSW 2150            ', 120459.56);
insert into Suppliers values ('N001','Netherlands Electrical Supply ',
                                     '124 Burbridge Road            ',
                                     'Hilton SA 5033                ',    127.99);
insert into Suppliers values ('N002','Nippon Electronics Importers  ',
                                     '19 Ashford Rd                 ',
                                     'Redfern NSW 2016              ',   2790.46);
```

Character strings must be enclosed inside *single* quotes. Numbers must not.

When a table has columns that can be null, unless all the attributes are specified, it is necessary to say which columns will receive values:

```
insert into Updates
    (YY_MM_DD, HH_MM_SS, Kind, Item_No, Account, Qty_Delivered, Cost)
    values ('01/02/06', '14:55:14', 'D', 'ALTCIU', 'P004', 10, 100.00);
insert into Updates
    (YY_MM_DD, HH_MM_SS, Kind, Account, Amount)
    values ('01/02/06', '14:56:23', '$', 'N002', 2790.46);
```

It is also possible to **insert** a *set* of rows, taken from another table:

```
insert into Products rows (select * from Old_Products);
```

Or to *create* a table as a copy of an existing table:

```
create table Products as (select * from Old_Products);
```

If one ignores the effort put into defining the schema, this one line is the equivalent of the entire Cobol program in Section 3.3 of *File Processing in Cobol 85*.

In addition, most DBMS's provide a means to load table data from text files. [6]

3.2 Formatting Columns

Oracle allows the user to interact with a database using a tool called SQL*Plus. Among other features, SQL*Plus provides a way to improve the readability of the output from SQL queries. Each column may be given a heading and a format, as follows:

```
column YY_MM_DD         heading 'YY_MM_DD'       format a8;
column HH_MM_SS         heading 'HH_MM_SS'       format a8;
column Kind             heading 'Kind'           format a4;
column Item_No          heading 'Item #'         format a6;
column Account          heading 'Acct'           format a4;
column Supplier         heading 'Acct'           format a4;
column Qty_Ordered      heading 'Orderd'         format 9,990;
column Qty_Delivered    heading 'Delvrd'         format 9,990;
column Qty_Sold         heading 'Sold'           format 9,990;
column Payment          heading 'Payment'        format $999,990.00;
column Price            heading 'Price'          format $9,990.00;
column Unit_Cost        heading 'Unit Cost'      format $9,990.00;
column Cost             heading 'Cost'           format $999,990.00;
column Amount           heading 'Amount'         format $999,990.00;
column Description      heading 'Description'    format a40;
column Name             heading 'Business Name'  format a30;
column Street           heading 'Number, Street' format a30;
column Suburb           heading 'Suburb, State'  format a30;
column Balance          heading 'Balance'        format $999,990.00;
column Credit_limit     heading 'Cr. Limit'      format $999,990;
column Available_Credit heading 'Avail. Credit'  format $999,990.00;
column Back_Orders      heading 'Back Orders'    format $999,990.00;
column Stock            heading 'Stock'          format 9,990;
```

6 *Oracle* provides the utility program 'sqlload' for this.

```
column On_Order         heading 'OnOrd'        format 9,990;
column Reorder_Level    heading 'RO.Lvl'       format 9,990;
column Reorder_Qty      heading 'RO.Qty'       format 9,990;
column Valuation        heading 'Valuation'    format $999,990.00;
```

A format such as '`a8`' is equivalent to Cobol '**`pic`** `x(8)`', '`9,990`' to '**`pic`**! `--,--9`', and '`$999,990.00`' to '**`pic`**! `-$$$$,$$9.99`'. The headings are chosen to fit the width of the column, given its format. Note that 'Kind' is padded to 4 characters, otherwise its heading would show as 'K'. These formats are assumed in all the examples that follow.

In addition to the above formats for columns appearing in the schema, some additional formats are specified below for computed results that will appear in later examples:

```
column Total_Qty        heading 'TotalQty'       format 999,990;
column Total_Cost       heading 'Total Cost'     format $999,990.00;
column Total_Amount     heading 'Total Amount'   format $9,999,990.00;
column Total_Value      heading 'Total Value'    format $999,990.00;
column Total_Paid       heading 'Total Paid'     format $999,990.00;
column Total_Balance    heading 'Total Balance'  format $9,999,990.00;
column Total_Ordered    heading 'Total Ordrd'    format 99,999,990;
```

4 Projection & Selection

4.1 Projection

Information is retrieved from the database using the SQL **select** statement.! The **from** option determines which tables are used. The following example lists all the product descriptions:

```
select Description from Products;
Description
-----------
Alcatel One Touch Phone
Altec Caller ID Unit
Altec Portable CD Including Headphones
Audioboss Deluxe 60W Car Radio Cassette
Audio-Gods Box Speakers
Canon Colour Bubble Jet Printer
Canon Flatbed Scanner
Fax Rolls
Freedom 6.5" Stereo Speakers
and so on ...
```

Choosing columns is called **projection**:[7]

```
select Name, Balance from Suppliers;
Name                                 Balance
----                                 -------
Blue Waters Nominees             $120,459.56
Netherlands Electrical Supply     $15,077.99
Nippon Electronics Importers         -$50.00
Pacific Auto Electronics          $11,226.06
South Aust. Import Trading             $0.00
US Audio Imports                     -$45.50
```

When *all* the columns of a table are required, in the order of the schema, an asterisk may be used instead of a list of column names:

```
select * from Suppliers;
Acct Business Name                  Number, Street      Suburb, State          Balance
---- -------------                  --------------      -------------      -----------
B007 Blue Waters Nominees           1 Francis Road      Paramatta NSW 2150 $120,459.56
N001 Netherlands Electrical Supply  124 Burbridge Road  Hilton SA 5033      $15,077.99
N002 Nippon Electronics Importers   19 Ashford Rd       Redfern NSW 2016       -$50.00
P004 Pacific Auto Electronics       PO Box 407          Sydney NSW 2000     $11,226.06
S002 South Aust. Import Trading     137 Burbridge Road  Hilton SA 5033           $0.00
U003 US Audio Imports               5 Penna Ave         Clayton  VIC 3109      -$45.50
```

---

[7] As in Coordinate Geometry, where, for example, a 3-dimensional solid is projected onto 2 dimensions by dropping one coordinate.

The following query discovers which products have been delivered:

```
select Item_No from Updates;
Item #
------
ALTCIU
ALTPCD
PLPDCD
PLPRCM
ALTCIU
ALTCIU
ALTCIU
```

We see that 'ALTCIU' has been listed 4 times.

4.2 All or Distinct

The previous example could have been written as:

```
select all Item_No from Updates;
```

All is the default. The **distinct** option removes any duplicate rows from the result:

```
select distinct Item_No from Updates;
Item_No
-------
ALTCIU
ALTPCD
PLPDCD
PLPRCM
```

Note that the results are presented in alphabetical order. That is because the quickest way to remove duplicates is to sort the results. This is exactly what we would have to do in Cobol. Again, one line of SQL is equivalent to an entire Cobol program.

4.3 Arithmetic Operators

The word **select** may be followed by a list of *expressions*, rather than column names. Values of attributes and constants can be combined using the operators *, /, + and –, and the usual range of mathematical functions. Evaluation is left to right, with * and / having higher precedence than + and –. Parentheses may be used to force any desired order of evaluation.

The following query, which is equivalent to the Cobol program of Section 6.3 of *File Processing in Cobol 85*, uses a **column alias** to give a name to a computed expression. Note that a comma is needed between expressions, but no comma is allowed between an expression and its column alias.

```
select Account, Name, (Credit_Limit - Available_Credit - Balance) Back_Orders
         from Customers;
Acct Business Name                    Back Orders
---- ------------------------------ ------------
A001 Autobarn Elizabeth                    -$0.50
B003 BCR Mobile Installations               $0.00
B007 Blaupunkt                          $2,168.00
B012 Bobs Electronic Repairs               -$0.50
C002 Car Audio Designs                      $0.00
C005 Car Audio Services                     $0.00
C007 Cargear Pty Ltd                      $135.00
C010 Cartronics                             $0.00
C020 Citisound                            $181.00
C027 Complete Audio                         $0.00
C031 Custom Audio Sound                   $362.00
and so on ...
```

4.4 Selection

Rows are selected using a **where** condition.! In effect, each row of a table is examined, but only those rows for which the condition is true are displayed.![8]

Where conditions may include the operators '+', '–', '*' and '/', mathematical functions, the relations '=', '<', '>', '<=', '>=', and the boolean operators **and**, **or** and **not**. However, '<>' is the 'not equals' operator, and there is no '**' operator. The precedence of operators is:

- * and /
- + and –
- comparison operators
- **not**
- **and**
- **or**

Parentheses may be used to force any desired order of evaluation.

The following query displays all attributes of customers that have an non-zero value of goods on back order, and is equivalent to the Cobol program of Section 6.1 of *File Processing in Cobol 85*.

```
select * from Customers where Available-Credit <> Credit_Limit - Balance;
Acct Business Name             Number, Street         Suburb, State            Balance  Cr.LimitAvail.Credit
---- ---------------------     ---------------------- -----------------      ---------------------------------
A001 Autobarn Elizabeth        61 Elizabeth Way       Elizabeth  SA 5158       $208.50    $5,000   $4,792.00
B007 Blaupunkt            Cnr Centre and McNaughton RdsClayton  VIC 3109     $10,295.00   $25,000  $12,537.00
B012 Bobs Electronic Repairs   28 Limbert Avenue      Seacombe Gardens  SA 5047 -$129.50    $1,000   $1,130.00
C007 Cargear Pty Ltd           453 Magill Road        St Morris  SA 5059         $0.00    $1,000     $865.00
C020 Citisound                 141-145 Franklin Street Adelaide  SA 5000        $35.00    $5,000   $4,784.00
C031 Custom Audio Sound        Unit 2 798 Marion Road Marion  SA 5045            $0.00    $1,000     $638.00
D015 Doug Sunstroms Sound Mart 6 Smart Road           Modbury  SA 5143         -$30.50    $1,000   $1,031.00
E003 Electric Bug Pty Ltd      199-203 Torrens Road   Croydon  SA 5013       $1,305.00    $5,000   $3,452.00
F002 Fujitsu Ten (Australia) P/L 75 Westgate Drive    Altona North VIC 3039  $1,182.00    $2,000   $1,818.00
N014 Northern Car Radio        1445 Main North Road   Para Hills West  SA 5150   $0.00    $1,000     $843.00
R003 RS Automotive Development 19 Warburton Road      Valley View  SA 5056   $4,720.85    $5,000     $279.00
S004 Sound 4 Australia Pty Ltd 22 Pinn Street         St Marys  SA 5038      $2,758.75    $5,000   $2,241.00
S015 Strathfield Car Radios    316 Gouger Street       Adelaide  SA 5000         $3.50    $1,000     $997.00
T003 Tonkins Car Audio Pty Ltd 116 Sherriffs Road     Morphett Vale  SA 5153   $492.50    $1,000     $481.00
```

Not surprisingly, projection and selection can be combined in the same query. The following query is equivalent the example of Section 6.3 of *File Processing in Cobol 85*.

```
select Account, Name, Credit_Limit - Available_Credit - Balance Back_Orders
         from Customers
         where Available-Credit <> Credit_Limit - Balance;
Acct Business Name                    Back Orders
---- ------------------------------ ------------
A001 Autobarn Elizabeth                    -$0.50
B007 Blaupunkt                          $2,168.00
B012 Bobs Electronic Repairs               -$0.50
C007 Cargear Pty Ltd                      $135.00
C020 Citisound                            $181.00
C031 Custom Audio Sound                   $362.00
D015 Doug Sunstroms Sound Mart             -$0.50
E003 Electric Bug Pty Ltd                 $243.00
F002 Fujitsu Ten (Australia) P/L      -$1,000.00
N014 Northern Car Radio                   $157.00
R003 RS Automotive Development              $0.15
S004 Sound 4 Australia Pty Ltd              $0.25
S015 Strathfield Car Radios                -$0.50
T003 Tonkins Car Audio Pty Ltd             $26.50
```

A common type of query is to retrieve a row with a given key:

```
select * from Customers where Account = 'C007';
Acct Business Name         Number, Street        Suburb, State           Balance Cr. Limit Avail. Credit
---- --------------------- --------------------- ----------------- ------------ --------- -------------
C007 Cargear Pty Ltd       453 Magill Road       St Morris  SA 5059        $0.00    $1,000       $865.00
```

8 There is a lack of consistency here in the use of the term 'selection'. The '**select** … **where** …' statement would more logically be '**project** … **select** …'.

The above query is equivalent to a specific use of the ‘findcust’ program of Section 5.7 of *File Processing in Cobol 85*. This query would almost certainly be executed by random access, taking advantage of the primary key index of the ‘Customers’ table. In contrast, the earlier queries require each row of the table to be scanned using sequential access. *There is no difference in syntax*.

In theory, the user of the SQL query sub-language need not be aware of the details of the database schema. For example,

```
select Stock from Products where Description = 'Altec Caller ID Unit';
 Stock
------
    12
```

is a valid query independently of whether the ‘Products’ table has a ‘Description’ index, or equivalently, whether ‘Description’ is declared to be **unique**. If such an index is created, all that should happen is that the query should take less time to execute.

Strings are compared from left to right according to the ASCII codes of their characters. Apart from upper/lower-case differences, this is the same as dictionary order. To ignore case, the ‘upper’ or ‘lower’ functions may be used:

```
select Stock from Products where lower(Description) = 'altec caller id unit';
 Stock
------
    12
```

Multiple comparisons may be combined using the boolean operators, **and**, **or**, and **not**. The following query finds the ‘Item_no’, ‘Description’ and ‘Reorder_Qty’ of all products that need re-ordering from the suppliers with accounts ‘P004’ and ‘U002’.

```
select Item_No, Description, Reorder_Qty from Products
      where (Supplier = 'P004' or Supplier = 'U002')
      and Stock + On_Order < Reorder_Level;
Item # Description                              RO.Qty
------ ---------------------------------------- ------
ACLOTP Alcatel One Touch Phone                      20
HOMPYS Home Party Speakers                          20
KITRCA Sansui Flipface+Amp with 4"+6x9" Spkrs       20
RDMDRC Roadmaster Detachable Face Radio Cass.       20
SUIDRC Sansui Hi-Powered Detachable Radio Cass.     50
SYODRC Sanyo Detachable Face Radio Cassette         50
SYOPCP Sanyo Portable Cassette Player               50
```

4.4.1 Range Testing

SQL provides a **between** operator for testing a range of values. The following query displays all supplier rows with $10,000 ≤ Balance ≤ $100,000.

```
select * from Suppliers
        where Balance between 10000 and 100000;
Acct Business Name                  Number, Street               Suburb, State                 Balance
---- ------------------------------ ---------------------------- ------------------------- ------------
N001 Netherlands Electrical Supply  124 Burbridge Road           Hilton SA 5033              $15,077.99
P004 Pacific Auto Electronics       PO Box 407                   Sydney NSW 2000             $11,226.06
```

The **between** operator may be preceded by **not**. The following query displays all supplier rows with Balance < $10,000 or Balance > $100,000.

```
select * from Suppliers
        where Balance not between 10000 and 100000;
Acct Business Name                  Number, Street               Suburb, State                 Balance
---- ------------------------------ ---------------------------- ------------------------- ------------
B007 Blue Waters Nominees           1 Francis Road               Paramatta NSW 2150         $120,459.56
N002 Nippon Electronics Importers   19 Ashford Rd                Redfern NSW 2016               -$50.00
S002 South Aust. Import Trading     137 Burbridge Road           Hilton SA 5033                   $0.00
U003 US Audio Imports               5 Penna Ave                  Clayton  VIC 3109              -$45.50
```

4.4.2 Pattern Matching

It is possible to compare a character column with a pattern (in that order) using the **like** operator. A pattern may contain wild cards: ‘%’ represents any *string*, including an empty string; ‘_’ represents any *single character*.

The following query retrieves the 'Item_No' and 'Description' of each product row in which the description includes the string 'speaker' in upper or lower case letters.

```
select Item_no, Description from Products
    where upper(Description) like '%SPEAKER%';
Item # Description
------ ----------------------------------------
AUGBOX Audio-Gods Box Speakers
FDM6SS Freedom 6.5" Stereo Speakers
HOMPYS Home Party Speakers
SYOCSS Sanyo Deluxe Car Stereo Speakers
```

4.4.3 Testing Null Values

The following query finds every row of the 'Updates' table that does not have an 'Item_No'.

```
select * from Updates where Item_No is null;
YY_MM_DD HH_MM_SS Kind Item # Acct Delvrd         Cost       Amount
-------- -------- ---- ------ ---- ------ ------------ ------------
01/02/06 14:56:23 $           N002                        $2,790.46
01/02/06 15:01:07 $           N002                           $50.00
01/02/14 15:56:17 $           N002                        $2,790.46
```

Note that one cannot write '`Item_No = null`' or '`Item_No <> null`'. If something is **null** it doesn't *have* a value. Null values are *not* zeros — and they are not empty strings either.

Because these same rows also have no 'Qty_Delivered', they fail any test involving it:

```
select * from Updates where Qty_Delivered <> 0;
YY_MM_DD HH_MM_SS Kind Item # Acct Delvrd         Cost       Amount
-------- -------- ---- ------ ---- ------ ------------ ------------
01/02/06 14:55:14 D    ALTCIU P004     10      $100.00
01/02/06 14:57:19 D    ALTPCD P004     30      $900.00
01/02/06 14:58:44 D    PLPDCD N001     50   $14,950.00
01/02/06 15:00:02 D    PLPRCM N001     50    $4,550.00
01/02/06 15:01:40 D    ALTCIU N001     10       $90.00
01/02/14 15:54:15 D    ALTCIU P004     10      $100.00
01/02/14 15:55:50 D    ALTCIU P004     10      $100.00
```

In a two-valued logic, rows with no 'Qty_Delivered' would *fail* the test '`Qty_Delivered = 0`' in the following query, but because of the negation, they would then *satisfy* the **where** condition. However, SQL is more sensible than this, and produces the expected result:

```
select * from Updates where not (Qty_Delivered = 0);
YY_MM_DD HH_MM_SS Kind Item # Acct Delvrd         Cost       Amount
-------- -------- ---- ------ ---- ------ ------------ ------------
01/02/06 14:55:14 D    ALTCIU P004     10      $100.00
01/02/06 14:57:19 D    ALTPCD P004     30      $900.00
01/02/06 14:58:44 D    PLPDCD N001     50   $14,950.00
01/02/06 15:00:02 D    PLPRCM N001     50    $4,550.00
01/02/06 15:01:40 D    ALTCIU N001     10       $90.00
01/02/14 15:54:15 D    ALTCIU P004     10      $100.00
01/02/14 15:55:50 D    ALTCIU P004     10      $100.00
```

The next query lists the price per unit of some rows in the 'Updates' table:

```
select YY_MM_DD, HH_MM_SS, Kind, Item_No, Account,
    Qty_Delivered, Cost, Cost/Qty_Delivered Unit_Cost
    from Updates;
YY_MM_DD HH_MM_SS Kind Item # Acct Delvrd         Cost  Unit Cost
-------- -------- ---- ------ ---- ------ ------------ ----------
01/02/06 14:55:14 D    ALTCIU P004     10      $100.00     $10.00
01/02/06 14:56:23 $           N002
01/02/06 14:57:19 D    ALTPCD P004     30      $900.00     $30.00
01/02/06 14:58:44 D    PLPDCD N001     50   $14,950.00    $299.00
01/02/06 15:00:02 D    PLPRCM N001     50    $4,550.00     $91.00
01/02/06 15:01:07 $           N002
01/02/06 15:01:40 D    ALTCIU N001     10       $90.00      $9.00
01/02/14 15:54:15 D    ALTCIU P004     10      $100.00     $10.00
01/02/14 15:55:50 D    ALTCIU P004     10      $100.00     $10.00
01/02/14 15:56:17 $           N002
```

SQL deals with these situations by using a *three-valued logic*: **true**, **false** and **unknown**. A comparison involving a null attribute yields the value **unknown**. Think of **true** as 1, **false** as 0, and **unknown** as 0.5. The boolean operators then obey the following rules:

X **and** Y = min(X,Y),
X **or** Y = max(X,Y),
not X = 1 - X.

Only **where** clauses that evaluate to **true** are included in the answer to a query; **false** and **unknown** are both excluded.

To avoid problems in arithmetic, if one of the operands of an expression is null, the whole expression is null. Null results are not displayed. If '`X`' is null, '`0*X`' and '`X-X`' both yield null.

4.4.4 System Variables

All SQL systems have a variable called **user** that contains the user's log-in username. In *Oracle* SQL, the variable **sysdate** contains the current date *and time*: '`sysdate+1`' is tomorrow, '`sysdate-1`' is yesterday, and so on.

4.4.5 Set Membership

It is possible to test for membership of a set. A set is written as a list of values separated by commas, within parentheses. The following query displays each row of 'Suppliers' whose 'Account' is in the set {B007, S002, U003}.

```
select * from Suppliers where Account in ('B007', 'S002', 'U003');
Acct Business Name                  Number, Street                 Suburb, State                 Balance
---- ------------------------------ ------------------------------ ----------------------- ------------
U003 US Audio Imports               5 Penna Ave                    Clayton  VIC 3109             -$45.50
S002 South Aust. Import Trading     137 Burbridge Road             Hilton SA 5033                  $0.00
B007 Blue Waters Nominees           1 Francis Road                 Paramatta NSW 2150        $120,459.56
```

The **in** operator may be preceded by **not**:

```
select * from Suppliers where Account not in ('B007', 'S002', 'U003');
Acct Business Name                  Number, Street                 Suburb, State                 Balance
---- ------------------------------ ------------------------------ ----------------------- ------------
N001 Netherlands Electrical Supply  124 Burbridge Road             Hilton SA 5033             $15,077.99
N002 Nippon Electronics Importers   19 Ashford Rd                  Redfern NSW 2016              -$50.00
P004 Pacific Auto Electronics       PO Box 407                     Sydney NSW 2000            $11,226.06
```

Set operations are particularly important in connection with nested queries, discussed in Section 8.

5 Grouping and Ordering

These two ideas are easily confused because they both involve sorting data. **Ordering** refers only to the order in which rows are displayed; **grouping** refers to finding **aggregates**, such as totals or averages.

5.1 Ordering

The following query displays the rows of the Customers table in descending order of 'Balance'. It is therefore the equivalent of the Cobol program in Section 7.2 of *File Algorithms in Cobol 85*.

```
select * from Customers order by Balance desc;
Acct Business Name                Number, Street      Suburb, State            Balance Cr. Limit Avail. Credit
---- --------------------------- ----------------    --------------------  ---------- --------- -------------
B007 Blaupunkt            Cnr Centre and McNaughton Rds Clayton  VIC 3109   $10,295.00   $25,000    $12,537.00
R003 RS Automotive Development   19 Warburton Road   Valley View  SA 5056    $4,720.85    $5,000       $279.00
S004 Sound 4 Australia Pty Ltd   22 Pinn Street      St Marys  SA 5038       $2,758.75    $5,000     $2,241.00
E003 Electric Bug Pty Ltd        199-203 Torrens Road Croydon  SA 5013       $1,305.00    $5,000     $3,452.00
F002 Fujitsu Ten (Australia) P/L 75 Westgate Drive   Altona North VIC 3039   $1,182.00    $2,000     $1,818.00
E007 Afrotechnics                117 Port Road       Hindmarsh  SA 5014      $1,038.00        $0    -$1,038.00
S011 Southern Car Audio          208 Dyson Road      Lonsdale  SA 5157       $1,038.00    $2,000       $962.00
... etc., until ...
G010 Global Car Audio            265 Gouger Street   Adelaide  SA 5000           $0.00    $1,000     $1,000.00
C031 Custom Audio Sound          Unit 2 798 Marion Road Marion  SA 5045          $0.00    $1,000       $638.00
N012 National Car Audio          208 Dyson Road      Lonsdale  SA 5157         -$20.00    $1,000     $1,020.00
D015 Doug Sunstroms Sound Mart   6 Smart Road        Modbury  SA 5143          -$30.50    $1,000     $1,031.00
B012 Bobs Electronic Repairs     28 Limbert Avenue   Seacombe Gardens SA 5047 -$129.50    $1,000     $1,130.00
```

The abbreviation '**desc**' means descending; '**asc**' means ascending. The default is '**asc**'. The following query sorts on more than one attribute:

```
select * from Products order by Stock, On_Order;
Item # Description                              Acct RO.Lvl RO.Qty  OnOrd  Stock     Price   Valuation
------ ---------------------------------------- ---- ------ ------ ------ ------ --------- -----------
JBLHSW JBL Hi-Power Sub Woofer                  N002     10     20     20      0    $62.50       $0.00
KKR10W Kicker 10" Power Sub Woofer              U003     20     50     50      0   $149.50       $0.00
AUBCRC Audioboss Deluxe 60W Car Radio Cassette  U003      5     10     10      1    $34.00      $25.00
NYS250 New York Sound 250 Watt Amplifier        U003      5     10     10      1    $75.50      $59.50
KENFCD Kenwood 160 Watt Flipface CD Tuner       N002      5     10     10      1   $259.50     $197.65
SST2CA Soundstream 2 Channel Amplifier          U003     10     20     20      1   $150.50     $112.50
KEN3CD Kenwood 3 Disc Carousel CD Changer       N002     20     50     50      2   $150.50     $225.00
PLPDCT Philips 120 Watt Detachable CD Tuner     N001     20     50     50      2   $150.00     $216.00
OMNLRC Omni Long Reach Digital Cordless         P004     10     20     20      3   $150.00     $315.00
NKAHFK Nokia Genuine In-Car Handsfree Kit       P004     10     20     24      3   $145.00     $315.00
ALTPCD Altec Portable CD Including Headphones   P004     10     30     30      3    $37.00      $75.00
AUGBOX Audio-Gods Box Speakers                  U003      5     10      0      4    $35.00      $90.00
... etc., until ...
SHP16K Sharp 16KB Organiser                     N002     50    100      0     79    $22.00   $1,364.57
VCAPLE Voca Phone Line Extension Cord           S002     50    100      0     84     $2.00      $92.40
SNYPDM Sony Portable Discman                    N002     50    100      0    110    $89.50   $7,775.90
FDM6SS Freedom 6.5" Stereo Speakers             P004     50    100      0    124    $18.50   $1,409.67
FAXROL Fax Rolls                                S002    200  1,000      0    243     $0.95     $197.36
TVLCDC Travel CD Case                           S002    200  1,000      0  1,004     $3.50   $2,510.00
PLPSEP Philips Stereo Earphones                 N001    250  1,000      0  1,196     $3.50   $2,392.00
```

5.2 Aggregate Functions

SQL offers **aggregate functions** for finding maxima, minima, averages, totals and counts. These are called **max**, **min**, **avg**, **sum** and **count.** The functions **avg** and **sum** are defined for numeric data only.

The following query finds the total amount that is owed to suppliers.

```
select sum(Balance) from Suppliers;
SUM(BALANCE)
------------
   146668.11
```

The next query counts the number of rows of the 'Suppliers' table.

```
select count(*) from Suppliers;
  COUNT(*)
----------
         6
```

The final query finds the average balance owed to each supplier.

```
select avg(Balance) from Suppliers;
AVG(BALANCE)
------------
   24444.685
```

5.3 Grouping

Aggregates are useful in summarising data about groups of related rows, therefore they are often associated with *grouping*.

Suppose we want to find total amounts charged by each supplier for deliveries, and the total amounts paid to each supplier. This is similar to the Cobol program in Section 7.3 of *File Algorithms in Cobol 85*.

```
select Account, sum(Cost) Total_Cost, sum(Amount) Total_Amount
      from Updates
      group by Account;
Acct   Total Cost   Total Amount
---- ------------ --------------
N001   $19,590.00
N002                   $5,630.92
P004    $1,200.00
```

In calculating any aggregate, null values are *ignored*.

Unlike the Cobol program, a separate query is needed to compute the grand totals:

```
select sum(Cost) Total_Cost, sum(Amount) Total_Amount from Updates;
  Total Cost   Total Amount
------------ --------------
  $20,790.00      $5,630.92
```

A similar pair of queries deals with the Cobol program in Section 7.5 of *File Algorithms in Cobol 85*:

```
select Item_No, sum(Qty_Delivered)  Total_Qty,
             sum(Cost) Total_Cost,
             sum(Cost)/sum(Qty) Unit_Cost
    from Updates
    group by Item_No;
Item # TotalQty   Total Cost  Unit Cost
------ -------- ------------ ----------
ALTCIU       40      $390.00      $9.75
ALTPCD       30      $900.00     $30.00
PLPDCD       50   $14,950.00    $299.00
PLPRCM       50    $4,550.00     $91.00
```

and,

```
select sum(Qty_Delivered) Total_Qty, sum(Cost) Total_Cost,
        sum(Cost)/sum(Qty_Delivered) Unit_Cost from Updates;
TotalQty   Total Cost  Unit Cost
-------- ------------ ----------
     170   $20,790.00    $122.29
```

Notice that when results are grouped, they appear in ascending order of the grouping attributes. The reason for this should be obvious from the Cobol examples: The easiest way to deal with groups is to sort the data. However, SQL does not *guarantee* to sort the results in this way — unless there is also an '**order by**' clause.

5.4 Selecting Aggregates

The **having** clause is easy to confuse with the **where** clause. The difference is that **having** is only applicable to queries with aggregates, and selects which aggregate *results* are displayed, *after they are calculated*. The **where** clause controls which *rows* of the table go into the calculation of the aggregates, *before they are calculated*.

Contrast the following two queries. In the first 'ALTCIU' is *not* listed, because no *single* delivery exceeds 10 units. In the second, it *is* listed, because the *total* of its deliveries does exceed 10 units.

```
select Item_No, sum(Qty_Delivered)  Total_Qty,
             sum(Cost) Total_Cost,
             sum(Cost)/sum(Qty) Unit_Cost
    from Updates
    where Qty_Delivered > 10
    group by Item_No;
Item # TotalQty   Total Cost  Unit Cost
------ -------- ------------ ----------
ALTPCD       30      $900.00     $30.00
PLPDCD       50   $14,950.00    $299.00
PLPRCM       50    $4,550.00     $91.00
select Item_No, sum(Qty_Delivered)  Total_Qty,
             sum(Cost) Total_Cost,
             sum(Cost)/sum(Qty) Unit_Cost
    from Updates
    group by Item_No
    having sum(Qty_Delivered) > 10;
Item # TotalQty   Total Cost  Unit Cost
------ -------- ------------ ----------
ALTCIU       40      $390.00      $9.75
ALTPCD       30      $900.00     $30.00
PLPDCD       50   $14,950.00    $299.00
PLPRCM       50    $4,550.00     $91.00
```

If an aggregate function is used and grouping is specified, one value of the aggregate statistic is calculated *for each group of rows* that have the same value of the columns in the **group by** clause. Omitting **group by** means that all the rows are considered as a *single* group. The number of rows in the result equals the number of groups, and will usually be less than the number of rows in the table.

*A common mistake is to use columns that appear neither as aggregates nor in the '**group by**' clause:*

```
select Item_No, Account, sum(Qty_Delivered)  Total_Qty,
             sum(Cost) Total_Cost,
             sum(Cost)/sum(Qty) Unit_Cost
    from Updates
    group by Item_No;
```

It does not make sense to try to display 'Account' because there is no guarantee that it will have the same value for every update that shares a given value of 'Item_No'. Indeed, the two deliveries for 'ALTCIU' in the 'Updates' table do have different values of 'Account'. On the other hand, the rule still applies even when there *is* such a guarantee.

There is an important *syntactical* rule concerning '**group by**':

*The only variables that can appear following '**select**' are either those that appear in the '**group by**' clause, or those that appear as aggregates.*

A second common mistake is to use aggregates in **where** conditions:

```
select Item_No, sum(Qty_Delivered)  Total_Qty,
                sum(Cost) Total_Cost,
                sum(Cost)/sum(Qty) Unit_Cost
    from Updates
    where Qty_Delivered > avg(Qty_Delivered)
    group by Item_No;
```

Here, '`avg(Qty_Delivered)`' can only be calculated *after* all the rows for a given 'Item_No' have been examined, but it is used to select them before calculating the average! (This kind of query can be answered using a nested sub-query. (*See Section 8.*)

6 Set Operators

Rows returned by **select** statements may be treated as sets, and combined using the operators **union, intersect**, and **minus**.!

Set operators can only be used if the sets of rows are **compatible**. To be compatible, they must contain the same number of columns, and each pair of columns must have the same description, ie., the same type, length, number of decimal places and the same **null** or **not null** property.

The following query displays all values of 'Account' that are found in both the 'Suppliers' and 'Customers' tables, as in the program in Section 8.1of *File Algorithms in Cobol!85*.

```
select Account from Suppliers
intersect
select Account from Customers;
Acct
----
B007
```

The following query displays the accounts of suppliers that are in the 'Suppliers' table, but whose 'Account' does not appear in the 'Products' table.

```
select Account from Suppliers
minus
select Supplier from Products;
Acct
----
B007
```

When a set operation returns several rows, they are typically in sorted order. This is because sort-merge is usually the best way to evaluate set operations:

```
select Name from Suppliers
union
select Name from Customers;
Business Name
------------------------------
Afrotechnics
Autobarn Elizabeth
BCR Mobile Installations
Blaupunkt
Blue Waters Nominees
Bobs Electronic Repairs
Car Audio Designs
Car Audio Services
Cargear Pty Ltd
Cartronics
Citisound
Complete Audio
Custom Audio Sound
Doug Sunstroms Sound Mart
Electric Bug Pty Ltd
Fujitsu Ten (Australia) P/L
Global Car Audio
JayCar Pty Ltd
National Car Audio
Netherlands Electrical Supply
Nippon Electronics Importers
Northern Car Radio
Pacific Auto Electronics
Pioneer Car Audio Services
RS Automotive Development
Sound 4 Australia Pty Ltd
South Aust. Import Trading
Southern Car Audio
Strathfield Car Radios
Tonkins Car Audio Pty Ltd
US Audio Imports
```

A set operation may involve more than one column, provided corresponding columns are compatible. The next query displays information about every delivery where the supplier is *not* the recommended supplier in the 'Products' table, specifically, any combination of 'Item_No' and 'Account' in the 'Updates' table that does not match a combination of 'Item_No' and 'Supplier' in the 'Products' table.

```
select Item_No, Account Supplier from Updates
minus
select Item_No, Supplier from Products;
Item # Acct
------ ----
ALTCIU N001
       N002
```

(The blank 'Item_No' is a silent repetition of 'ALTCIU'.)

A query may contain more than one set operator, but the order of evaluation must always be given explicitly using parentheses. Parentheses are needed even when the result does not depend on the order of evaluation, eg., '*a* **union** *b* **union** *c*' is invalid; it is necessary to specify '(*a* **union** *b*) **union** *c*' or '*a*!**union** (*b*!**union** *c*)'.

The following query displays each row of Suppliers whose 'Account' is in the set {B007, S002, U003}. This is a poor way of asking the query in Section 4.4.5 using the '**in**' operator.

```
select * from Suppliers where Account  = 'B007'
union (select * from Suppliers where Account  = 'S002'
       union select * from Suppliers where Account  = 'U003')
order by Account;
Acct Business Name               Number, Street      Suburb, State          Balance
---- --------------------------- ------------------- ----------------- -----------
B007 Blue Waters Nominees        1 Francis Road      Paramatta NSW 2150 $120,459.56
S002 South Aust. Import Trading 137 Burbridge Road  Hilton SA 5033           $0.00
U003 US Audio Imports            5 Penna Ave         Clayton  VIC 3109      -$45.50
```

Note that an **order by** clause applies to a whole query, not to individual **select** statements.

7 Joins

7.1 Cartesian Product

A **Cartesian product** is what is obtained by taking *every* pair of rows from two tables. The number of rows in the result is the product of the numbers of rows in the original tables. There are 45 rows in the 'Products' table, and 6 rows in the 'Suppliers' table, so there are 270 rows in their Cartesian product (denoted by 'Products × Suppliers').

The following query displays the Cartesian product of the 'Products' and 'Suppliers' tables.

```
select * from Products, Suppliers;
```

Unfortunately, the full result takes too much space to print here. The following projection of the product shows the general idea,

```
select Item_No, Description, Account Acct, Name
    from Products, Suppliers;
Item # Description                              Acct Business Name
------ ---------------------------------------- ---- ------------------------------
ACLOTP Alcatel One Touch Phone                  B007 Blue Waters Nominees
ALTCIU Altec Caller ID Unit                     B007 Blue Waters Nominees
ALTPCD Altec Portable CD Including Headphones   B007 Blue Waters Nominees
AUBCRC Audioboss Deluxe 60W Car Radio Cassette  B007 Blue Waters Nominees
AUGBOX Audio-Gods Box Speakers                  B007 Blue Waters Nominees
CANCBJ Canon Colour Bubble Jet Printer          B007 Blue Waters Nominees
CANFBS Canon Flatbed Scanner                    B007 Blue Waters Nominees
FAXROL Fax Rolls                                B007 Blue Waters Nominees
... and so on ...
UDNCID Uniden Cordless Phone With Caller ID     B007 Blue Waters Nominees
VCAPLE Voca Phone Line Extension Cord           B007 Blue Waters Nominees
ACLOTP Alcatel One Touch Phone                  N001 Netherlands Electrical Supply
ALTCIU Altec Caller ID Unit                     N001 Netherlands Electrical Supply
ALTPCD Altec Portable CD Including Headphones   N001 Netherlands Electrical Supply
AUBCRC Audioboss Deluxe 60W Car Radio Cassette  N001 Netherlands Electrical Supply
AUGBOX Audio-Gods Box Speakers                  N001 Netherlands Electrical Supply
CANCBJ Canon Colour Bubble Jet Printer          N001 Netherlands Electrical Supply
CANFBS Canon Flatbed Scanner                    N001 Netherlands Electrical Supply
FAXROL Fax Rolls                                N001 Netherlands Electrical Supply
... and so on ...
SYOPCP Sanyo Portable Cassette Player           U003 US Audio Imports
TVLCDC Travel CD Case                           U003 US Audio Imports
UDNCID Uniden Cordless Phone With Caller ID     U003 US Audio Imports
VCAPLE Voca Phone Line Extension Cord           U003 US Audio Imports
```

The definition of cartesian product may be extended to cases of three or more tables. The product operation is associative, so that $(a \times b) \times c = a \times (b \times c) = a \times b \times c$.[9]

7.2 Types of Join

A full Cartesian product is rarely useful. Any subset of a Cartesian product is called a **join**. The most often used form of join is an **equi-join**, which requires that some columns from the two tables should have matching values. If the columns have the same names, the equi-join is called a **natural join**.

The following query is an equi-join that links each row of the 'Products' table to its related row in the 'Suppliers' table.

```
select Description, Name from Products, Suppliers where Account = Supplier;
Description                              Business Name
---------------------------------------- ------------------------------
Alcatel One Touch Phone                  Pacific Auto Electronics
Altec Caller ID Unit                     Pacific Auto Electronics
Altec Portable CD Including Headphones   Pacific Auto Electronics
Audioboss Deluxe 60W Car Radio Cassette  US Audio Imports
Audio-Gods Box Speakers                  US Audio Imports
Canon Colour Bubble Jet Printer          Nippon Electronics Importers
... etc., until ...
Uniden Cordless Phone With Caller ID     Pacific Auto Electronics
Voca Phone Line Extension Cord           South Aust. Import Trading
```

9 A schema defines a set of columns. Each column has a range of possible values. The database instance is a subset of the Cartesian product of all possible attribute values. In mathematics, a subset of a Cartesian product is called a *relation*. Therefore a table expresses a relation between its columns. Hence the term 'relational database'.

Although it is 'obvious' that the values of 'Account' and 'Supplier' in the two tables should match, SQL does *not* assume it automatically, *even though the foreign key-primary key link was declared in the schema*. Without the **where** clause, the Cartesian product, with 270 rows, would have been listed, giving every possible combination of 'Description' and 'Name'. Note that columns on which the join is made do not need to be displayed.

The following query displays the name of any supplier and the name of any customer who share the same value of 'Account', and their common 'Account'. It has a similar effect to an earlier example using **intersect**, but it displays more information.

```
select Suppliers.Account Acct, Suppliers.Name, Customers.Name
     from Suppliers, Customers
     where Suppliers.Account = Customers.Account;
Acct Business Name                  Business Name
---- ------------------------------ ------------------------------
B007 Blue Waters Nominees           Blaupunkt
```

Where the same column name appears in both tables, it is necessary to **qualify** it with the table name, as in 'Suppliers.Account' and 'Customers.Account'. As in Cobol, it is only necessary to qualify names that are ambiguous. However, although it doesn't matter which table supplies the value of 'Account', it remains necessary to choose one or the other.

To save typing, it is possible to **alias** the *table* names. The next query is exactly equivalent to the previous one.

```
select S.Account Acct, S.Name, C.Name
     from Suppliers S, Customers C
     where S.Account = C.Account;
Acct Business Name                  Business Name
---- ------------------------------ ------------------------------
B007 Blue Waters Nominees           Blaupunkt
```

A join can involve several tables. The following query links each row of the 'Updates' table to its associated 'Products' and 'Suppliers' rows.

```
select YY_MM_DD, HH_MM_SS, Qty_Delivered Qty, Cost, Description, Name
         from Updates, Products, Suppliers
         where Updates.Item_No = Products.Item_No
         and Updates.Account = Suppliers.Account;
YY_MM_DD HH_MM_SS Delvrd       Cost Description                              Business Name
-------- -------- ------ ---------- ---------------------------------------- ----------------------------
01/02/06 14:55:14     10    $100.00 Altec Caller ID Unit                     Pacific Auto Electronics
01/02/06 14:57:19     30    $900.00 Altec Portable CD Including Headphones   Pacific Auto Electronics
01/02/06 14:58:44     50 $14,950.00 Philips Dtchble Radio Cass. w. CD Chngr Netherlands Electrical Supply
01/02/06 15:00:02     50  $4,550.00 Philips Radio Cass. w. Matching Spkrs    Netherlands Electrical Supply
01/02/06 15:01:40     10     $90.00 Altec Caller ID Unit                     Netherlands Electrical Supply
01/02/14 15:54:15     10    $100.00 Altec Caller ID Unit                     Pacific Auto Electronics
01/02/14 15:55:50     10    $100.00 Altec Caller ID Unit                     Pacific Auto Electronics
```

Note that only deliveries appear in the join; payments have no 'Item_No' so cannot be joined to the 'Product' table.

The above joins are all equi-joins. Most useful joins are.[10]

A **self-join** involves the same table twice, which makes *aliasing* essential. The following join finds all pairs of customers who share the same suburb.

```
select First.Name, Second.Name, First.Suburb Suburb
    from Customers First, Customers Second
    where First.Suburb = Second.Suburb;
```

However, the results are not what one would hope for:

10 Although the term 'natural join' strictly means an equi-join between columns with the same name, it is also loosely used to mean any join involving a foreign key and its associated primary key.

```
Business Name                  Business Name                  Suburb, State
------------------------------ ------------------------------ ------------------------
Citisound                      Citisound                      Adelaide  SA 5000
Global Car Audio               Citisound                      Adelaide  SA 5000
Strathfield Car Radios         Citisound                      Adelaide  SA 5000
JayCar Pty Ltd                 Citisound                      Adelaide  SA 5000
Tonkins Car Audio Pty Ltd      Citisound                      Adelaide  SA 5000
Citisound                      Global Car Audio               Adelaide  SA 5000
Global Car Audio               Global Car Audio               Adelaide  SA 5000
Strathfield Car Radios         Global Car Audio               Adelaide  SA 5000
JayCar Pty Ltd                 Global Car Audio               Adelaide  SA 5000
Tonkins Car Audio Pty Ltd      Global Car Audio               Adelaide  SA 5000
Citisound                      Strathfield Car Radios         Adelaide  SA 5000
Global Car Audio               Strathfield Car Radios         Adelaide  SA 5000
Strathfield Car Radios         Strathfield Car Radios         Adelaide  SA 5000
JayCar Pty Ltd                 Strathfield Car Radios         Adelaide  SA 5000
Tonkins Car Audio Pty Ltd      Strathfield Car Radios         Adelaide  SA 5000
Citisound                      JayCar Pty Ltd                 Adelaide  SA 5000
... and so on ...
Pioneer Car Audio Services     Pioneer Car Audio Services     Sefton Park  SA 5014
Car Audio Services             Car Audio Services             Sefton Park, SA 5014
Sound 4 Australia Pty Ltd      Sound 4 Australia Pty Ltd      St Marys  SA 5038
Cargear Pty Ltd                Cargear Pty Ltd                St Morris  SA 5059
Complete Audio                 Complete Audio                 Tea Tree Gully  SA 5157
RS Automotive Development      RS Automotive Development      Valley View  SA 5056
Cartronics                     Cartronics                     Wayville  SA 5014
```

Every name is at least paired with itself. We should insist that each row should have a different value of 'Account':

```
select First.Name, Second.Name, First.Suburb Suburb
    from Customers First, Customers Second
    where First.Suburb = Second.Suburb
    and First.Account <> Second.Account;
Business Name                  Business Name                  Suburb, State
------------------------------ ------------------------------ ------------------------
Global Car Audio               Citisound                      Adelaide  SA 5000
Strathfield Car Radios         Citisound                      Adelaide  SA 5000
JayCar Pty Ltd                 Citisound                      Adelaide  SA 5000
Tonkins Car Audio Pty Ltd      Citisound                      Adelaide  SA 5000
Citisound                      Global Car Audio               Adelaide  SA 5000
Strathfield Car Radios         Global Car Audio               Adelaide  SA 5000
JayCar Pty Ltd                 Global Car Audio               Adelaide  SA 5000
Tonkins Car Audio Pty Ltd      Global Car Audio               Adelaide  SA 5000
Citisound                      Strathfield Car Radios         Adelaide  SA 5000
Global Car Audio               Strathfield Car Radios         Adelaide  SA 5000
JayCar Pty Ltd                 Strathfield Car Radios         Adelaide  SA 5000
Tonkins Car Audio Pty Ltd      Strathfield Car Radios         Adelaide  SA 5000
Citisound                      JayCar Pty Ltd                 Adelaide  SA 5000
Global Car Audio               JayCar Pty Ltd                 Adelaide  SA 5000
Strathfield Car Radios         JayCar Pty Ltd                 Adelaide  SA 5000
Tonkins Car Audio Pty Ltd      JayCar Pty Ltd                 Adelaide  SA 5000
Citisound                      Tonkins Car Audio Pty Ltd      Adelaide  SA 5000
Global Car Audio               Tonkins Car Audio Pty Ltd      Adelaide  SA 5000
Strathfield Car Radios         Tonkins Car Audio Pty Ltd      Adelaide  SA 5000
JayCar Pty Ltd                 Tonkins Car Audio Pty Ltd      Adelaide  SA 5000
Southern Car Audio             National Car Audio             Lonsdale  SA 5157
National Car Audio             Southern Car Audio             Lonsdale  SA 5157
```

Notice how the pairs are grouped in ascending order of 'Suburb'. Sorting on 'Suburb' makes it easier and faster to match the corresponding pairs of rows.

This still may not be what is desired; for every (A,B) pair, there is a corresponding (B,A) pair. The following version gets rid of the unwanted pairs.

```
select First.Name, Second.Name, First.Suburb Suburb
    from Customers First, Customers Second
    where First.Suburb = Second.Suburb
    and First.Account < Second.Account;
Business Name                  Business Name                  Suburb, State
------------------------------ ------------------------------ ------------------------
Citisound                      Global Car Audio               Adelaide  SA 5000
Citisound                      JayCar Pty Ltd                 Adelaide  SA 5000
Global Car Audio               JayCar Pty Ltd                 Adelaide  SA 5000
National Car Audio             Southern Car Audio             Lonsdale  SA 5157
Citisound                      Strathfield Car Radios         Adelaide  SA 5000
Global Car Audio               Strathfield Car Radios         Adelaide  SA 5000
JayCar Pty Ltd                 Strathfield Car Radios         Adelaide  SA 5000
Citisound                      Tonkins Car Audio Pty Ltd      Adelaide  SA 5000
Global Car Audio               Tonkins Car Audio Pty Ltd      Adelaide  SA 5000
JayCar Pty Ltd                 Tonkins Car Audio Pty Ltd      Adelaide  SA 5000
Strathfield Car Radios         Tonkins Car Audio Pty Ltd      Adelaide  SA 5000
```

Although equi-joins are used most often, other joins are possible. A join using a comparison other than '=' is called a **θ-join** (theta-join). The following example displays cases where the 'Price' of a product exceeds the 'Credit_Limit' of a Customer.

```
select Description, Price, Name, Credit_Limit
    from Products, Customers
    where Price > Credit_Limit;
Description                                   Price Business Name                  Cr. Limit
---------------------------------------- ---------- ------------------------------ ---------
Sony 500W CD Tuner+6x9" Spkrs+4 Ch. Amp.  $1,029.50 Bobs Electronic Repairs           $1,000
Sony 500W CD Tuner+6x9" Spkrs+4 Ch. Amp.  $1,029.50 Car Audio Designs                 $1,000
Sony 500W CD Tuner+6x9" Spkrs+4 Ch. Amp.  $1,029.50 Cargear Pty Ltd                   $1,000
Sony 500W CD Tuner+6x9" Spkrs+4 Ch. Amp.  $1,029.50 Cartronics                        $1,000
Sony 500W CD Tuner+6x9" Spkrs+4 Ch. Amp.  $1,029.50 Complete Audio                    $1,000
Sony 500W CD Tuner+6x9" Spkrs+4 Ch. Amp.  $1,029.50 Custom Audio Sound                $1,000
Sony 500W CD Tuner+6x9" Spkrs+4 Ch. Amp.  $1,029.50 Doug Sunstroms Sound Mart         $1,000
Sony 500W CD Tuner+6x9" Spkrs+4 Ch. Amp.  $1,029.50 Doug Sunstroms Sound Mart         $1,000
Alcatel One Touch Phone                      $69.50 Afrotechnics                          $0
Altec Caller ID Unit                         $12.00 Afrotechnics                          $0
Altec Portable CD Including Headphones       $37.00 Afrotechnics                          $0
... and so on ...
Travel CD Case                                $3.50 Afrotechnics                          $0
Uniden Cordless Phone With Caller ID        $150.50 Afrotechnics                          $0
Voca Phone Line Extension Cord                $2.00 Afrotechnics                          $0
Sony 500W CD Tuner+6x9" Spkrs+4 Ch. Amp.  $1,029.50 Global Car Audio                  $1,000
Sony 500W CD Tuner+6x9" Spkrs+4 Ch. Amp.  $1,029.50 National Car Audio                $1,000
Sony 500W CD Tuner+6x9" Spkrs+4 Ch. Amp.  $1,029.50 Northern Car Radio                $1,000
Sony 500W CD Tuner+6x9" Spkrs+4 Ch. Amp.  $1,029.50 Pioneer Car Audio Services        $1,000
Sony 500W CD Tuner+6x9" Spkrs+4 Ch. Amp.  $1,029.50 Strathfield Car Radios            $1,000
Sony 500W CD Tuner+6x9" Spkrs+4 Ch. Amp.  $1,029.50 Tonkins Car Audio Pty Ltd         $1,000
Sony 500W CD Tuner+6x9" Spkrs+4 Ch. Amp.  $1,029.50 Tonkins Car Audio Pty Ltd         $1,000
Sony 500W CD Tuner+6x9" Spkrs+4 Ch. Amp.  $1,029.50 Tonkins Car Audio Pty Ltd         $1,000
```

Often, there is no more efficient way of evaluating a θ-join than using the nested-loops algorithm. Here, sorting the 'Products' table on descending 'Price' and the 'Customers' table on ascending 'Credit_Limit' would presumably speed up the search for rows that satisfy the join condition.

7.3 Outer Joins

Earlier, we joined each row of the 'Updates' table with its associated 'Description' and 'Name', but no payments were displayed, because they had no 'Item_No'. If we want *every* row of the 'Updates' table to appear, we must use an **outer join**.

```
select YY_MM_DD, HH_MM_SS, Qty_Delivered Qty, Cost, Description, Name
         from Updates, Products, Suppliers
         where Updates.Item_No = Products.Item_No (+)
         and Updates.Account = Suppliers.Account;
YY_MM_DD HH_MM_SS Delvrd       Cost Description                              Business Name
-------- -------- ------ ---------- ---------------------------------------- -----------------------------
01/02/06 14:55:14     10    $100.00 Altec Caller ID Unit                     Pacific Auto Electronics
01/02/06 14:56:23                                                            Nippon Electronics Importers
01/02/06 14:57:19     30    $900.00 Altec Portable CD Including Headphones   Pacific Auto Electronics
01/02/06 14:58:44     50 $14,950.00 Philips Dtchble Radio Cass. w. CD Chngr  Netherlands Electrical Supply
01/02/06 15:00:02     50  $4,550.00 Philips Radio Cass. w. Matching Spkrs    Netherlands Electrical Supply
01/02/06 15:01:07                                                            Nippon Electronics Importers
01/02/06 15:01:40     10     $90.00 Altec Caller ID Unit                     Netherlands Electrical Supply
01/02/14 15:54:15     10    $100.00 Altec Caller ID Unit                     Pacific Auto Electronics
01/02/14 15:55:50     10    $100.00 Altec Caller ID Unit                     Pacific Auto Electronics
01/02/14 15:56:17                                                            Nippon Electronics Importers
```

The '`(+)`' appearing alongside '`Products.Item_No`' signifies that if no match can be found for it, a dummy row of the 'Products' table should be assumed to exist, *with null values for each attribute.*

8 Nested Queries

Queries that return a single column occupy a special place in SQL. They may return a single value:

```
select avg(Balance) from Suppliers;
AVG(BALANCE)
------------
   24444.685
```

Alternatively, a single column query may return a *set* of values:

```
select Account from Suppliers where Balance > 1000;
Acct
----
B007
N001
P004
```

8.1 Sub-Queries

Single-column queries may be used to supply values in **where** and **having** clauses, but *only* as a *right-hand* operand.

The following query lists all supplier rows whose 'Balance' is less than the average:

```
select * from Suppliers where Balance <
       (select avg(Balance) from Suppliers);
Acct Business Name                  Number, Street               Suburb, State                 Balance
---- ------------------------------ ---------------------------- ------------------------ ------------
N001 Netherlands Electrical Supply  124 Burbridge Road           Hilton SA 5033             $15,077.99
N002 Nippon Electronics Importers   19 Ashford Rd                Redfern NSW 2016              -$50.00
P004 Pacific Auto Electronics       PO Box 407                   Sydney NSW 2000            $11,226.06
S002 South Aust. Import Trading     137 Burbridge Road           Hilton SA 5033                  $0.00
U003 US Audio Imports               5 Penna Ave                  Clayton  VIC 3109             -$45.50
```

The nested query is evaluated first, yielding a single value of 24444.685. This value is then substituted in the outer query, making it effectively, '`select * from Suppliers where Balance < 24444.685`'. It is clear that the 'Suppliers' table must be scanned *twice*.

In the following query, we ask which products are normally purchased from suppliers that are owed more than $1,000.

```
select Item_No, Description, Supplier from Products
         where Supplier in
         (select Account from Suppliers where Balance > 1000);
Item # Description                              Acct
------ ---------------------------------------- ----
ACLOTP Alcatel One Touch Phone                  P004
ALTCIU Altec Caller ID Unit                     P004
ALTPCD Altec Portable CD Including Headphones   P004
FDM6SS Freedom 6.5" Stereo Speakers             P004
HOMPYS Home Party Speakers                      P004
KITRCA Sansui Flipface+Amp with 4"+6x9" Spkrs   P004
LXKCPR Lexmark Colour Printer                   P004
NKADTC Nokia Desktop Charger                    P004
NKAHFK Nokia Genuine In-Car Handsfree Kit       P004
OMNLRC Omni Long Reach Digital Cordless         P004
PLPDCD Philips Dtchble Radio Cass. w. CD Chngr  N001
PLPDCT Philips 120 Watt Detachable CD Tuner     N001
PLPRCM Philips Radio Cass. w. Matching Spkrs    N001
PLPSEP Philips Stereo Earphones                 N001
RDMDRC Roadmaster Detachable Face Radio Cass.   P004
SUI2WC Sansui 2-Way Cmpnnt Mid/Woofer/Tweeter   P004
SUIDRC Sansui Hi-Powered Detachable Radio Cass. P004
SYOCSS Sanyo Deluxe Car Stereo Speakers         P004
SYODRC Sanyo Detachable Face Radio Cassette     P004
SYOPCP Sanyo Portable Cassette Player           P004
UDNCID Uniden Cordless Phone With Caller ID     P004
```

The sub-query gives the set {B007, N002, P004}. The outer query is therefore equivalent to,

```
select Item_No, Description, Supplier from Products
    where Supplier in ('B007', 'N002', 'P004');
```

Again, two scans of the 'Products' table are needed to answer the query.

Sub-queries can be nested to any required depth. The following query asks, “Which deliveries were for products normally purchased from suppliers who are owed over $1,000?”

```
select YY_MM_DD, HH_MM_SS, Item_No, Qty_Delivered from Updates
     where Item_no in
     (select Item_No from Products
           where Supplier in
           (select Account from Suppliers where Balance > 1000));
YY_MM_DD HH_MM_SS Item # Delvrd
-------- -------- ------ ------
01/02/06 14:55:14 ALTCIU     10
01/02/06 14:57:19 ALTPCD     30
01/02/06 14:58:44 PLPDCD     50
01/02/06 15:00:02 PLPRCM     50
01/02/06 15:01:40 ALTCIU     10
01/02/14 15:54:15 ALTCIU     10
01/02/14 15:55:50 ALTCIU     10
```

8.2 All and Any or Some

SQL allows any comparison relation to be used with a set of values, by using the qualifier **all** or **any**. **Some** is just another way of spelling **any**.

The following query finds the customer with the greatest value of Balance.

```
select * from Customers where Balance >= all (select Balance from Customers);
Acct Business Name      Number, Street                 Suburb, State        Balance Cr. Limit Avail. Credit
---- ------------------ ------------------------------ ------------------ ---------- --------- -------------
B007 Blaupunkt          Cnr Centre and McNaughton Rds  Clayton  VIC 3109    $10,295.00   $25,000  $12,537.00
```

The use of the word **all** is counter-intuitive here, and **any** seems more natural. The rule is that **all** means that the relation must be true for *all* members of the set for the condition to succeed, but **any** only needs the relation to be true once. If **any** had been used instead of **all**, every row would have been listed. Note that ‘> **all**’ or ‘< **all**’ can only return empty results.

In general, if θ is a relation such as =, <, <>, etc., then ‘*x*!θ!**all**!(*a*,!*b*,!*c*, …)’ means ‘*x* θ *a* **and** *x*!θ!*b* **and** *x* θ *c* …’, whereas ‘*x* θ **any** (*a*,!*b*,!*c*, …)’ means ‘*x*!θ!*a* **or** *x* θ *b* **or** *x* θ *c* …’.

Another way to write the same query is as follows:

```
select * from Customers where Balance = (select max(Balance) from Customers);
Acct Business Name      Number, Street                 Suburb, State        Balance Cr. Limit Avail. Credit
---- ------------------ ------------------------------ ------------------ ---------- --------- -------------
B007 Blaupunkt          Cnr Centre and McNaughton Rds  Clayton  VIC 3109    $10,295.00   $25,000  $12,537.00
```

where it is also clear that the ‘Customers’ table must be scanned twice.

8.3 Correlated Sub-Queries

In the previous examples the nested sub-query could be evaluated once, and the value or set it returned could be used by the outer query. This need not be the case. The following query finds all suppliers who supply more than 10 different products:

```
select * from Suppliers
    where 10 < (select count(*) from Products
                where Suppliers.Account = Products.Supplier);
Acct Business Name                  Number, Street                 Suburb, State                  Balance
---- ------------------------------ ------------------------------ ------------------------- ------------
N002 Nippon Electronics Importers   19 Ashford Rd                  Redfern NSW 2016                -$50.00
P004 Pacific Auto Electronics       PO Box 407                     Sydney NSW 2000              $11,226.06
```

In the case of a **correlated** sub-query, it is impossible for the inner query to be evaluated only once; the count of the ‘Products’ rows depends on which supplier row is being considered. Thus it is likely that the inner query will need to be evaluated 6 times, once for each row of the ‘Suppliers’ table. Correlated sub-queries can therefore take a long time to evaluate.

There are two things to notice about the syntax of this example: First, it is necessary to write ‘`10 < (select …)`’, rather than ‘`(select …) > 10`’. This is because a nested sub-query may only appear as a right-hand operand. Second, the sub-query refers to the table used in the outer query. This is the *signature* of a correlated sub-query.

Because correlated sub-queries can make heavy demands on a database server, they are best avoided where possible, even at the cost of making the query less direct. The above query can be rephrased as follows,

```
select * from Suppliers
    where Account in (select Supplier from Products
            group by Supplier having count(*) > 10);
Acct Business Name                  Number, Street              Suburb, State                 Balance
---- ------------------------------ --------------------------- ------------------------ ------------
N002 Nippon Electronics Importers   19 Ashford Rd               Redfern NSW 2016              -$50.00
P004 Pacific Auto Electronics       PO Box 407                  Sydney NSW 2000            $11,226.06
```

8.4 Exists

The **exists** condition is true if and only if the result of a sub-query contains at least one row. The following query lists the 'Item_No' and 'Description' of all products that have at least one delivery in the Updates table.

```
select Item_No, Description from Products
    where exists (select * from Updates
                  where Updates.Item_No = Products.Item_No);
Item # Description
------ ----------------------------------------
ALTCIU Altec Caller ID Unit
ALTPCD Altec Portable CD Including Headphones
PLPDCD Philips Dtchble Radio Cass. w. CD Chngr
PLPRCM Philips Radio Cass. w. Matching Spkrs
```

This is merely syntactic sugar for the query,

```
select Item_No, Description from Products
    where 0 < (select count(*) from Updates
               where Updates.Item_No = Products.Item_No);
Item # Description
------ ----------------------------------------
ALTCIU Altec Caller ID Unit
ALTPCD Altec Portable CD Including Headphones
PLPDCD Philips Dtchble Radio Cass. w. CD Chngr
PLPRCM Philips Radio Cass. w. Matching Spkrs
```

The **not exists** condition is the converse of **exists**. The following query displays the accounts and names of suppliers that are in the 'Suppliers' table, but whose 'Account' does not appear in the 'Products' table.

```
select Account, Name from Suppliers
    where not exists (select * from Products
                      where Products.Supplier = Suppliers.Account);
Acct Business Name
---- ------------------------------
B007 Blue Waters Nominees
```

These are all correlated sub-queries.

8.5 A Logic Trap

Suppose we want to find which products have *not* been delivered by supplier 'N001'. We might erroneously write the following query:

```
select Item_No, Account from Updates where Supplier <> 'N001';
Item # Acct
------ ----
ALTCIU P004
       N002
ALTPCD P004
       N002
ALTCIU P004
ALTCIU P004
       N002
```

The error is revealed by the following query:

```
select Item_No, Account from Updates where Supplier = 'N001';
Item # Acct
------ ----
PLPDCD N001
PLPRCM N001
ALTCIU N001
```

where we see that 'ALTCIU' *has* been delivered by 'N001' after all. A correct way to phrase the question is as follows:

```
select Item_No, Account from Updates
    where Item_No not in (select Item_No from Updates where Account = 'N001');
Item # Acct
------ ----
ALTPCD P004
```

The preceding examples illustrate a common form of logic trap. The fact that a product was delivered by a supplier other than 'N001' does *not* mean that it wasn't delivered by 'N001' as well.

In predicate calculus, if 'P(x)' is some predicate on 'x', remember that the complement of:

$\exists x\ (P(x))$

('There exists an x such that P(x) is true'), is not:

$\exists x\ (\textbf{not}\ P(x))$,

('There exists an x such that P(x) is false'), but is:

$\forall x\ (\textbf{not}\ P(x))$,

('For all x, P(x) is false').

In short, be cautious of any query that involves negation. Make sure you negate the right thing!

9 Execution of Select Statements

How a **select** statement is evaluated depends on the DBMS. Most systems have a **query optimiser** that chooses an efficient method, based on the sizes of the tables, the existence of indices, and so on. But from the user's point of view, *any* method of evaluation must give the same results as the following algorithm:

1. Form all possible combinations of rows of the tables in the **from** clause, ie., their Cartesian product.
2. If there is a **where** clause, select only those combinations that satisfy the **where** condition. (This involves evaluating any sub-queries appearing in the **where** condition.)
3. If **group by** is specified, sort the rows into groups, otherwise form a single group.
4. If an aggregate (eg., **sum**) is specified, reduce each group to a single result row.
5. If there is a **having** clause, select only those result rows that satisfy the **having** condition. (This involves evaluating any sub-queries appearing in the **having** condition.)
6. Project onto the columns in the **select** clause.
7. If **distinct** is used, discard duplicate results.
8. If a set operator such as **union** is used, combine the results of each **select** using the operator, discarding any duplicate rows.
9. If an **order by** clause is specified, sort the result rows into order.

With one exception, these steps are executed in the same order as their corresponding clauses appear in the **select** statement. The exception is that the projected columns are stated first, but projection does not occur until step 6. This is good human-factors design: The first thing a user is likely to be sure of is what columns are required in the result; the next questions are what tables they come from, how they should be joined, and so on.

There are two obvious ways in which query optimisation can improve on this model:

First, since a Cartesian product is usually very large, rather than generate all terms in the product, then reject those that do satisfy the **where** clause, they may never be generated at all.

For example, in forming the natural join on 'Account' between the 'Updates' table and the 'Suppliers' table, where there is a many-one relation, it is best to take each row of the 'Updates' table in turn and match it with the corresponding 'Products' row directly, either using the 'Account' index on the primary key of the 'Products' table, or by sorting and merging. *Any* query optimiser should be expected to take advantage of foreign key–primary key joins. In terms of implementation, this means using the random access or sort-merge join algorithm instead of the nested loops algorithm.

Second, unless a nested sub-query is correlated, a query optimiser will arrange to evaluate it only once, not for each row of the product. It is easy to detect whether a sub-query is correlated; a correlated sub-query refers to a column of a table in the enclosing query; an independent sub-query does *not*.

10 Views

A **view** is a named *persistent* **select** statement — a bit like a library procedure. For example, the following view creates a version of the 'Customers' table for customers that have an non-zero value of goods on back order, similar to the first query in Section 4.4, and is equivalent to the Cobol program of Section 6.1 of *File Processing in Cobol 85*.

```
create view Back_Order_Customers as
    (select * from Customers where Available-Credit <> Credit_Limit - Balance);
select * from Back_Order_Customers;
```

Acct	Business Name	Number, Street	Suburb, State	Balance	Cr. Limit	Avail. Credit
A001	Autobarn Elizabeth	61 Elizabeth Way	Elizabeth SA 5158	$208.50	$5,000	$4,792.00
B007	Blaupunkt	Cnr Centre and McNaughton Rds	Clayton VIC 3109	$10,295.00	$25,000	$12,537.00
B012	Bobs Electronic Repairs	28 Limbert Avenue	Seacombe Gardens SA 5047	–$129.50	$1,000	$1,130.00
C007	Cargear Pty Ltd	453 Magill Road	St Morris SA 5059	$0.00	$1,000	$865.00
C020	Citisound	141–145 Franklin Street	Adelaide SA 5000	$35.00	$5,000	$4,784.00
C031	Custom Audio Sound	Unit 2 798 Marion Road	Marion SA 5045	$0.00	$1,000	$638.00
D015	Doug Sunstroms Sound Mart	6 Smart Road	Modbury SA 5143	–$30.50	$1,000	$1,031.00
E003	Electric Bug Pty Ltd	199–203 Torrens Road	Croydon SA 5013	$1,305.00	$5,000	$3,452.00
F002	Fujitsu Ten (Australia) P/L	75 Westgate Drive	Altona North VIC 3039	$1,182.00	$2,000	$1,818.00
N014	Northern Car Radio	1445 Main North Road	Para Hills West SA 5150	$0.00	$1,000	$843.00
R003	RS Automotive Development	19 Warburton Road	Valley View SA 5056	$4,720.85	$5,000	$279.00
S004	Sound 4 Australia Pty Ltd	22 Pinn Street	St Marys SA 5038	$2,758.75	$5,000	$2,241.00
S015	Strathfield Car Radios	316 Gouger Street	Adelaide SA 5000	$3.50	$1,000	$997.00
T003	Tonkins Car Audio Pty Ltd	116 Sherriffs Road	Morphett Vale SA 5153	$492.50	$1,000	$481.00

A view may be based on a query of arbitrary complexity. Not surprisingly, projection and selection can be combined in the same view. The following view is similar to the second query of Section 4.4, and the example of Section 6.3 of *File Processing in Cobol 85*.

```
create view Back_Orders as
    (select Account, Name, Credit_Limit - Available_Credit - Balance Back_Orders
         from Customers where Available-Credit <> Credit_Limit - Balance);
select * from Back_Orders;
Acct Business Name                    Back Orders
---- ------------------------------ ------------
A001 Autobarn Elizabeth                    -$0.50
B007 Blaupunkt                          $2,168.00
B012 Bobs Electronic Repairs               -$0.50
C007 Cargear Pty Ltd                      $135.00
C020 Citisound                            $181.00
C031 Custom Audio Sound                   $362.00
D015 Doug Sunstroms Sound Mart             -$0.50
E003 Electric Bug Pty Ltd                 $243.00
F002 Fujitsu Ten (Australia) P/L      -$1,000.00
N014 Northern Car Radio                   $157.00
R003 RS Automotive Development              $0.15
S004 Sound 4 Australia Pty Ltd              $0.25
S015 Strathfield Car Radios                -$0.50
T003 Tonkins Car Audio Pty Ltd             $26.50
```

SQL also allows a syntax that names the columns of the view, rather than using aliases:

```
create view Back_Orders (Account, Name, Back_Orders) as
    (select Account, Name, Credit_Limit - Available_Credit - Balance
          from Customers where Available-Credit <> Credit_Limit - Balance);
select * from Back_Orders;
Acct Business Name                  Back Orders
---- ----------------------------- ------------
A001 Autobarn Elizabeth                  -$0.50
B007 Blaupunkt                        $2,168.00
B012 Bobs Electronic Repairs             -$0.50
etc., ...
```

We can use a view to define a version of the ‘Updates’ table that contains only deliveries:

```
create view Deliveries as
    (select YY_MM_DD, HH_MM_SS, Item_No, Account, Qty_Delivered, Cost
        from Updates where Kind = 'D');
select * from Deliveries;
YY_MM_DD HH_MM_SS Item # Acct Delvrd         Cost
-------- -------- ------ ---- ------ ------------
01/02/06 14:55:14 ALTCIU P004     10      $100.00
01/02/06 14:57:19 ALTPCD P004     30      $900.00
01/02/06 14:58:44 PLPDCD N001     50   $14,950.00
01/02/06 15:00:02 PLPRCM N001     50    $4,550.00
01/02/06 15:01:40 ALTCIU N001     10       $90.00
01/02/14 15:54:15 ALTCIU P004     10      $100.00
01/02/14 15:55:50 ALTCIU P004     10      $100.00
```

A view may usually be treated just like a table; eg., displayed by **select** statements.

Views serve four purposes:

They enable complex queries to be factored into simpler queries.

They provide a means of viewing data from a different perspective, (eg., ‘Updates’ may be grouped by ‘Item_No’ or by ‘Account’).

They may be used to limit another user’s access rights to a table.

They can enforce database integrity. (*See* Section 11.5).

10.1 Views and Tables

A similar syntax may also be used to create a new *table*, rather than a view:

```
create table Deliveries_Snapshot as
    (select YY_MM_DD, HH_MM_SS, Item_No, Account, Qty_Delivered, Cost
        from Updates where Kind = 'D');
select * from Deliveries_Snapshot;
YY_MM_DD HH_MM_SS Item # Acct Delvrd         Cost
-------- -------- ------ ---- ------ ------------
01/02/06 14:55:14 ALTCIU P004     10      $100.00
01/02/06 14:57:19 ALTPCD P004     30      $900.00
01/02/06 14:58:44 PLPDCD N001     50   $14,950.00
01/02/06 15:00:02 PLPRCM N001     50    $4,550.00
01/02/06 15:01:40 ALTCIU N001     10       $90.00
01/02/14 15:54:15 ALTCIU P004     10      $100.00
01/02/14 15:55:50 ALTCIU P004     10      $100.00
```

What is the difference between the table and the view? Suppose that, after ‘Deliveries’ and ‘Deliveries_Snapshot’ are created, a new row is inserted into the ‘Updates’ table, recording a delivery of product ‘SST2CA’ from supplier ‘U003’, say. Then ‘Deliveries’ will include the new delivery, but ‘Deliveries_Snapshot’ will not. This explained as follows: A table contains data, but a view is a *procedure*. The table ‘Deliveries_Snapshot’ is a *copy* of some of the rows from the ‘Updates’ table, made at the time of the **create** statement. The copy doesn’t change when the new row is added to the original table, but the view ‘Deliveries’ is a stored query that is executed again whenever it is named. By substitution,

```
select * from Deliveries;
```

is equivalent to:

```
select YY_MM_DD, HH_MM_SS, Item_No, Account, Qty_Delivered, Cost
        from Updates where Kind = 'D';
```

It is possible to create a view (or a table) that joins several tables or existing *views*. In which case, it will be a good idea to give its columns new names, because the default names that result from the query will typically include table names as qualifiers.

```
create view Extended_Deliveries
    (YY_MM_DD, HH_MM_SS, Description, Name, Qty, Cost, Unit_Cost) as
    (select YY_MM_DD, HH_MM_SS, Description, Name, Qty_Delivered, Cost,
            Cost/Qty_Delivered)
        from Updates, Products, Suppliers
        where Updates.Item_No = Products.Item_No
        and Updates.Account = Suppliers.Account
        and Kind = 'D');
select * from Extended_Deliveries;
YY_MM_DD HH_MM_SS Description                          Business Name                 Delvrd        Cost  Unit Cost
-------- -------- ------------------------------------ ----------------------------- ------ ----------- ----------

01/02/06 14:55:14 Altec Caller ID Unit                 Pacific Auto Electronics          10     $100.00     $10.00
01/02/06 14:57:19 Altec Portable CD Including Headphones Pacific Auto Electronics        30     $900.00     $30.00
01/02/06 14:58:44 Philips Dtchble Radio Cass. w. CD Chngr Netherlands Electrical Supply  50  $14,950.00    $299.00
01/02/06 15:00:02 Philips Radio Cass. w. Matching Spkrs Netherlands Electrical Supply    50   $4,550.00     $91.00
01/02/06 15:01:40 Altec Caller ID Unit                 Netherlands Electrical Supply     10      $90.00      $9.00
01/02/14 15:54:15 Altec Caller ID Unit                 Pacific Auto Electronics          10     $100.00     $10.00
01/02/14 15:55:50 Altec Caller ID Unit                 Pacific Auto Electronics          10     $100.00     $10.00
```

10.2 Grouped Views

A view that finds an aggregate and includes a **group by** or a **having** clause, is called a **grouped view**. Some DBMS's place severe restrictions on the use of grouped views; others are more tolerant. Typically, a grouped view may not be joined with any other table, and a **select** statement that uses a grouped view is not allowed to have a **where** clause, **group by** clause or **having** clause.

As an example, consider a view of the 'Products' table that finds the total stock and valuation for each value of 'Supplier':

```
create view Supplier_Valuation  (Account, Stock, Valuation) as
    (select Supplier, sum(Stock), sum(Valuation) from Products group by Supplier);
select * from Supplier_Valuation;
Acct  Stock    Valuation
---- ------ ------------
N001  1,216    $5,725.86
N002    379   $23,869.10
P004    307   $11,256.52
S002  1,342    $6,672.53
U003     64    $5,012.98
```

The following **select** statement works with *Oracle*, but might fail with a different DBMS:

```
select * from Supplier_Valuation where Valuation > 10000;
Acct  Stock    Valuation
---- ------ ------------
N002    379   $23,869.10
P004    307   $11,256.52
```

The reason is that the query may be treated as equivalent to the following statement, which would be obtained by textual substitution,

```
select Supplier, sum(Stock), sum(Valuation) from Products
        where sum(Valuation) > 10000 group by Supplier
```

This statement would not be legal, because an aggregate such as **sum** can belong in a **having** clause, but not a **where** clause. On the other hand,

```
select * from Supplier_Valuation having Valuation > 10000;
```

is a syntax error because a **having** clause can only appear after a **group by** clause.

10.3 Dropping Views

Views are *persistent*; they remain defined until they are **altered**, or until they are **dropped**. The following statement would remove the 'Supplier_Valuation' view,

```
drop view Supplier_Valuation;
```

The effect of this is that the view can no longer be used, but its underlying table is *unchanged*.

11 Updating

Updating a table means keeping it up-to-date, so that it accurately models the real world. There are three kinds of SQL update statement, **insert**, **delete**, and **update**.![11] The first adds new rows, the second deletes existing rows, and the third modifies existing rows.

11.1 Insert

We have already seen examples of single row **insert** statements in Section 3. There is also a multi-row form of insert. This offers a second way to create the 'Deliveries_Snapshot' table. In this case, we introduce an additional column for 'Unit_Cost':

```
create table Deliveries_Snapshot (
    YY_MM_DD       char(8),
    HH_MM_SS       char(8),
    Item_No        char(6),
    Account        char(4),
    Qty            number(4),
    Cost           number(8,2),
    Unit_Cost      number(6,2),
    primary key    (YY_MM_DD, HH_MM_SS)
    );
insert into Deliveries_Snapshot
    (select YY_MM_DD, HH_MM_SS, Item_No, Account, Qty_Delivered, Cost,
        Cost/Qty_Delivered
        from Updates where Kind = 'D');
select * from Deliveries_Snapshot;
YY_MM_DD HH_MM_SS Item # Acct Delvrd         Cost  Unit Cost
-------- -------- ------ ---- ------ ------------ ----------
01/02/06 14:55:14 ALTCIU P004     10      $100.00     $10.00
01/02/06 14:57:19 ALTPCD P004     30      $900.00     $30.00
01/02/06 14:58:44 PLPDCD N001     50   $14,950.00    $299.00
01/02/06 15:00:02 PLPRCM N001     50    $4,550.00     $91.00
01/02/06 15:01:40 ALTCIU N001     10       $90.00      $9.00
01/02/14 15:54:15 ALTCIU P004     10      $100.00     $10.00
01/02/14 15:55:50 ALTCIU P004     10      $100.00     $10.00
```

This approach allows us to specify the size and precision of 'Unit_Cost'.

11.2 Delete

We can **delete** rows from a table. The row or rows to be deleted must be specified by a **where** condition. The following statement deletes all rows of the 'Products' that are neither in stock nor on order.

```
delete from Products where Stock = 0 and On_Order = 0;
```

The next example deletes all products normally supplied by 'Netherlands Electrical Supply'.

```
delete from Products where Item_No in
    (select Item_No from Suppliers
        where Name = 'Netherlands Electrical Supply');
```

11.3 Update

An **update** statement allows rows to be **set** to new values. We can update 'Products' as follows, increasing 'On_Order' by 'Reorder_Qty' if a product needs to be re-ordered.

```
update Products
    set On_Order = On_Order + Reorder_Qty
    where Stock + On_Order < ReOrder_Level;
```

The expressions that can be used to update columns are restricted; they may only be constants or values derived from columns of the table that is being updated. They may not be sub-queries, and may not use aggregates (such as **sum**, **count**, etc.). The effect of these rules is that it is impossible to use **update** to modify one table (eg., a master file) according to the contents of a second table (eg., a transaction file). There is no such problem with **insert** or **delete.** One solution is to **create** a new version of the table.

---

[11] Confusingly, **update** is one kind of update statement.

11.4 Updating Views

Views *may* be updated, but only in restricted circumstances. Consider again the view,

```
create view Supplier_Valuation  (Account, Stock, Valuation) as
    (select Supplier, sum(Stock), sum(Valuation) from Products group by Supplier);
```

Then consider the update:

```
insert into Supplier_Valuation values ('N001', 100, 3265.00);
```

This implies that 100 units have been received from supplier 'N001' at a cost of $3,265. But which product or products were involved? Since 'Supplier_Valuation' is merely a view of the underlying 'Products' table, this question needs to be answered.

The general theoretical question of which views could be updated is complex. *Oracle* SQL currently makes the following restrictions:

- Rows can only be deleted from single table views that do not include aggregate functions (eg., **sum**), or include **group by** or **distinct**.
- Rows can only be updated if the deletion conditions are satisfied, and none of the columns in the view are given by expressions.
- Rows can only be added if the updating conditions are satisfied, and the view includes all the **not null** columns of the base table.

The following view is *not* updateable,

```
create view Deliveries as
    (select YY_MM_DD, HH_MM_SS, Item_No, Account, Qty_Delivered, Cost
        from Updates where Kind = 'D');
```

because the 'Updates' table specifies 'Kind' as **not null**.

The following version *is* updateable.

```
create view Deliveries as
    (select YY_MM_DD, HH_MM_SS, Kind, Item_No, Account, Qty_Delivered, Cost
        from Updates where Kind = 'D');
```

Adding a row to this view would cause the 'Updates' table to contain a new row with a null 'Amount' — but this is exactly what is required in a 'Delivery':

```
insert into Deliveries
    values ('01/04/27', '12:30:23', 'D', 'AUGBOX', 'U003', 10, 225);
```

11.5 With Check Option

Surprisingly, it may be possible to add a row to a view so that the new row is not visible via the view. This happens if 'Kind' is chosen incorrectly:

```
insert into Deliveries
    values ('01/04/27', '12:30:23', '$', 'AUGBOX', 'U003', 10, 225);
select * from Deliveries;
YY_MM_DD HH_MM_SS Kind Item # Acct Delvrd         Cost  Unit Cost
-------- -------- ---- ------ ---- ------ ------------ ----------
01/02/06 14:55:14 D    ALTCIU P004     10      $100.00     $10.00
01/02/06 14:57:19 D    ALTPCD P004     30      $900.00     $30.00
01/02/06 14:58:44 D    PLPDCD N001     50   $14,950.00    $299.00
01/02/06 15:00:02 D    PLPRCM N001     50    $4,550.00     $91.00
01/02/06 15:01:40 D    ALTCIU N001     10       $90.00      $9.00
01/02/14 15:54:15 D    ALTCIU P004     10      $100.00     $10.00
01/02/14 15:55:50 D    ALTCIU P004     10      $100.00     $10.00
```

This is because a view is a procedure for *answering* a query, not for updating. But the view satisfies the restrictions on updateable views, so the update is valid.

The **with check option** clause means that whenever a row is added to a table via a view, it is first checked to see that it would be visible via the view. The following view will only allow insertions where `Kind = 'D'`.

```
create view Deliveries as
    select YY_MM_DD, HH_MM_SS, Kind, Item_No, Account, Qty_Delivered, Cost
        from Updates where Kind = 'D'
        with check option;
```

12 Database Integrity

Integrity is that property of a database that ensures that it represents a possible real world situation. For example, it would not do to record a delivery from supplier 'C003', because there is no such row in the 'Suppliers' table.

A database **constraint** is an arbitrary condition on the database *instance*. There are six classes of constraint:

- **type** constraints,
- **range** constraints,
- **row** constraints,
- **key** constraints,
- **referential** constraints, and
- **general** constraints.

Type constraints are enforced by column declarations; eg., it is not possible to assign a character string value to a numeric field.

Range constraints are an extension of type constraints. For example, it might be specified that the value of 'Qty_Delivered' must be in the range 1–2000.

A **row** constraint is a relationship between various attributes *from the same row* of a table.

A **key** constraint guarantees that no two rows of a table have the same key. We have already seen examples of key constraints in Sections 2.2 and 2.3.

A **referential** constraint is, for example, that the 'Supplier' attribute of a product must refer to the 'Account' of a row in the 'Suppliers' table. This kind of condition is enforced by a **foreign key** constraint. This feature is valuable; it is easy to violate referential integrity accidentally. For example, if a row is deleted from the 'Suppliers' table, several 'Products' rows may be left still referring to the now non-existent row.

Finally, a **general** constraint is any constraint that is not one of the above. For example, suppose the 'Suppliers' table contained an additional column called 'Total_Valuation'. The value in this column might always need to equal the sum of 'Valuation' over all 'Products' rows that have a matching value of 'Supplier'. Whenever the 'Products' table was updated, some values of 'Total_Valuation' might to need to be updated too.

12.1 Range and Row Constraints

If a constraint involves individual rows of a table, one way to enforce it is by creating a view having the **with check option**.

```
create view Deliveries as
    select YY_MM_DD, HH_MM_SS, Kind, Item_No, Account, Qty_Delivered, Cost
        from Updates
        where Kind = 'D'
        and Qty_Delivered between 1 and 2000
        with check option;
```

Another way to achieve the same result is by using a **check** clause when creating the table itself. A **check** clause can concern one column (in which case it is a range constraint), or a row as a whole. A constraint may be given a *name* by a **constraint** clause. Named constraints can be dropped later, perhaps being replaced by a different constraint. The following example checks that 'Kind' has a sensible value, and that the correct attributes are present for deliveries or payments. 'Qty_Delivered' is constrained to the range 1–2000, and 'Item_No' and 'Account' must refer to valid parent rows. Finally, 'YY_MM_DD' must signify a date in the year 2001.

```
create table Updates (
   YY_MM_DD       char(8),
   HH_MM_SS       char(8),
   Kind           char(1) not null check (Kind in ('D', '$')),
   Item_No        char(6),
   Account        char(4),
   Qty_Delivered  number(4) check (Qty_Delivered between 1 and 2000),
   Cost           number(8,2),
   Amount         number(8,2),
   check          ((Kind = 'D'
                      and Item_No is not null
                      and Qty_Delivered is not null
                      and Cost is not null)
                  or (Kind = '$'
                      and Amount is not null)),
   constraint     Year_is_2001
                    check (YY_MM_DD between '01/01/01' and '01/12/31'),
   primary key    (YY_MM_DD, HH_MM_SS),
   foreign key    (Item_No) references Products(Item_No),
   foreign key    (Account) references Suppliers(Account)
   );
```

It is possible to add or drop constraints using the **alter table** statement. Only constraints already named in a **constraint** clause can be dropped.

```
alter table Updates
    drop Year_is_2001
    add constraint Year_is_2002
        check (YY_MM_DD between '02/01/01' and '02/12/31');
```

12.2 Unique

It is possible to specify that a column or a combination of several columns should be **unique**. Since the only efficient way to check for uniqueness is to maintain an index, *Oracle* will create one automatically. The primary key of a table is automatically unique. This was already discussed in Section 2.3. Although the 'Name', 'Street' and 'Suburb' of a customer are not necessarily unique individually, their combination needs to be. The following example shows how to specify this constraint.

```
create table Customers (
    Account           char(4),
    Name              char(30) not null,
    Street            char(30) not null,
    Suburb            char(30) not null,
    Balance           number(8,2) not null,
    Credit_Limit      number(6) not null,
    Available-Credit  number(8,2) not null,
    primary key       (Account),
    unique            (Name, Street, Suburb)
    );
```

12.3 Referential Integrity

After a column or set of columns have been declared as a **primary key** in one table, they may be declared as a **foreign key** in a second table.[12] The first table is usually called the **parent**, and the second table called the **child** or **dependent**.! The following statement ensures that every 'Products' row refers to a 'Suppliers' row.

---

12 This means that a circular series of references is impossible.

```
create table Products (
    Item_No           char(6),
    Supplier          char(4) not null,
    Description       char(40) not null,
    Reorder_Level     number(4) not null,
    Reorder_Qty       number(4) not null,
    On_Order          number(4) not null,
    Stock             number(4) not null,
    Price             number(6,2) not null,
    Available-Credit  number(8,2) not null,
    primary key       (Item_no),
    foreign key       (Supplier) references Suppliers(Account)
                      on delete restrict
                      on update restrict
    );
```

Here, 'Suppliers' is the parent, and 'Products' is the dependent. For any given foreign key, there may be several dependents of a parent row, but only one parent row of a dependent.

Some have difficulty remembering whether the foreign key declaration is made in the parent table or the dependent table. It is made in the dependent table; if the parent table referred to the dependent table, it would need a *list* of keys, which is not possible.

Declaring 'Supplier' as a foreign key implicitly enforces an update rule: Whenever a new row is created or an existing row is modified in the dependent table ('Products'), the value of 'Supplier' must be the primary key of a existing row in the parent table ('Suppliers').

The **on delete restrict** clause is more subtle; it refers to deleting a row from the *parent* table, *not* the dependent. It means that if a row of the 'Products' table refers to a 'Suppliers' row, and the 'Suppliers' row is deleted, that is an error, because that would leave the 'Products' row with a meaningless foreign key. The **on update restrict** clause means that it would be a similar error to update the 'Account' column of a 'Suppliers' row while it was still referred to by any 'Products' row. **Restrict** is also the default action if no clause is specified.

There are two other options that can take the place of **restrict**: **set null** or **cascade**.

Set null means that instead of an error occurring when the primary key of the parent table is changed, the meaningless foreign keys in its dependent rows become **null**. (So at least they won't be left referring to a non-existent row.)

Cascade has different effects for **update** and **delete**. If a parent primary key is updated, the foreign keys of its dependent rows are changed too, so that they still refer to the same parent *row* (rather than the same key). But if a parent row is deleted, **cascade** means that that its dependent rows are deleted too. Since the primary keys of the dependent rows might also be foreign keys in a third table that also has the **cascade** option, the deletion can propagate recursively — hence the name 'cascade'. For example, if the 'Products' table and 'Updates' tables both specified the **cascade** option, deleting a supplier row would delete all products associated with the supplier, and all updates associated with those products.

12.4 Look-Up Tables

Suppose an attribute must have one of a restricted set of values, as in the case of the 'Kind' attribute of the 'Updates' table. If the set of values is likely to remain fixed for all time, they can be set up as a range constraint in *each* table that has a 'Kind' attribute. But if the set is likely to change, as when new kinds of updates are added to the system, it may be better to store the values in a single column table.

The next example creates a 'Valid_Kinds' table, to check that 'Updates' rows have correct values of 'Kind', specified as a foreign key.

```
create table Valid_Kinds (
    Kind char(1),
    primary key (Kind)
    );
insert into Valid_Kinds values ('D');
insert into Valid_Kinds values ('$');
create table Updates (
    YY_MM_DD          char(8),
    HH_MM_SS          char(8),
    Kind              char(1) not null,
    Item-No           char(6),
    Account           char(4),
    Qty_Delivered     number(4),
    Cost              number(8,2),
    Amount            number(8,2),
    primary key       (YY_MM_DD, HH_MM_SS),
    foreign key       (Item_No) references Products(Item_No),
    foreign key       (Account) references Suppliers(Account),
    foreign key       (Kind) references Valid_Kinds(Kind)
    );
```

12.5 Triggers

General constraints can often be maintained by *triggers*. Triggers are non-SQL procedures that can be invoked when rows are inserted or deleted, or columns are updated. For example a trigger could be defined to update the 'Balance' attribute in the 'Suppliers' table every time a new row is inserted in the 'Updates' table. Triggers are not part of the current ANS SQL standard, although they will probably become part of eventually.

13 Transactions

The other way to guarantee general constraints involving more than one table is by careful updating.! Suppose it is desired to maintain a new column in the 'Suppliers' table, called 'Total_Valuation', which is the sum of 'Valuation' in all 'Products' rows that match the supplier account. If the 'Valuation' of the 'Products' table is updated, it is necessary to update 'Total_Valuation' in the corresponding 'Suppliers' row. The following sequence of statements makes it possible.

```
update Products
    set Stock = Stock + 10, Valuation = Valuation + 225
    where Item_No = 'AUGBOX';
update Suppliers
    set Total_Valuation = Total_Valuation + 225
    where Account = 'U003';
```

Between these two actions the database is **inconsistent**. This will not matter unless it is inspected at a point in time between the two updates. Although this is a silly thing for one user to do, it might happen if the database is *shared* with other users. Suppose another user has decided to audit the values of 'Total_Valuation'.

```
select Account from Suppliers
    where Total_Valuation <>
    (select sum(Valuation) from Products
        where Products.Supplier = Suppliers.Account);
Account
-------
U003
```

Although it is a 'logical impossibility' that this query should ever return any rows, it *could* happen — but not reproducibly! The above output would only be possible if the query happens to inspect 'Total_Valuation' at a point in time *before* the update to 'Suppliers', but inspects the values of 'Valuation' *after* the update to 'Products'. To avoid such problems, SQL offers the **commit** command.

13.1 Commit and Rollback

Conceptually, no changes are made to a database by **insert** or **update** commands until a **commit** command is executed, then *all* the updates are made *atomically*. The updates are held

in a buffer until the **commit** is executed. While the **commit** takes place, no other user can access the updated records. A set of updates that is committed at one time is called a *transaction*.

Alternatively, SQL can *undo* updates by giving a **rollback** command, discarding the buffered updates. This would normally only be done after the DBMS has detected some kind of error that makes it impossible to complete the transaction.

Interactions between concurrent users of the same data are governed by the criterion of **serialisability**: The answers to queries and the final state of the database should correspond to some possible sequence of updates and queries made by a hypothetical single user. This sequence need not be unique; for example, if two users make two updates to two different suppliers, it does not matter in what order they are considered to be made.

13.2 Recovery

Recovery is the process of restoring the database to a consistent state after a hardware failure. This is done by keeping a *log* (or 'audit trail'), which records all changes made to the database. Typically, a log contains a series of pairs of *before-images* and *after-images*. A before-image is a copy of a row before it was updated; its after-image is a copy of the same row after the update. The series of pairs of images are punctuated by records that represent **commit** commands. These records mark points in time when the database was in a consistent state.

Two recovery strategies are possible: Starting from the current state of the database, it may be *rolled back* to an earlier consistent state by restoring updated rows from their before-images.[13] This usually takes little time. But if the database is unreadable, it may also be *recovered* from a previous *backup* copy, followed by restoring the updated rows from their after-images (*roll forward*).

14 Database Security

Database **security** concerns the confidentiality of data, and who may read or write it. Users may have rights to read only certain tables, or only certain views. Access rights are controlled using SQL's Data Control Language (DCL). All rights originate from an omnipotent user called **public** or **system**.

14.1 Grant

Brown can allow Smith and Jones to read her 'Suppliers' and 'Products' tables, as follows:

```
grant select on Suppliers, Products to Smith, Jones;
```

The issuer of the **grant** statement (Brown) is the **grantor**, the receivers of the privilege (Smith and Jones) are the **grantees**. Smith and Jones must refer to Brown's tables as 'Brown.Suppliers' and 'Brown.Products'.

Specifying **public** as the grantee makes tables available to all users of the database.

In *standard* SQL, the possible privileges are **select**, **insert**, **update**, **delete**, **alter**, **index** and **references**. (The word **all** may be used to stand for the whole set.) The first five privileges allow the grantee to use the corresponding SQL statements on the tables concerned. The **index** privilege allows the grantee to construct an index to the table. (There may seem no need to prevent this, but every index takes extra time to update whenever its associated table is updated.)

The **references** privilege is needed to support referential integrity. If a grantee is given **insert** privilege to the 'Products' table, for example, then the integrity of 'Supplier' cannot be checked without some form of access to the 'Suppliers' table. Granting **reference** privilege to the 'Products' table allows its foreign key to be checked, *without* giving the grantee the right to browse the parent table. (There are some complications here when cascaded updates are allowed.)

---

[13] This is typically a larger scale operation than is provided by the **rollback** *command*. During recovery, updates will be rolled back that have already been committed.

14.2 Revoke

The **revoke** statement withdraws privileges previously given by a **grant** statement. The following statement shows how Brown takes away all the privileges she gave to Jones.

```
revoke all on Suppliers, Products from Jones;
```

The *owner* of a table has the right to grant all privileges. The *grantee* cannot grant rights to a third party unless given **grant** privilege, by the grantor using the **with grant option** clause. Brown can give Smith the right to pass on **select** privilege to the 'Suppliers' table:

```
grant select on Suppliers to Smith with grant option;
```

The grantee can then pass on only those privileges to a third party. Smith can give Jones the right to read the 'Suppliers' table too.

```
grant select on Brown.Suppliers to Jones;
```

If a grantee has privileges revoked, then they are withdrawn from third parties too. The DBMS remembers the links in the chain. For example, if Brown revokes Smith's privileges, Jones loses them as well.

```
revoke all on Suppliers from Smith;
```

In *Oracle*, detailed privileges are programmed by assigning **roles**, which can give very fine control over users accessing or updating individual columns, in whatever way is desired. The standard roles are **connect**, **resource**, and **dba**. The **connect** role is given to users of an SQL application, **resource** allows programmers to create applications, and **dba** is reserved for the database administrator.