



# *the* training specialist

bringing people *and* technology together

Oracle  
Database

Oracle  
Development

Oracle Fusion  
Middleware

Oracle Business  
Intelligence

Oracle  
e-Business

MySQL

MariaDB

MongoDB

Unit Testing  
Frameworks

Maven

Hibernate

Acceptance Testing  
Frameworks

Java  
Technology

Web Development  
HTML & CSS

JavaScript  
& jQuery

PHP, Python  
Perl & Ruby

OOAD

Spring

UNIX

Solaris, AIX  
& HP-UX

Red Hat  
Linux

SUSE  
Linux

LPI  
Linux

Business Analysis  
ITIL® & PRINCE2®

London | Birmingham | Leeds | Manchester | Sunderland | Bristol | Edinburgh  
scheduled | closed | virtual training

[www.stayahead.com](http://www.stayahead.com)

+44 (0) 20 7600 6116

[sales@stayahead.com](mailto:sales@stayahead.com)



# **SQL for PostgreSQL**

Although StayAhead Training Limited makes every effort to ensure that the information in this manual is correct, it does not accept any liability for inaccuracy or omission.

StayAhead Training does not accept any liability for any damages, or consequential damages, resulting from any information provided within this manual.



# SQL for PostgreSQL

## 3 days training

[Home](#) [Enquiries](#)

---

### SQL for PostgreSQL Course Overview

The SQL for PostgreSQL course is designed to give delegates practical experience in writing SQL statements and scripts using PostgreSQL. The basic SELECT statement, the use of SQL functions, and the basic table and view handling statements are introduced.

Exercises and examples are used throughout the course to give practical hands-on experience with the techniques covered.

The delegate will learn and acquire skills as follows:

- Creating SQL statements to query database tables
- Using standard aggregate functions and related SELECT statement clauses
- Joining Tables
- Using basic subqueries and the set operators
- Using numeric, character and date functions
- Using conversion and other miscellaneous functions
- Using SQL parameters
- Using complex subqueries
- Creating and altering tables and views
- Inserting, updating and deleting rows in database tables
- Creating and listing indexes
- **Managing sequences**

### Who will the Course Benefit?

This is an ideal course for anyone who needs to use and understand the Structured Query Language (SQL) to query, report and update data in a PostgreSQL database.

### Course Objectives

The course aims to provide the delegate with the knowledge to be able to use SQL to manipulate data held in a PostgreSQL database and to progress their SQL skills beyond the basics.

### Requirements

There are no formal pre-requisites for this SQL for PostgreSQL course, although an understanding of relational databases and exposure to information technology in general would be useful.

## **Follow-On Courses**

- [PostgreSQL Administration](#)

### **Notes:**

- Course technical content is subject to change without notice.
- Course content is structured as sessions, this does not strictly map to course timings. Concepts, content and practicals often span sessions.

## TABLE OF CONTENTS

### 1. DATABASE CONCEPTS

What is a Database? .....	1- 2
Database Management Systems .....	1- 2
Overview of PostgreSQL .....	1- 3
Relational Database Structures .....	1- 4
Tables.....	1- 4
Rows .....	1- 4
Columns .....	1- 4
The Schema .....	1- 5
Primary Keys.....	1- 5
Unique Constraints.....	1- 5
Relationships and Foreign Keys.....	1- 6
Indexes.....	1- 6
The Data Dictionary .....	1- 7
Supported Data Types.....	1- 8
Workshop Exercises .....	1-10

### 2. USING PSQL

What is psql.....	2- 2
Getting Started.....	2- 3
Logging In .....	2- 3
Useful psql backslash commands .....	2- 5
Display a List of Databases .....	2- 6
List Available Tables .....	2- 7
Execute a Script.....	2- 8
Direct Output to a File.....	2- 8
Execute a Script from the Command Line .....	2- 9
Create and Edit a SQL File from a text editor .....	2-10
Switch Users .....	2-12

### 3. RETRIEVING DATA WITH THE SELECT STATEMENT

The SQL SELECT Statement.....	3- 2
The SELECT and FROM Clauses.....	3- 3
Conditions and the WHERE Clause .....	3- 4
Other Conditional Operators.....	3- 6
Logical Operators.....	3- 8
The ORDER BY Clause.....	3- 9
Column Aliases .....	3-11
Arithmetic Expressions .....	3-12
Precedence of Operators.....	3-13
Workshop Exercises .....	3-15

**4. AGGREGATE FUNCTIONS**

Overview of Built In Aggregate Functions .....	4- 2
The GROUP BY Clause .....	4- 4
The HAVING Clause.....	4- 6
Workshop Exercises .....	4- 7

**5. JOINING TABLES**

Understanding Joins .....	5- 2
Cross Joins .....	5- 3
Using Inner Joins .....	5- 4
Table Aliases.....	5- 6
Using Outer Joins .....	5- 7
Left Outer Joins.....	5- 7
Right Outer Joins .....	5- 8
Full Outer Joins.....	5- 8
Using Self Joins .....	5- 9
Additional ANSI Join Syntax.....	5-10
Traditional Join Syntax .....	5-11
Workshop Exercises .....	5-13

**6. BASIC SUBQUERIES AND SET OPERATORS**

Overview of Subqueries.....	6- 2
Using a Subquery as an Alternative to Join .....	6- 3
Set Operators.....	6- 4
The Union Operator .....	6- 4
The Intersect Operator.....	6- 5
The Except Operator .....	6- 5
Limit and Offset clauses .....	6- 6
Workshop Exercises .....	6- 9

**7. NUMERIC, CHARACTER AND DATE FUNCTIONS**

Overview of Functions .....	7- 2
Function Types .....	7- 2
Testing Functions.....	7- 2
Numeric Functions .....	7- 3
Character Functions.....	7- 5
String Concatenation .....	7- 7
Character Function Examples .....	7- 8
Date Arithmetic and Date Functions.....	7-12
Date and Time column types.....	7-12
Datestyle .....	7-13
Functions to return Date Time .....	7-14
Functions to extract components from Date Time .....	7-15
Date Arithmetic .....	7-16
Extract .....	7-17
Date_Part.....	7-18
Interval.....	7-19
Workshop Exercises .....	7-21



**8. CONVERSION AND MISCELLANEOUS FUNCTIONS**

Conversion Functions .....	8- 2
Converting and Formatting Dates.....	8- 2
Examples with TO_NUMBER.....	8- 5
Converting and Formatting Numbers .....	8- 6
CAST Function.....	8- 7
Miscellaneous Functions .....	8- 8
CASE Expressions.....	8- 8
The Simple CASE Expression .....	8- 7
The Searched CASE Expression.....	8- 8
Further Functions .....	8-11
The COALESCE Function.....	8-11
The NULLIF Function.....	8-12
Workshop Exercises .....	8-13

**9. COMPLEX SUBQUERIES**

Overview .....	9- 2
Subquery Usages .....	9- 2
Subquery in a Having Clause .....	9- 3
Subquery in a SELECT clause .....	9- 4
Subquery in a FROM clause .....	9- 5
WITH Queries (Common Table Expressions).....	9- 6
Correlated Subqueries.....	9- 7
Subqueries with Joins.....	9- 8
Multi Column Subqueries.....	9- 9
Using the ANY, ALL and SOME Operators.....	9-10
Subquery Rules .....	9-11
Workshop Exercises .....	9-13

**10. MANAGING DATA**

Inserting Rows .....	10- 2
INSERT using a Value List .....	10- 2
Updating Rows.....	10- 5
UPDATE with User-Specified Values.....	10- 5
UPSERT Rows .....	10- 6
Deleting Rows.....	10- 7
The DELETE Statement .....	10- 7
The TRUNCATE Statement.....	10- 8
Transaction Control.....	10- 9
The COMMIT and ROLLBACK Commands.....	10- 9
Transactions.....	10-10
Savepoints .....	10-10
Implicit Commits.....	10-13
Verifying Updates.....	10-14
Importing Data using csv files.....	10-16
Exporting Data .....	10-17
Copy command.....	10-17
Copy with \copy command.....	10-18
Workshop Exercises .....	10-19

**11. MANAGING TABLES**

Introduction .....	11- 2
Creating Tables.....	11- 3
Datatypes .....	11- 4
Constraints .....	11- 5
Constraint Examples.....	11- 7
Notes on Key Constraints .....	11- 8
Table Creation Example .....	11- 9
Managing Tables .....	11-10
Column Operations .....	11-10
Constraint Operations .....	11-13
Listing Constraints .....	11-14
Copying Tables in PostgreSQL .....	11-15
Dropping Tables.....	11-16
Workshop Exercises .....	11-17

**12. MANAGING INDEXES AND VIEWS**

Indexes.....	12- 2
What is an Index? .....	12- 2
Some Performance Considerations .....	12- 3
Managing B-Tree Indexes .....	12- 4
Listing Indexes .....	12- 6
Function Based Indexes .....	12- 8
Views.....	12- 9
What is a View? .....	12- 9
Creating Views.....	12- 9
Restrictions .....	12-10
Updatable Views .....	12-10
Dropping Views.....	12-11
Listing Views .....	12-12
Workshop Exercises .....	12-13

**13. MANAGING SEQUENCES**

Sequences .....	13- 2
Using Sequences.....	13- 4
Dropping Sequences .....	13- 6
Listing Sequences.....	13- 6
Workshop Exercises .....	13- 7

**APPENDICES**

<b>A. DEMONSTRATION TABLES .....</b>	<b>A1-A8</b>
<b>B. QUICK REFERENCE .....</b>	<b>B1-B2</b>

**Section 1**

**DATABASE  
CONCEPTS**



## DATABASE CONCEPTS

In this section the following topics relating to Database Concepts will be covered:

- ◆ **What is a Database?**
- ◆ **Relational Database Management Systems**
- ◆ **Overview of PostgreSQL**
- ◆ **Relational Database Structures**

## WHAT IS A DATABASE?

A database is an organised collection of data.

A simple example is a telephone directory; a collection of subscribers' names, addresses and telephone numbers.

In order to facilitate the retrieval of any one telephone number, the directory is organised in alphabetic sequence of subscriber surname and initials.

## DATABASE MANAGEMENT SYSTEMS

A Database Management System (DBMS) is a program which allows users to create and access databases stored on a computer according to a set of pre-defined rules.

The most common type of DBMS is a Relational Database Management System.

A Relational Database Management System, or RDBMS, is based on the concept that data can be grouped into logically distinct sets and relationships can be determined between those sets of data.

PostgreSQL is one such RDBMS. Others include Sybase, Informix, Ingres, Access and Oracle.

## OVERVIEW OF **POSTGRESQL**

PostgreSQL is an open source relational database management system (RDBMS).

PostgreSQL began as a research project in 1986 evolving from the Ingres research project that launched one of the early commercial relational databases. Originally named POSTGRES (Post-Ingres) it was renamed as PostgreSQL in 1996 to reflect the support for SQL, although it still frequently referred to as Postgres. It runs on all major operating systems, including Linux, UNIX and Windows.

It is fully ACID compliant and so supports transactions when maintaining data.

It has full support for joins, views, triggers, and stored procedures.

PostgreSQL conforms to the ANSI-SQL:2008 standard and offers most of the SQL:2008 data types.

It offers data integrity features including primary keys, foreign keys with restricting and cascading updates / deletes, check constraints, unique constraints, and not null constraints.

It also has a host of extensions and advanced features. Among these are auto-increment columns through sequences, and LIMIT / OFFSET allowing the return of partial result sets.

PostgreSQL's source code is available under an open source license: the PostgreSQL License. This license provides the freedom to use, modify and distribute PostgreSQL in any form. PostgreSQL is not only a powerful database system capable of running an enterprise, it is also a platform upon which to develop in-house, web, or commercial software products that require a fully-featured RDBMS.

## RELATIONAL DATABASE STRUCTURES

An organisation needs to hold information regarding many different **entities**; for example, information relating to its customers, products, suppliers, sales orders, invoices, employees, company cars, buildings and so on.

Each such entity has a number of **attributes**. For example, a customer has a name, an address, a telephone number, a credit limit etc.; whilst a company car has a model name, an engine size, a cost price, a depreciation rate and other attributes.

### TABLES

In a relational database, information about any one set or group of entities is stored in a single **table**. For example, information relating to all suppliers is held in one table, whilst information relating to all company cars is held in another table. A table is a two-dimensional object consisting of **rows** and **columns**. Each table in a database has a unique name.

### ROWS

Information about an individual entity constitutes a **row** in the relevant table. For example, data relating to an individual customer constitutes a row in the customer table; whilst data relating to a particular sales order constitutes a row in the sales order table.

### COLUMNS

A **column** (or data item) represents a single attribute common to all rows in a table. In an employee table, for example, each employee will have a name, a date of birth, a salary, a job title etc. Each of these attributes is represented by a single column in the table. Columns may contain **NULL** values if the data is unknown or not required for a particular row in the table. Each column within a table has a unique name.



## THE SCHEMA

A **schema** is a collection of related objects: tables, indexes, stored routines, etc.

### PRIMARY KEYS

In order to access an individual row within a table, each row must have one or more columns which contain values unique to that row. This column (or set of columns) is known as the **primary key**.

In most cases no single column (or reasonable set of columns) provides such a unique reference, so an extra column is frequently created for this purpose; a customer number, employee number or product code for example.

### UNIQUE CONSTRAINTS

A column (or set of columns) can be defined as **unique**. A unique constraint is similar to a primary key constraint, except that columns which form a unique constraint may be null. Unique in this case really means *unique when not null*.

Tables can have more than one unique constraint.

## RELATIONSHIPS AND FOREIGN KEYS

In reality there are many relationships between different entities. For example, there are relationships between customers and sales orders, sales orders and products, suppliers and stock items, suppliers and purchase orders, departments and employees, employees and company cars etc.

A relational database can reflect these relationships by means of foreign keys. A **foreign key** is a column (or set of columns) in a table which contain value(s) corresponding to the primary key of a related table.

For example, by including a column containing customer numbers in the sales order table a relationship between customers and sales orders can be determined.

When retrieving customer information, data relating to that customer's sales orders may also be retrieved by 'joining' the tables on the data common to both (the customer number). Similarly, when retrieving information for a sales order, data relating to the customer who placed the order can also be retrieved.

Foreign key relationships are used to maintain referential integrity.

## INDEXES

In order to improve speed of access to a table, one or more indexes may be created for it. An **index** consists of the values of one or more columns of the table together with the physical locations on disk of the associated rows.

An index is maintained (separately from its associated table) in value sequence; normally ascending numeric or alphabetic sequence.

Indexes may be unique or allow for duplicate values. If a unique index is specified, the DBMS prevents the addition of rows containing identical index values.

When a primary key or unique constraint is explicitly defined for a table a unique index is generated automatically for that table based on the primary key or unique constraint values.

## THE DATA DICTIONARY

The structure of the database is defined in a **Data Dictionary**. The Data Dictionary consists of a set of tables which can be interrogated by the user in the same way as database tables.

By querying the Data Dictionary the user can find out information about the tables which constitute the database. Such information includes, for example:

- ◆ *table attributes* - such as their names, sizes and physical locations.
- ◆ *column attributes* - such as their names, data types, sizes, the tables they apply to and whether or not they are allowed to have **NULL** values. Supported data types are listed overleaf.
- ◆ *primary key attributes* - such as their names, the tables they apply to and the columns from which they are formed.
- ◆ *foreign key attributes* - such as their names, the tables they apply to, the tables to which they relate and the columns from which they are formed.
- ◆ *index attributes* - such as their names, the tables they apply to, the columns from which they are formed and whether or not duplicate values are allowed.

The Data Dictionary also includes information about many other objects held on the database such as views, synonyms, sequences, triggers, procedures, functions and packages and other information such as valid users and access privileges.

In PostgreSQL, metadata about schemata is contained in system catalogs. Table names begin with **pg\_** so for example **pg\_tables** and **pg\_indexes** could be queried for information about a particular table or index in a schema.

PostgreSQL also provides the **Information Schema**, which has tables about columns, constraints, roles and privileges, tables and views, and schemata amongst others.

## SUPPORTED DATA TYPES

The following main data types are supported in PostgreSQL:

<b>CHAR( <i>n</i> )</b>	<p>A fixed-length character string <i>n</i> characters long. Default length is 1 character if no size is specified. The maximum allowed size is 10485760.</p> <p>Equivalent: CHARACTER ( <i>n</i> )</p> <p>Example: national_ins_nr <b>CHAR</b>( 9 )</p>
<b>VARCHAR( <i>n</i> )</b>	<p>A variable length character string up to <i>n</i> characters long. The maximum allowed size is 10485760.</p> <p>Equivalent: CHARACTER VARYING( <i>n</i> ).</p> <p>Example: surname <b>VARCHAR</b>( 30 )</p>
<b>TEXT</b>	<p>Store strings of any length, up to a maximum size of about 1Gb.</p> <p>Equivalent: VARCHAR</p>
<b>SMALLINT</b>	<p>A 2 byte signed integer in the range -32,768 to 32,767</p> <p>Equivalent: INT2</p>
<b>INT</b>	<p>A 4 byte signed integer in the range -2,147,483,648 to 2,147,483,647</p> <p>Equivalent: INT4</p>
<b>BIGINT</b>	<p>An 8 byte signed integer in the range -9223372036854775808 to +9223372036854775807</p> <p>Equivalent: INT8</p>
<b>NUMERIC( <i>p</i>, <i>s</i> )</b>	<p>A signed number <i>p</i> digits long (in total) including <i>s</i> decimal places with <math>p \leq 1000</math> and <math>s \leq p</math>.</p> <p>Equivalent: DECIMAL( <i>p</i>, <i>s</i> )</p> <p>Example: salary <b>NUMERIC</b>( 6,2 )</p>
<b>NUMERIC</b>	<p>Numeric values of any precision and scale can be stored, up to 131072 digits before the decimal point and 16383 digits after the decimal point</p>
<b>DATE</b>	<p>A valid date</p> <p>Example: start_date <b>DATE</b></p>
<b>TIME [with time zone]</b>	<p>A valid time, optionally with a time zone.</p>
<b>TIMESTAMP [with time zone]</b>	<p>Date and time, optionally with a timezone.</p>
<b>INTERVAL</b>	<p>To store a unit of time</p>

## WORKSHOP EXERCISES

Take some time to study the tables described in Appendix A. These are the tables which you will be working with throughout this course.

The names of the tables are indicated within parentheses.

The name, data type and length (specified in brackets after the data type) are shown for each column. If a column is not allowed to contain a **NULL** value this, too, is indicated.

Note the relationships which exist between these tables by referring to the specified primary and foreign key columns. Draw a simple diagram which indicates these relationships. (Such a diagram is known as an entity-relationship diagram.)

Is there a relationship between the employee and salary grade tables?

**Note:**

Table names and column names are shown in lowercase in Appendix A and throughout this workbook in order to distinguish them from SQL keywords.



## **Section 2**

**USING  
psql**





## USING PSQL

In this section the following topics relating to Using psql will be covered:

- ♦ **What is psql**
- ♦ **Logging In and Out**
- ♦ **Switching User Accounts**
- ♦ **Describing Tables**
- ♦ **Entering and Executing SQL Statements**
- ♦ **Creating, Editing and Executing SQL Script Files**

## WHAT IS PSQL?

psql is a basic **command line utility** which enables the user to interrogate the database; create, alter and drop tables; add, update and delete rows; specify primary keys, foreign keys and indexes; grant and revoke access privileges and so on.

In order to perform these functions the user enters and executes a series of SQL statements; these may be saved for re-use if required.

SQL (Structured Query Language) was developed in the 1970's at IBM. As the name suggests it was initially devised for querying a (relational) database only. It is now used for all database operations.

The version of SQL currently supported by PostgreSQL is based on the latest ANSI standards for SQL and incorporates a number of extensions.

There are essentially three types of SQL statement.

<b>DML</b>	Data Manipulation Language statements for managing stored data	(Insert, Update, Delete, Select)
<b>DDL</b>	Data Definition Language statements for managing database objects and setting parameters	(Create, Alter, Drop, Rename ... )
<b>DCL</b>	Data Control Language statements for assigning privileges	(Grant, Revoke ... )

## GETTING STARTED

### LOGGING IN

In order to log in to psql the Database Administrator must already have allocated a user name and a password for your use.

psql is started at an operating system prompt by entering **psql**, or from a graphical user interface by selecting the appropriate icon.

```
C:\Program Files\PostgreSQL\10\bin>psql -d sat -U postgres -W
Password for user postgres:
psql (10.1)
WARNING: Console code page (850) differs from Windows code page (1252)
         8-bit characters might not work correctly. See psql reference
         page "Notes for Windows users" for details.
Type "help" for help.
sat=#
```

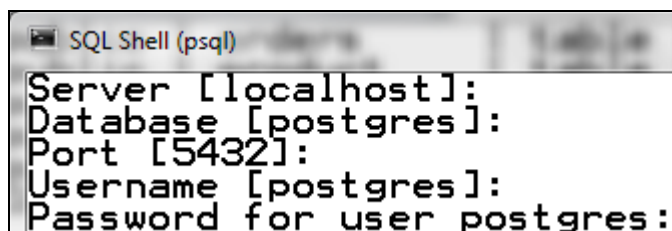
In the example above the psql executable is being run with the arguments provided:

- d** the database name, sat in this case
- U** the username, postgres in this case
- W** which will cause the user to be prompted to supply their (masked) password

Other arguments which can be used where needed are:

- h** to select a servername / hostname
- p** to indicate the port (default: 5432)
- f** followed by a filename to execute a script from command line (see page 9 for an example)

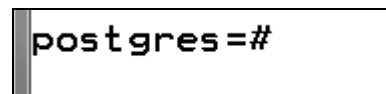
In Windows you could access psql from **Start > PostgreSQL 10 > SQL Shell (psql)**. You will then see the following:



```
SQL Shell (psql)
Server [localhost]:
Database [postgres]:
Port [5432]:
Username [postgres]:
Password for user postgres:
```

The user is prompted for Server, Database, Port and Username. To accept the default value for each hit Return. Note that the password is not displayed.

When these credentials have been entered correctly psql displays its own command line prompt; by default `databases=#` for a superuser or `databases=>` for a regular user.

A rectangular box representing a terminal window. Inside the box, the text 'postgres=#' is displayed in a monospaced font. A vertical grey bar is on the left side of the box, representing the terminal's scrollbar or a window border.

The following pages shows a number of tasks that can be carried using backslash ( \ ) commands from this prompt.

## USEFUL PSQL BACKSLASH COMMANDS

<b>\?</b>	Lists help on all \ commands
<b>\c dbname</b>	Change to the named database
<b>\d tablename</b>	Describe a table
<b>\d+ tablename</b>	Describe a table, with more detail
<b>\dn</b>	List available Schemas in current database
<b>\df</b>	List available Functions in the current database
<b>\dt</b>	List tables in the current database
<b>\dv</b>	List available Views
<b>\du</b>	List Users and their Roles
<b>\g</b>	Execute the previous command
<b>\i filename</b>	Execute commands from a file (see example)
<b>\l</b>	Display a list of databases
<b>\o filename</b>	Direct output to a file instead of the screen (see example)
<b>\q</b>	Hit enter then press any key to quit
<b>\s</b>	Display command history (where version supports this)
<b>\s filename</b>	Save command history to file
<b>\timing</b>	To switch on (and off) the display of timing for queries
<b>\x</b>	Switch on/ off Expanded display ("Pretty format")

Examples of some of these commands in use are set out on the following pages.

## DISPLAY A LIST OF DATABASES

Use the `\l` command to display a list of available databases.

```
postgres=# \l
   Name           | Owner          | Encoding | Access privileges |
-----+-----+-----+-----+
 dvdrental        | postgres       | UTF8     |                    |
 Kingdom.1252     |                |          |                    |
 postgres         | postgres       | UTF8     |                    |
 Kingdom.1252     |                |          |                    |
 sat              | postgres       | UTF8     |                    |
 Kingdom.1252     |                |          |                    |
 template0        | postgres       | UTF8     |                    |
 Kingdom.1252     | =c/postgres   |          |                    |
                  | postgres=CtC/postgres |          |                    |
 template1        | postgres       | UTF8     |                    |
 Kingdom.1252     | =c/postgres   |          |                    |
                  | postgres=CtC/postgres |          |                    |
(5 rows)
```

## CHANGE DATABASES

Use `\c` to switch to another database:

```
postgres=# \c sat
WARNING: Console code page (850) differs from Windows code page (1252)
        8-bit characters might not work correctly. See psql reference
        page "Notes for Windows users" for details.
You are now connected to database "sat" as user "postgres".
sat=#
```

A confirmation message is displayed and the command line prompt is updated with the new database name.

## LIST AVAILABLE TABLES

Use `\dt` to list available tables in the current database.

```
postgres=# \c sat
WARNING: Console code page (850) differs
        8-bit characters might not work
        page "Notes for Windows users"
You are now connected to database "sat"
sat=# \dt
```

Schema	Name	Type	Owner
public	client	table	postgres
public	customer	table	postgres
public	department	table	postgres
public	employee	table	postgres
public	employee2	table	postgres
public	orders	table	postgres
public	product	table	postgres
public	prospect	table	postgres
public	salgrade	table	postgres
public	supplier	table	postgres

(10 rows)

## DESCRIBE A TABLE

Use `\d` and the tablename to describe a table. The columns, their datatypes, constraints and indexes are shown.

```
sat=# \d orders
```

Column	Type	Collation	Nullable	Default
order_nr	smallint		not null	
product_code	character varying(10)		not null	
customer_nr	smallint		not null	
order_date	date			
quantity	smallint		not null	

Indexes:  
 "ord\_pkey" PRIMARY KEY, btree (order\_nr)  
 Foreign-key constraints:  
 "ord\_fkey\_cust" FOREIGN KEY (customer\_nr) REFERENCES customer(customer\_nr)  
 "ord\_fkey\_prod" FOREIGN KEY (product\_code) REFERENCES product(product\_code)

## EXECUTE A SCRIPT

Here is an example of using `\i` to execute a file. Note that the `.sql` filename extension is not implicit in PostgreSQL.

```
sat=# \i c:/temp/staff.sql
 employee_nr | surname | initials | sex | start_date |
ry | department_nr
-----+-----+-----+-----+-----+-----
-----+-----+-----+-----+-----+-----
.00 |      1000 | King    | J.R. | M | 2016-06-24 |
.00 |      1036 | Blake   | S.T. | M | 2016-09-25 |
.00 |      30
```

## DIRECT OUTPUT TO A FILE

```
sat=# \o c:/temp/jbctest
sat=# select *
sat-# from employee;
sat=# select *
sat-# from department;
sat=# \o
```

Use `\o` and the pathname and filename to direct output to a file.

The first line enables the capture of output to the named file.

The second line directs a full listing of the employee table to that file.

The fourth line will append a full listing of the department table to the same file.

The final line, with `\o` alone, terminates the direction of output to the file so that for any subsequent commands, output will revert to the screen.



## EXECUTE A SCRIPT AND DIRECT THE OUTPUT TO A FILE

This just puts together the previous two examples:

```
sat=# \o c:/temp/manager_list.txt
sat=# \i c:/temp/managers.sql
sat=# \o
sat=#
```

Line 1 switches on the capture of data into a file called manager\_list.txt.

Line 2 retrieves and executes the managers.sql script.

Line 3 switches off the capture of data to the file. Subsequent script output will be to the screen.

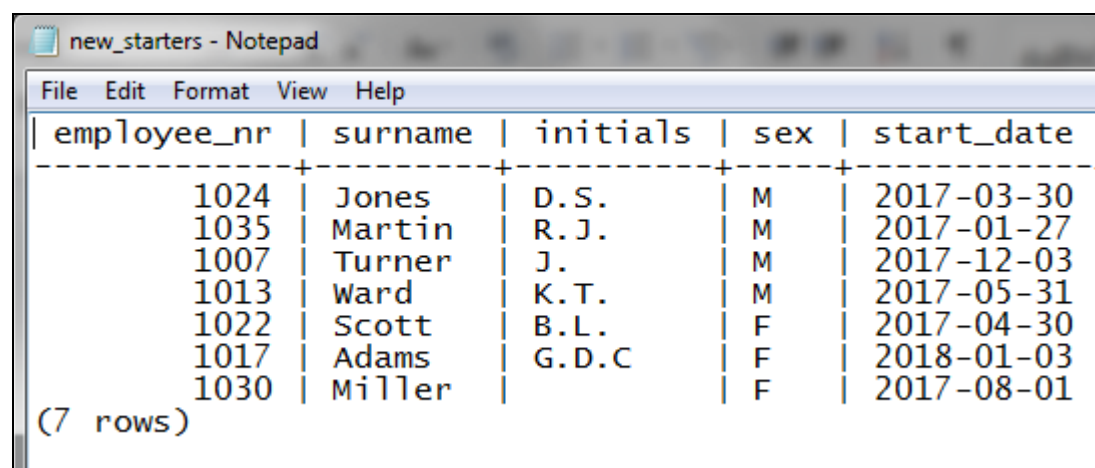
## EXECUTE A SCRIPT FROM THE COMMAND LINE

This example shows the user providing connection credentials, retrieving and running a script and then being returned to the command line.

```
C:\Program Files\PostgreSQL\10\bin>psql -d sat -U postgres -W -f c:\temp\new_starters.sql
Password for user postgres:
C:\Program Files\PostgreSQL\10\bin>
```

- d the database name
- U the username
- W to ensure the password is prompted for
- f filename

In this case the script new\_starters.sql runs a listing of recent joiners and directs the output to a text file.



employee_nr	surname	initials	sex	start_date
1024	Jones	D.S.	M	2017-03-30
1035	Martin	R.J.	M	2017-01-27
1007	Turner	J.	M	2017-12-03
1013	Ward	K.T.	M	2017-05-31
1022	Scott	B.L.	F	2017-04-30
1017	Adams	G.D.C	F	2018-01-03
1030	Miller		F	2017-08-01

(7 rows)

## CREATING AND EDITING A SQL FILE FROM A TEXT EDITOR

### CREATING A SQL FILE

The command `\e` can be used to edit the SQL statement currently in the command buffer. The default editor for the operating system will be used.

If vi is the default editor, save the file using the command `:w filename.sql`. If Notepad is the default editor, select the **File -> Save As** option and enter the required filename (and **.sql** extension) in double quotes.

### EDITING A SQL FILE

If `\e` is used with the name of a SQL script file, the file is opened for editing.

`\e [pathname]filename[.ext]`

For example:

```
sat=#  
sat=#  
sat=# \e c:/temp/managers.sql
```

Edit as required and save.

### EXAMPLE

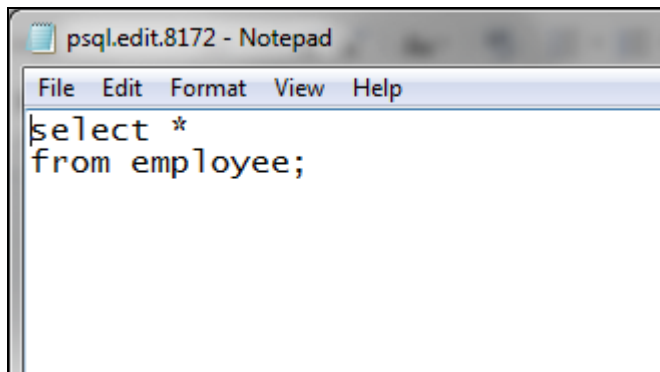
The user runs a query to retrieve all the data from the Employee table:

```
sat=# select *  
sat-# from employee;
```

They decide to edit the previous commands on an ad hoc basis to retrieve manager records only.

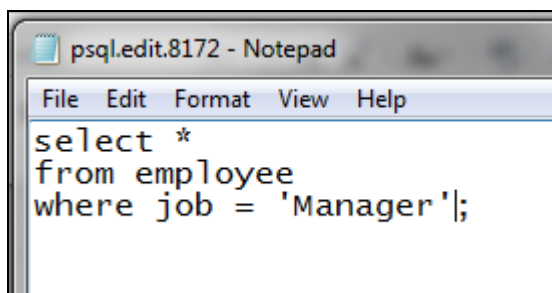
Enter `\e`

The previous commands are displayed in the text editor, in this case Notepad:

A screenshot of a Notepad window titled 'psql.edit.8172 - Notepad'. The menu bar includes 'File', 'Edit', 'Format', 'View', and 'Help'. The text area contains the SQL query: 

```
select *  
from employee;
```

Alterations can then be made.

A screenshot of a Notepad window titled 'psql.edit.8172 - Notepad'. The menu bar includes 'File', 'Edit', 'Format', 'View', and 'Help'. The text area contains the edited SQL query: 

```
select *  
from employee  
where job = 'Manager';
```

To save the changes and return to psql, in this case use **File > Save** and close Notepad. The edited query will be returned to psql and executed. If using the vi editor, you would use :w to write the change and :q to quit, or :x to do both things together.

## SWITCH USERS

To change users during a session you can use the **SET ROLE** command, as in PostgreSQL when a new user is created it is done via the **CREATE ROLE** command.

For example, having logged on as the superuser postgres, the user / role is changed to user0.

```
sat=#  
sat=#  
sat=# set role user0;  
SET  
sat=>
```

The # prompt becomes a > prompt due to the change from a superuser to a regular user.

## **Section 3**

# **RETRIEVING DATA WITH THE SELECT STATEMENT**



## RETRIEVING DATA WITH THE SELECT STATEMENT

In this section the following topics relating to Retrieving Data with the Select Statement will be covered:

- ♦ **The SQL SELECT Statement**
- ♦ **Basic SELECT Statements**
- ♦ **The SELECT and FROM Clause**
- ♦ **Conditions and the WHERE Clause**
- ♦ **Conditional Operators**
- ♦ **Logical Operators**
- ♦ **The ORDER BY Clause**
- ♦ **Column Aliases**
- ♦ **Arithmetic Expressions**
- ♦ **Precedence of Operators**

## THE SQL SELECT STATEMENT

The **SELECT** statement is used to retrieve data from the database. It consists of the following basic clauses:

<b>SELECT</b>	Specifies the columns which are to be retrieved
<b>FROM</b>	Specifies the tables from which data is to be retrieved
<b>WHERE</b>	Defines any selection criteria required
<b>GROUP BY</b>	Specifies how output is to be grouped and summarised
<b>HAVING</b>	Determines which group summaries to include in the output
<b>ORDER BY</b>	Defines how the output will be sorted

Only the **SELECT** clause is mandatory; the rest are optional.

### Example

```
SELECT current_user, current_date
```

Displays the current user role and date.

This section describes the standard SQL **SELECT** statement; PostgreSQL-specific SQL functions are described in later sections.

Throughout this and subsequent sections SQL keywords are shown in capital letters to distinguish them from column names, table names, aliases and other database items or user-defined words. In practice case is ignored by the SQL engine.

Examples in this and all the following sections are based on the tables specified in Appendix A.



## THE SELECT AND FROM CLAUSES

Throughout this manual square brackets [ ] are used to indicate optional values and are not required as part of the syntax of commands

To retrieve data from tables, the following syntax is used:

```
SELECT [ DISTINCT ] item-list  
FROM table-list
```

where ***item-list*** is a comma-separated list of items to be retrieved (database columns, functions, fixed text etc.) and ***table-list*** is a comma-separated list of tables from which the columns are to be retrieved.

The keyword **DISTINCT** ensures only unique rows will be returned.

If an asterisk is used in place of ***item-list*** all columns from the specified tables will be retrieved.

### Examples

```
SELECT * FROM employee
```

Selects all columns from all rows of the employee table.

```
SELECT supplier_nr, name, telephone FROM supplier
```

Selects the specified columns from all rows of the supplier table.

```
SELECT DISTINCT job FROM employee
```

Lists all distinct jobs from the employee table.

## CONDITIONS AND THE WHERE CLAUSE

The **WHERE** clause is used to specify the conditions under which data is to be retrieved. Its basic syntax is:

**WHERE** *conditions*

The simplest condition is a test for equality. For example:

```
SELECT employee_nr, surname, initials, salary
FROM employee
WHERE department_nr = 20
```

This statement displays all employees working in department 20

The main conditional operators include:

=	equal to	!= or <>	not equal to
>	greater than	<	less than
>=	greater than or equal to	<=	less than or equal to

### Examples

This statement displays all employees earning no more than £2000 per month

```
SELECT * FROM employee
WHERE salary <= 2000
```

This statement displays all employees who started after 1<sup>st</sup> March 2017

```
SELECT surname, start_date FROM employee
WHERE start_date > '2017-03-01'
```

Dates must be bounded by single quotes. In addition, dates must be specified in the expected format (which is yyyy-mm-dd).

This statement displays all customers based in London

```
SELECT surname, initials, town, credit_limit  
FROM    customer  
WHERE town = 'London'
```

String values must be quoted as shown above. It is also important to note that string comparisons are case sensitive. In the example above only those rows where the *town* column contains the capitalised value *London* will be shown.

To overcome this restriction, PostgreSQL supplies the functions **LOWER** and **UPPER** which may be used to convert column values to lower or upper case respectively. These are covered later in this course

Where a string value contains a single quote or apostrophe, the embedded quote (apostrophe) must be specified as two *single* quotes. The example below shows data for all employees whose surname is *O'Neill*:

```
SELECT surname, job, salary  
FROM    employee  
WHERE surname = 'O'Neill'
```

An alternative solution is for the user to prefix the string with an E and then use the \ character to escape the metacharacter meaning of the immediately following quote character as shown here:

```
SELECT surname, job, salary  
FROM    employee  
WHERE surname = E'O\Neill'
```

In this way the apostrophe is interpreted as a literal and not as delimiting the string.

## OTHER CONDITIONAL OPERATORS

<b>IN</b>	Tests if an item value appears in a specified list of values.
<b>BETWEEN</b>	Tests if an item value lies (inclusively) between two specified values.
<b>IS NULL</b>	Tests if an item has a <b>NULL</b> value. <b>NULL</b> is neither zero (in a numeric item) nor spaces (in a character item) - it is the absence of a value.
<b>LIKE</b> Or ~~	Tests if an item value matches a string containing wildcard characters. The wildcard characters are: %     Meaning zero or more occurrences of any character _ (underscore) Meaning a single occurrence of any character
<b>ILIKE</b> Or ~~*	As for LIKE but renders it case-insensitive.

The **IN**, **BETWEEN**, **IS NULL**, **LIKE** and **ILIKE** operators can be negated using the keyword **NOT** as shown below:

<b>NOT IN</b>	<b>NOT BETWEEN</b>	<b>IS NOT NULL</b>	<b>NOT LIKE</b> Or: !~~	<b>NOT ILIKE</b> Or: !~~*
---------------	--------------------	--------------------	----------------------------	------------------------------

### Examples

This statement displays Customers who are based in Birmingham, London or Manchester.

```
SELECT *
FROM customer
WHERE town IN ( 'Birmingham', 'London', 'Manchester' )
```

This statement displays suppliers who have no telephone number.

```
SELECT supplier_nr, name
FROM supplier
WHERE telephone IS NULL
```

This statement displays employees with salaries either below 2000 or above 3000.

```
SELECT department_nr, employee_nr, surname, salary
FROM employee
WHERE salary NOT BETWEEN 2000 AND 3000
```

This query displays orders for the year 2017:

```
SELECT customer_nr, order_nr, order_date, product_code, quantity
FROM orders
WHERE order_date BETWEEN '2017-01-01' AND '2017-12-31'
```

The following query would show only 4 letter surnames as there are just 4 underscore characters in the comparison string. Note that wild card characters are only interpreted if the LIKE operator is used. Wild card characters are treated as literal values with an = symbol.

```
SELECT surname
FROM employee
WHERE surname LIKE '____'
```

The following example shows all products where the description begins with an A, whether in lower or upper case, and is of any length.

```
SELECT product_code, description, cost_price
FROM product
WHERE description ILIKE 'A%'
```

The example below shows the backslash ( \ ) being used as an escape character with the LIKE (or ~~ as shown here) operator:

```
SELECT details
FROM atable
WHERE details ~~ '%5\%%%' ESCAPE '\'
```

The *details* column will be selected only if it contains the string **5%**. The ESCAPE keyword (and quoted backslash) can be omitted. It would be required to use some other character as an escape character e.g.: a '?'.  
 The *details* column will be selected only if it contains the string **5%**. The ESCAPE keyword (and quoted backslash) can be omitted. It would be required to use some other character as an escape character e.g.: a '?'.

## LOGICAL OPERATORS

The logical operators **AND** and **OR** can be used to combine conditions. Logical **ANDs** take precedence over (ie: are evaluated before) logical **ORs**. Parentheses can be used to change the normal order of evaluation.

### Examples

This statement displays the manager of Department 10. Both parts of the condition have to be met as **AND** strengthens the current condition.

```
SELECT * FROM employee
WHERE department_nr = 10
      AND job = 'Manager'
```

This statement displays everyone who works in department 10 and all managers regardless of department. **OR** starts a fresh condition.

```
SELECT * FROM employee
WHERE department_nr = 10
OR     job = 'Manager'
```

This example retrieves data relating to managers in Department 10 and analysts in Department 20.

```
SELECT * FROM employee
WHERE department_nr = 10
      AND job = 'Manager'
OR     department_nr = 20
      AND job = 'Analyst'
```

In order to change the normal order of evaluation parentheses can be used:

```
SELECT * FROM employee
WHERE department_nr = 30
      AND (    job = 'Manager'
            OR salary < 1000 )
```

This will retrieve data just from Department 30 for both the manager and those earning less than £1000 per month.

## THE ORDER BY CLAUSE

The **ORDER BY** clause is used to define how the output should be sequenced. One or more items may be defined; the precedence operates from left to right.

By default output is sorted in ascending host system collating sequence; in ASCII, space, digits, uppercase characters, lowercase characters. This can be specified explicitly by using the keyword **ASC** after the name of the item. The order of output can be reversed by using the keyword **DESC** following the name of an item.

### Examples

This statement displays all suppliers in alphabetical sequence of supplier name.

```
SELECT * FROM supplier  
ORDER BY name
```

This statement displays products in descending order of sales price for those with more than 100 in stock.

```
SELECT * FROM product  
WHERE instock > 100  
ORDER BY sales_price DESC
```

This statement displays orders placed in the years 2016 and 2017 shown in ascending alphabetic order of product within descending date sequence.

```
SELECT * FROM orders  
WHERE order_date BETWEEN '2016-01-01' AND '2017-12-31'  
ORDER BY order_date DESC, product_code ASC
```

This statement shows all employees in alphabetic order of name within numeric department sequence.

```
SELECT department_nr, surname, initials, job, salary  
FROM employee  
ORDER BY department_nr, surname, initials
```

In the ORDER BY clause the ordinal position of the items specified in the SELECT clause can be used instead of column names as shown below. This will sort the data firstly numerically by department, then alphabetically by name. This technique can be useful when dealing with expressions mentioned later in this section.

```
SELECT surname, initials, department_nr FROM employee
WHERE job LIKE '%Clerk%'
ORDER BY 3, 1, 2
```

**NULL** values are sorted last in a (default) ascending sort; first in a descending sort. If required, this default sorting order can be changed using the **NULLS FIRST | LAST** keywords to an **ORDER BY** clause.

```
SELECT surname, job, salary, department_nr, manager
FROM employee
ORDER BY manager DESC NULLS LAST
```

The default sort order is the collation sequence set at database creation, which cannot be changed. However, it can be overridden by including a **COLLATE** clause.

```
SELECT surname, job, salary, department_nr, manager
FROM employee
ORDER BY manager COLLATE "es_ES"
```



## COLUMN ALIASES

Alias names can be specified for columns in the **SELECT** clause. The main reasons for doing so are to help identify columns on output and to give names to selected items which result from calculations or functions (described later).

Alias names are specified following the selected items and the optional (but recommended) keyword **AS**. They must be bounded by double quotes if they contain spaces or other special characters.

Unquoted alias names are converted to lowercase. If case is to be preserved then double quotes should be used.

*Column aliases can be referenced only in the **ORDER BY** clause.*

### Example

This statement displays all Departments located outside London.

```
SELECT department_nr AS Dept, name AS Department_Name, location
FROM department
WHERE location != 'London'
ORDER BY Department_Name
```

This statement displays employee data with user-friendly headings:

```
SELECT surname AS "Surname", job AS "Job",
      salary AS "Monthly Salary"
FROM employee
ORDER BY "Monthly Salary"
```

## ARITHMETIC EXPRESSIONS

Arithmetic expressions can be used in both the **SELECT** and **WHERE** clauses. The standard arithmetic operators **+**, **-**, **\*** and **/** can be used; parentheses are also allowed so that more complex calculations can be performed. It is important to note that any calculation involving a **NULL** value will result in a **NULL** value.

Column aliases should ideally be given to all items which represent arithmetic (and other) expressions in order to help identify the resulting output.

### Examples

This statement displays annual instead of monthly salaries.

```
SELECT surname, initials, salary * 12 AS Annual_Salary
FROM employee
ORDER BY Annual_Salary
```

This statement shows the markup percentage on products.

```
SELECT description, cost_price, sales_price,
        ( sales_price - cost_price ) / cost_price * 100 AS pct_profit
FROM product
ORDER BY pct_profit DESC
```

This statement shows products with a markup of more than 10%

```
SELECT product_code, description,
        ( sales_price - cost_price ) / cost_price * 100 AS Profit
FROM product
WHERE ( sales_price - cost_price ) / cost_price * 100 > 10
ORDER BY product_code
```

Note that column aliases cannot be used in a **WHERE** clause.

Arithmetic may also be performed on date items. Examples are given in a later section where other PostgreSQL date functions are introduced.

## PRECEDENCE OF OPERATORS

- ◆ Mathematical operators ( **+** **-** **/** **\*** ) take precedence over (i.e. are evaluated before) conditional operators.
- ◆ Division and multiplication have equal precedence.
- ◆ Both division and multiplication take precedence over addition and subtraction.
- ◆ Addition and subtraction have equal precedence.
- ◆ Conditional operators ( **>** **<** **=** etc.) take precedence over logical operators.
- ◆ The logical **AND** operator takes precedence over the logical **OR** operator.
- ◆ Operators inside parentheses take precedence over operators outside parentheses.



## WORKSHOP EXERCISES

Write SQL statements or scripts to answer the following questions.

1. List details of employees that are salesmen.

2. List those employees that have no manager.

3. List those suppliers that are not from Brighton or Birmingham.

4. List the order number, product code and quantity from the *orders* table where quantity is greater than 75.

5. List all orders for customer number 1317.

6. List details of products with a sales price at least £100 in excess of the cost price.

7. List customers who live in towns, which do not contain the letter m (uppercase or lowercase).

8. Apart from managers and the chairman, which employees earn more than £30,000 per year?

*Optional additional questions (if time permits) on next page*

**OPTIONAL ADDITIONAL QUESTIONS (IF TIME PERMITS)**

9. List employees with salaries from 1000 to 2000 inclusive.

10. List orders with dates more recent than 1st January 2017.

11. List customers with 'i' as the third letter in their surname.

12. List customers with exactly one 'l' in the surname.  
(just use lower case - no surnames start with 'L')

13. List prospects which have an 'm' as either the second letter or the second from last letter of their surname. but which do not have an 'm' anywhere else in their surname.

## SAMPLE ANSWERS

```
1.  SELECT surname, initials, job
FROM employee
WHERE job = 'Salesman'
```

```
2.  SELECT surname, initials, manager
FROM employee
WHERE manager IS NULL
```

```
3a. SELECT supplier_nr, name, town
FROM supplier
WHERE town NOT IN ( 'Birmingham', 'Brighton' )
```

```
3b. SELECT supplier_nr, name, town
FROM supplier
WHERE town != 'Birmingham'
AND town != 'Brighton'
```

```
4.  SELECT order_nr, product_code, quantity
FROM orders
WHERE quantity > 75
```

```
5.  SELECT *
FROM orders
WHERE customer_nr = 1317
```

```
6.  SELECT *  
    FROM product  
    WHERE ( sales_price - cost_price ) > 100
```

```
7a. SELECT surname, initials, town  
    FROM customer  
    WHERE town NOT LIKE '%M%' AND town NOT LIKE '%m%'
```

```
7b. SELECT surname, initials, town  
    FROM customer  
    WHERE town NOT ILIKE '%M%'
```

```
8a. SELECT surname AS Name, job AS Job_Title,  
        salary * 12 AS "Annual Salary"  
    FROM employee  
    WHERE job != 'Manager' AND job != 'Chairman'  
        AND salary * 12 > 30000
```

```
8b. SELECT surname AS Name, job AS Job_Title,  
        salary * 12 AS "Annual Salary"  
    FROM employee  
    WHERE job NOT IN ('Manager','Chairman')  
        AND salary * 12 > 30000
```

```
9.  SELECT *  
    FROM employee  
    WHERE salary BETWEEN 1000 AND 2000
```

```
10. SELECT *  
    FROM orders  
    WHERE order_date > '2017-01-01'
```



```
11.  SELECT *
      FROM customer
      WHERE surname ~~ '___i%'      -- 2 underscores before the 'i'
```

```
12.  SELECT *
      FROM customer
      WHERE surname LIKE '%l%'
      AND surname NOT LIKE '%l%l%l%'
```

```
13.  SELECT *
      FROM prospect
      WHERE (surname LIKE '_m%' OR surname LIKE '%m_')
      AND surname NOT LIKE '___%m%__' /* 2 underscores at start & end */
      AND surname NOT LIKE 'M%'
      AND surname NOT LIKE '%m'
```



## **Section 4**

# **AGGREGATE FUNCTIONS**



## AGGREGATE FUNCTIONS

In this section the following topics relating to Aggregate Functions will be covered:

- ♦ **Aggregate Functions**
- ♦ **The GROUP BY Clause**
- ♦ **The HAVING Clause**

## OVERVIEW OF BUILT IN AGGREGATE FUNCTIONS

ANSI standard SQL includes a number of built-in *aggregate* functions to provide for summarised output. The most common of these are described below. Functions specific to PostgreSQL are discussed in later sections.

The most common standard aggregate functions are:

<b>COUNT ( * )</b>	Which counts the number of selected rows
<b>COUNT ( DISTINCT <i>item</i> )</b>	Which counts all unique values (except nulls) of <i>item</i> in the selected rows
<b>MAX ( <i>item</i> )</b>	Which displays the highest value of <i>item</i>
<b>MIN ( <i>item</i> )</b>	Which displays the lowest value of <i>item</i>

And the following functions which operate only on numeric items:

<b>AVG ( <i>item</i> )</b>	<b>SUM ( <i>item</i> )</b>
<b>VARIANCE ( <i>item</i> )</b>	<b>STDDEV ( <i>item</i> )</b>

All the aggregate functions can use **DISTINCT**, but it is only meaningful with **COUNT**.

Aggregate functions ignore **NULL** values.

The results of aggregate functions are only calculated after processing both the **WHERE** and the **GROUP BY** clauses.

Aggregate functions cannot be used in a **WHERE** clause as they are not *single row* functions. The **HAVING** clause (described later) is used instead to provide conditions on summarised group values.

Some examples are shown on the next page.

## Examples

List salary statistics for the employees:

```
SELECT SUM ( salary ), AVG ( salary ), MAX ( salary ), MIN ( salary )  
FROM    employee
```

As before, but this time excluding the chairman's data:

```
SELECT SUM ( salary ), AVG ( salary ), MAX ( salary ), MIN ( salary )  
FROM    employee  
WHERE    job != 'Chairman'
```

Count all Birmingham customers in the customer table:

```
SELECT COUNT ( * ) AS Birmingham_Customers  
FROM    customer  
WHERE    town = 'Birmingham'
```

Count how many different jobs there are in the employee table:

```
SELECT COUNT ( DISTINCT job ) AS Number_of_Jobs  
FROM    employee
```

Display the Average and the Maximum credit limit from the customer table:

```
SELECT AVG (credit_limit), MAX (credit_limit)  
FROM    customer
```

Show the average quantity of all orders taken since the beginning of 2017:

```
SELECT AVG (quantity) AS Average_Quantity  
FROM    orders  
WHERE    order_date >= '2017-01-01'
```

## THE GROUP BY CLAUSE

When column names and aggregate functions are specified in the **SELECT** clause, the **GROUP BY** clause is mandatory. Furthermore the **GROUP BY** clause must identify *all* the non-aggregate items specified in the **SELECT** clause.

The **GROUP BY** clause enables aggregates to be accumulated and summarised on output. The following example uses the *employee* table to calculate the mean annual salary by department:

```
SELECT    department_nr, AVG (salary )  
FROM      employee  
GROUP BY department_nr
```

If an **ORDER BY** clause is not used, the output sequence is indeterminate.

It is possible to group by more than one column. This statement shows the average salary for each unique combination of department and job.

```
SELECT    department_nr, job, AVG (salary)  
FROM      employee  
GROUP BY department_nr, job
```

It is also possible to include more than one aggregate function:

```
SELECT    department_nr, job, AVG (salary), MAX (salary)  
FROM      employee  
GROUP BY department_nr, job
```

An explicit order by clause should be included if the ordering of the data is significant.

Where used, the **ORDER BY** clause must follow the **GROUP BY** clause. Ordering by aggregate functions is allowed.



The next example summarises the total quantity of orders by customer and sorts the output in descending sequence of total quantity:

```
SELECT    customer_nr, SUM ( quantity ) AS Total_Quantity
FROM      orders
GROUP BY  customer_nr
ORDER BY  Total_Quantity DESC
```

The ROLLUP option causes a grand total to be produced. An Order By clause should be included to avoid the Grand Total being displayed in the first row.

```
SELECT    department_nr, SUM ( salary )
FROM      employee
GROUP BY  ROLLUP (department_nr)
ORDER BY  department_nr
```

If grouping by more than one column, the ROLLUP option causes sub-totals and a grand total to be produced. Include an Order By clause in order to see employees of each department, a sub-total at each change in department, and a Grand Total on the final row of output.

```
SELECT    department_nr, job, SUM ( salary )
FROM      employee
GROUP BY  ROLLUP (department_nr, job)
ORDER BY  department_nr, job
```

## THE HAVING CLAUSE

The **HAVING** clause specifies which groups are to be selected for output. Groups are only output if they match the condition (or conditions) specified by the **HAVING** clause.

The following example lists all departments that have a total employee monthly salary outgoing that exceeds 10000:

```
SELECT    department_nr, SUM ( salary )  
FROM      employee  
GROUP BY  department_nr  
HAVING    SUM (salary) > 10000
```

The following example lists all departments, which have more than 4 employees:

```
SELECT    department_nr, COUNT (*)  
FROM      employee  
GROUP BY  department_nr  
HAVING    COUNT (*) > 4
```

This example lists all departments which employ two or more salesmen.

```
SELECT    department_nr, COUNT ( * ) AS Salesmen  
FROM      employee  
WHERE     job = 'Salesman'  
GROUP BY  department_nr  
HAVING    COUNT (*) >= 2
```

## WORKSHOP EXERCISES

Write SQL statements or scripts to answer the following questions.

1. How many rows are there in the prospect table?

2. How many employees have the surname 'Jones'?

3. Find the maximum, minimum and average salary. (This can be done in a single statement).

4. Count how many orders there are for each product code. (Use the ORDERS table)

5. As question 4, but only for product codes with more than 3 orders.

6. Find the sum of the salary for each job within each department.

*Optional additional questions (if time permits) on next page*

**OPTIONAL ADDITIONAL QUESTIONS (IF TIME PERMITS)**

7. Apart from orders for customers 1317 and 1223, list the average quantity grouped by product code and customer nr, as long as the average is less than 10.

8. Find the maximum, minimum and average salary of employees who are neither analysts nor the chairman, grouping this output by job and only including jobs where the average salary is between £1000 and £1500 inclusive. Order your data by minimum salary.

## SAMPLE ANSWERS

```
1.  SELECT COUNT (*)  
    FROM prospect
```

```
2.  SELECT COUNT (*)  
    FROM employee  
    WHERE surname = 'Jones'
```

```
3.  SELECT MAX (salary), MIN (salary), AVG (salary)  
    FROM employee
```

```
4.  SELECT product_code, COUNT (*)  
    FROM orders  
    GROUP BY product_code
```

```
5.  SELECT product_code, COUNT (*)  
    FROM orders  
    GROUP BY product_code  
    HAVING COUNT (*) > 3
```

```
6.  SELECT department_nr, job, SUM (salary)  
    FROM employee  
    GROUP BY department_nr, job
```

```
7.  SELECT product_code, customer_nr, AVG (quantity)
    FROM orders
   WHERE customer_nr NOT IN (1317, 1223)
  GROUP BY product_code, customer_nr
 HAVING AVG (quantity) < 10;
```

```
8.  SELECT job,
          AVG (salary) AS average,
          MAX (salary) AS maximum,
          MIN (salary) AS minimum
    FROM employee
   WHERE job NOT IN ('Analyst', 'Chairman')
  GROUP BY job
 HAVING AVG (salary) BETWEEN 1000 AND 1500
 ORDER BY MIN (salary)
```

## **Section 5**

# **JOINING TABLES**





## JOINING TABLES

In this section the following topics relating to Joining Tables will be covered:

- ◆ **Joining Tables**
- ◆ **Inner Joins**
- ◆ **Table Aliases**
- ◆ **Outer Joins**
- ◆ **Self Joins**
- ◆ **Additional ANSI Join syntax**
- ◆ **Traditional Join syntax**

## UNDERSTANDING JOINS

Data can be retrieved from more than one table by specifying the required tables in the **FROM** clause. Joining is commonly required either where selection criteria is held in a different table from the desired data or where information from more than one table needs to be viewed simultaneously.

The tables are joined by using the key word **JOIN** in the **FROM** clause and a condition to match the rows. This syntax is known as **ANSI join** syntax as it is compliant with the 1992 ANSI standard.

Typically the 'join' condition will use a primary key column in one of the tables and its matching foreign key column in the other table.

Tables can also be joined using the **WHERE** clause, and this is included at the end of the chapter for completeness as this type of syntax may still be encountered in existing SQL scripts.

There are three aspects to joining tables:

1. What data will be included:
  - ◆ **INNER JOINS** show only the data that can be matched across the tables
  - ◆ **OUTER JOINS** also include data from one of the tables with no match in the other, for example, departments without employees, customers that have placed no orders.
2. The operator used to match the rows:
  - ◆ **Equi-joins** are the normal type of join, an exact match between department numbers in the Department and Employee tables.
  - ◆ **Non-equijoins** use an operator such as **BETWEEN**, used to work out the employee's salary grade (example shown on Page 5-5).
3. The actual syntax used:
  - ◆ ANSI Join syntax which is standard across relational databases
  - ◆ "Traditional" join syntax, which was the only available option before the ANSI 1992 standard.

**Note:** a table can be joined to itself, normally where there is some sort of hierarchical relationship between the rows, such as in the Employee table, where the Manager column represents the employee\_nr of the person who manages this employee. Self-joins can use any type of join syntax and can be inner or outer joins.

**Note:** failure to specify a join condition results in a **cross join**, also known as the *Cartesian product*, which relates every row in the one table to every row in the other.

## CROSS JOINS

Usually tables are joined in a meaningful way to indicate their relationships with each other. If this is not done then the number of rows output is the product of all selected rows in all specified tables. Such a query is called a *Cross Join* and the result returned as the *Cartesian Product*. It is rare that such a query will be useful.

A Cross Join between two tables can be achieved using the following ANSI/ISO standard syntax:

```
SELECT *  
FROM customer  
CROSS JOIN orders
```

An alternative syntax for producing the Cartesian Product is shown below:

```
SELECT *  
FROM customer, orders
```

CROSS JOIN returns the Cartesian product of rows from tables in the join. In other words, it will produce rows which combine each row from the first table with each row from the second table.

## USING INNER JOINS

The basic ANSI standard syntax is:

```
FROM table-a [INNER] JOIN table-b ON condition
```

In the example below, the tables are joined by specifying an equality condition between the primary key of the customer table (*customer\_nr*) and the related foreign key in the orders table (also *customer\_nr*).

As the names of both join columns are identical they must be qualified in the SELECT statement by prefixing them with the name of the table and a dot.

```
SELECT surname, initials, order_date, product_code, quantity  
FROM    customer  
      INNER JOIN orders  
      ON customer.customer_nr = orders.customer_nr
```

Note that the keyword **INNER** is optional and that the condition(s) goes after the join.

In the following example, the column *customer\_nr* has been added to the select list and so also needs to be qualified with a table name, either *customer* or *orders*.

```
SELECT customer.customer_nr, surname, initials, order_date,  
        product_code, quantity  
FROM    customer  
      JOIN orders  
      ON customer.customer_nr = orders.customer_nr
```

The following example displays an employee name and department if an employee earns more than 2000.

```
SELECT employee.surname, employee.initials, department.name  
FROM    employee  
      JOIN department  
      ON employee.department_nr = department.department_nr  
WHERE employee.salary > 2000
```

The next example shows a join between three tables, the *customer* table, the *orders* table and the *product* table. It does not matter which table you start with, but following tables must be able to be joined to those specified before. It displays which customers placed orders for which products

```
SELECT customer.surname, customer.initials, orders.order_date,
        product.description, orders.quantity * product.sales_price AS value
FROM   customer
JOIN   orders
ON     customer.customer_nr = orders.customer_nr
JOIN   product
ON     orders.product_code = product.product_code
ORDER BY customer.surname, customer.initials, product.description
```

Qualification of all selected columns will prevent the query failing in the future if a change to a table adds a column that exists in another table used in the query. Using table aliases (as described on the next page) simplifies column qualification.

All the above are examples of equijoins.

*Non-equijoins* are table joins in which some operator other than the *equals* operator is used. In the example below a join is engineered between the employee and salary grade tables by means of the **BETWEEN** operator:

This statement displays which employees are on which grade.

```
SELECT salgrade.grade, employee.surname, employee.initials,
        employee.salary
FROM   employee
JOIN   salgrade
ON     employee.salary BETWEEN salgrade.low_salary
        AND salgrade.high_salary
ORDER BY 1, 2, 3
```

## USING TABLE ALIASES

Tables, like columns, may also be given aliases; these are defined after each table name specified in the **FROM** clause (the keyword **AS** can be optionally included). Table aliases are particularly useful where qualification of columns is necessary, as they simplify readability and reduce the amount of typing.

You must use table aliases when joining a table to itself to be able to distinguish between the different instances of the table.

```
SELECT c.customer_nr, surname, initials, order_date,  
        product_code, quantity  
FROM   customer c  
        JOIN orders o  
        ON   c.customer_nr = o.customer_nr
```

In this example, all columns are qualified with a one-letter alias. It displays customers that have placed orders for products since the beginning of January 2017:

```
SELECT c.surname, c.initials, o.order_nr, o.order_date, p.product_code,  
        p.description, o.quantity * p.sales_price AS value  
FROM   customer c  
        JOIN orders o  
        ON   c.customer_nr = o.customer_nr  
        JOIN product p  
        ON   o.product_code = p.product_code  
WHERE o.order_date >= '2017-01-01'  
ORDER BY c.surname, c.initials, p.description
```

## USING OUTER JOINS

Sometimes it is necessary to retrieve data from **all** rows of one table even if some of its rows have no associated data in a related table. This is achieved by means of an **outer join**.

An outer join can be a

- ◆ Left outer join
- ◆ Right outer join
- ◆ Full outer join

Left and right outer joins are determined simply by the order in which the tables are specified. A full outer join includes unmatched rows from both tables.

The basic syntax is:

```
FROM table-a LEFT | RIGHT | FULL [OUTER] JOIN table-b ON condition
```

## LEFT OUTER JOINS

In a left outer join, the data includes any unmatched data from the table to the left of the **JOIN** clause. The keyword **OUTER** is optional as the keyword **LEFT** indicates an outer join.

### Example

These statements displays all departments and the employees within them, *but includes any departments that have no employees*.

```
SELECT d.*, e.employee_nr, e.surname, e.job, e.salary
FROM   department d
LEFT OUTER JOIN employee e
ON     d.department_nr = e.department_nr
```

```
SELECT d.*, e.employee_nr, e.surname, e.job, e.salary
FROM   department d
LEFT JOIN employee e
ON     d.department_nr = e.department_nr
```

## RIGHT OUTER JOINS

In a right outer join the data includes any unmatched rows from the table to the right of the **JOIN** clause. The keyword **OUTER** is likewise optional.

### Example

These statements are the same as the previous examples, but the order of the tables is different:

```
SELECT d.*, e.employee_nr, e.surname, e.job, e.salary
FROM   employee e
      RIGHT OUTER JOIN department d
      ON d.department_nr = e.department_nr
```

```
SELECT d.*, e.employee_nr, e.surname, e.job, e.salary
FROM   employee e
      RIGHT JOIN department d
      ON d.department_nr = e.department_nr
```

## FULL OUTER JOINS

In a full outer join, both matched and unmatched data is returned from tables on both sides of the **JOIN** clause.

This statement displays all departments and the employees within them, *but includes any departments that have no employees and any employees who do not have a department.*

```
SELECT *
FROM   department d FULL OUTER JOIN employee e
      ON d.department_nr = e.department_nr
```



## USING SELF JOINS

Occasionally it is necessary to retrieve data from a single table based upon comparing the value of a column in one row of the table with another column in a different row of the same table. This is achieved by joining the table to itself.

As this means that the table must be specified twice in the **FROM** clause, the use of table aliases is mandatory. These queries are easier to understand if meaningful table aliases are used.

The example below retrieves data from the *employee* table which includes the name of each employee's manager. Effectively the table is used twice, once for the employee data using the alias **e** and a second time for the name of the manager using the alias **m**.

In order to do this it is necessary to compare the value of an employee's *manager* column with the value of the *employee\_nr* column in another row of the table and retrieve the employee name (i.e. the name of the manager) where a match is found:

```
SELECT  e.employee_nr, e.surname, e.initials, e.job,
        m.surname AS manager_name
FROM    employee e
JOIN    employee m
ON      e.manager = m.employee_nr
ORDER BY e.surname, e.initials
```

If the managing director or any other employee has no manager (in other words, if the manager column is **NULL**) then no information for that employee will be output. If such an employee's details are required, then an outer join can be specified as follows. A **LEFT OUTER JOIN** is needed as we need the extra employee (the Chairman) without a manager:

```
SELECT  e.employee_nr, e.surname, e.initials, e.job,
        m.surname AS manager_name
FROM    employee e
LEFT OUTER JOIN employee m
ON      e.manager = m.employee_nr
ORDER BY e.surname, e.initials
```

## ADDITIONAL ANSI JOIN SYNTAX

Where the names of the join columns are identical in the tables being joined then the **JOIN .. USING** syntax may be used as shown here:

```
SELECT customer_nr, c.surname, o.order_nr
FROM   customer c
      JOIN orders o USING (customer_nr);
```

Note that the join column or columns (in this case *customer\_nr*) must *not* be qualified with a table name if this method is used.

The **NATURAL JOIN** syntax joins tables *automatically* on *all* columns which have a common name and datatype. Using this method, if supported, then the above statement may be written as:

```
SELECT customer_nr, c.surname, o.order_nr
FROM   customer c
      NATURAL JOIN orders o;
```

Once again, the join column or columns must not be qualified with a table name.

The NATURAL JOIN method can cause problems:

- ◆ Firstly, there may be columns with the same name in the two tables, but in each table the column name refers to something different.
- ◆ Secondly, if there are no columns with the same name in the two tables, then a Cartesian product will be performed.

## TRADITIONAL JOIN SYNTAX

An alternative join method just lists the tables to be joined in the **FROM** clause and uses **WHERE** clause conditions to match the rows between the tables. A disadvantage of this type of join is that the **WHERE** clause is used both to join the tables and to limit the data to be retrieved.

### Examples

This statement displays the customers together with details of their orders.

```
SELECT surname, initials, order_date, product_code, quantity
FROM    customer, orders
WHERE   customer.customer_nr = orders.customer_nr
```

This statement displays the customers together with details of their orders and the product details.

```
SELECT c.surname,c.initials,p.description, o.order_date, o.quantity
FROM    customer c, orders o, product p
WHERE   c.customer_nr = o.customer_nr
AND    o.product_code = p.product_code
```

This statement displays the customers together with details of orders with a quantity of at least 100. The **WHERE** clause is used both for the join condition and for the condition on the quantity column.

```
SELECT surname, initials, order_date, product_code, quantity
FROM    customer c, orders o
WHERE   c.customer_nr = o.customer_nr
AND    o.quantity >= 100
```

The order in which the joins are specified in a multi-table join of more than two tables is important. The joins are effectively carried out from left to right.



## WORKSHOP EXERCISES

Write SQL statements or scripts to answer the following questions.

1. List the employee number, surname, job, department name and department location for each employee. Order the results by location.

2. List the product code, description, sales price and name of supplier for all products. Order the output by name of supplier.

3. List details of those Products supplied by 'London Computers Ltd.' (Don't forget the dot on the end).

4. List details of all suppliers and the products they supply (if any).

5. List the customer\_nr, credit\_limit, order\_nr, product\_code, supplier\_nr and name of supplier for all customers with a credit\_limit greater than 1000

## OPTIONAL ADDITIONAL QUESTIONS (IF TIME PERMITS)

6. List those customers who have placed orders which, **in total** exceed a value of £40,000. ( "Value" is quantity \* sales\_price )

7. By selling all products currently in stock what profit could be made? Group the output by supplier name.

( more questions on the next page )

8. Display customer surname and the number of orders they have placed (customers with no orders should display a count of 0).

9. Display supplier name, product description and total sales generated.

THIS PAGE HAS BEEN INTENTIONALLY LEFT BLANK

## SAMPLE ANSWERS

```
1a.  SELECT e.employee_nr, e.surname, e.job, d.name, d.location
      FROM employee e JOIN department d
      ON e.department_nr = d.department_nr
      ORDER BY d.location
```

OR

```
1b.  SELECT e.employee_nr, e.surname, e.job, d.name, d.location
      FROM employee e, department d
      WHERE e.department_nr = d.department_nr
      ORDER BY d.location
```

```
2a.  SELECT p.product_code, p.description, p.sales_price, s.name
      FROM product p JOIN supplier s
      ON p.supplier_nr = s.supplier_nr
      ORDER BY s.name
```

OR

```
2b.  SELECT p.product_code, p.description, p.sales_price, s.name
      FROM product p, supplier s
      WHERE p.supplier_nr = s.supplier_nr
      ORDER BY s.name
```

```
3a.  SELECT p.description, p.cost_price, p.sales_price, p.instock
      FROM product p JOIN supplier s
      ON p.supplier_nr = s.supplier_nr
      WHERE s.name = 'London Computers Ltd.'
      ORDER BY p.description
```

OR

```
3b.  SELECT p.description, p.cost_price, p.sales_price, p.instock
      FROM product p, supplier s
      WHERE p.supplier_nr = s.supplier_nr
        AND s.name = 'London Computers Ltd.'
      ORDER BY p.description
```



```
4.  SELECT s.name AS Supplier, p.description AS Product
    FROM supplier s LEFT OUTER JOIN product p
    ON s.supplier_nr = p.supplier_nr
```

```
5a. SELECT c.customer_nr, c.credit_limit, o.order_nr, p.product_code, s.supplier_nr,
        s.name
    FROM customer c
        JOIN orders o ON c.customer_nr = o.customer_nr
        JOIN product p ON o.product_code = p.product_code
        JOIN supplier s ON p.supplier_nr = s.supplier_nr
    WHERE c.credit_limit > 1000
```

OR

```
5b. SELECT c.customer_nr, c.credit_limit, o.order_nr, p.product_code, s.supplier_nr,
        s.name
    FROM customer c, orders o, product p, supplier s
    WHERE c.customer_nr = o.customer_nr
        AND o.product_code = p.product_code
        AND p.supplier_nr = s.supplier_nr
        AND c.credit_limit > 1000
```

```
6.  SELECT c.surname, SUM( o.quantity * p.sales_price ) as Total_Sales
    FROM customer c JOIN orders o ON c.customer_nr = o.customer_nr
        JOIN product p ON p.product_code = o.product_code
    GROUP BY c.surname
    HAVING SUM ( o.quantity * p.sales_price ) > 40000;
```

```
7.  SELECT s.name, SUM( (p.sales_price - p.cost_price) * p.instock ) AS Profit
    FROM product p JOIN supplier s ON s.supplier_nr = p.supplier_nr
    GROUP BY s.name;
```

```
8.  SELECT c.surname, COUNT(o.order_nr) AS orders_placed
    FROM customer c LEFT JOIN orders o ON c.customer_nr = o.customer_nr
    GROUP BY c.surname, c.customer_nr ;
```

```
9.  SELECT s.name, p.description, SUM( o.quantity*p.sales_price) AS Total_Sales
    FROM orders o JOIN product p ON o.product_code=p.product_code
        JOIN supplier s ON s.supplier_nr = p.supplier_nr
    GROUP BY s.name, p.description
```

## **Section 6**

# **BASIC SUBQUERIES AND SET OPERATORS**



## BASIC SUBQUERIES AND SET OPERATORS

In this section the following topics will be covered:

- ◆ **Using Basic Subqueries**
- ◆ **Union**
- ◆ **Intersect**
- ◆ **Except**
- ◆ **Limit and Offset**

## OVERVIEW OF SUBQUERIES

More complex queries can be constructed using subqueries. A subquery is a **SELECT** statement enclosed within parentheses as part of a **WHERE**, **HAVING** or **FROM** clause of an outer (or parent) **SELECT** statement.

### Examples

This query finds the employee(s) earning the most. You cannot mix aggregate functions with ordinary data or use them in the **WHERE** clause. The query selects the maximum salary first and uses the resulting value in the **WHERE** clause:

```
SELECT *  
FROM employee  
WHERE salary = ( SELECT MAX (salary)  
                FROM employee )
```

This query is similar, finding the employees earning more than the average:

```
SELECT surname, initials, job, salary  
FROM employee  
WHERE salary > ( SELECT AVG ( salary )  
                FROM employee )  
ORDER BY salary DESC
```

The subquery is performed first, and its value or values are used in the main query. *Single-row subqueries* return one row to the parent query. *Multiple-row subqueries* return more than one row, in which case the parent query's **WHERE** clause must use an operator appropriate for multiple values, such as **IN**. A query can only be guaranteed not to return more than one row if it uses aggregate functions without **Group by** or its **WHERE** clause uses the table's primary key column.

Subqueries can be nested - in other words a subquery may itself contain a subquery. In the case of nested subqueries, the innermost subquery will be evaluated first. There is no stated limit on nesting subqueries, however using the **WITH** clause (which is covered in Section 9: *Complex Subqueries*) is likely to be easier to read and maintain, and may provide better performance.

## USING A SUBQUERY AS AN ALTERNATIVE TO JOIN

In most cases a join will perform better than a subquery but occasionally a subquery may be easier to construct or read. For example:

```
SELECT *  
FROM orders  
WHERE customer_nr IN ( SELECT customer_nr  
                        FROM customer  
                        WHERE town = 'London' )
```

In the above example an **IN** must be used rather than an equals (=) symbol since there is potentially more than one row in the result set returned by the subquery.

To change the above example to find all orders placed by customers not based in London, the operator **NOT IN** is used:

```
SELECT *  
FROM orders  
WHERE customer_nr NOT IN ( SELECT customer_nr  
                           FROM customer  
                           WHERE town = 'London' )
```

The above query (which returns values that do *not* match) is referred to as an *ANTI-JOIN*.

The following query likewise finds details of customers who have not placed orders:

```
SELECT *  
FROM customer  
WHERE customer_nr NOT IN ( SELECT customer_nr  
                           FROM orders )
```

More complex subqueries are covered later in the course (Section 9).

## SET OPERATORS

It is possible to combine information of a similar type from more than one query. A classic example is the generation of a mailing list from two or more customer or prospect tables.

PostgreSQL supports the **UNION**, **INTERSECT** and **EXCEPT** operators all of which combine tables in different ways. In all cases the following two rules apply:

- ◆ The same number of items must be selected from each query.
- ◆ The items selected must be of the same data type and should be logically related. (However, they do not have to be the same length or the same column name.)

### THE UNION OPERATOR

The **UNION** operator is used to combine rows from two or more queries whilst at the same time avoiding the output of duplicate rows. The output from a **UNION** is not sorted in PostgreSQL so include an ORDER BY clause to display the output in the desired sort order.

The example below retrieves all the unique combinations of name and address from separate queries using the customer and prospect tables; no duplicate rows are shown and column headings are obtained from the first query:

```
SELECT surname, initials, street, town, county, post_code AS postcode
FROM    prospect
UNION
SELECT surname, initials, street, town, county, post_code
FROM    customer
ORDER BY surname
```

If the operator **UNION ALL** is used, duplicate rows *are* returned and again the data is not sorted. It is commonly used to combine output from two or more queries where it is known that there are no duplicate rows or the presence of duplicate rows is irrelevant. It is more efficient as it does not have to perform a sort, which may be of relevance if large volumes of data are involved.



## THE INTERSECT OPERATOR

The **INTERSECT** operator is used to combine rows from two or more queries and return only those rows which appear in all the queries. The output from **INTERSECT** is likewise sorted as a side-effect of combining the rows.

The following example retrieves the names of all employees who are also customers, sorting them in order of surname and initials.

```
SELECT surname, initials FROM employee
INTERSECT
SELECT surname, initials FROM customer
ORDER BY surname, initials
```

## THE EXCEPT OPERATOR

The **EXCEPT** operator combines two queries and returns only rows from the first query **not** returned by the second query.

The example below retrieves details of prospects (from a prospect table) who are not yet customers and sorts them in telephone number order using **ORDER BY**.

```
SELECT surname, initials, telephone
FROM prospect
EXCEPT
SELECT surname, initials, telephone
FROM customer
ORDER BY telephone
```

### Note

If the column *names* differ the output uses the column names from the first query. If column aliases are needed they should be given to the first query and can, of course, be used in the **ORDER BY** clause. Alternatively the **ORDER BY** clause can specify the ordinal positions of the required sort columns. Each query can have its own **ORDER BY** provided you parenthesize each query.

## LIMIT AND **OFFSET** CLAUSES

You can use **LIMIT n** to fetch n number of rows from a query. Ensure you include an ORDER BY clause to avoid selecting an indeterminate collection of rows. LIMIT with a NULL value specified, or **LIMIT ALL** both have the same effect as omitting the LIMIT clause. LIMIT with a numeric value will fetch no more than that number of rows but could fetch fewer if there are fewer rows of results.

PostgreSQL also supports the ANSI standard **FETCH FIRST|NEXT** clauses to limit the number of rows returned. If a numeric value is not specified, it defaults to 1 row. The FIRST and NEXT keywords are synonyms, provided to allow semantic clarity. From PostgreSQL version 13, the **WITH TIES** option can be used instead of **ONLY** to return any additional rows that tie for the last place in the result set according to the ORDER BY clause.

The examples below show both syntaxes to fetch the first four rows starting with the highest paid employee:

```
SELECT employee_nr, surname, salary
FROM employee
ORDER BY salary DESC
LIMIT 4;
```

```
SELECT employee_nr, surname, salary
FROM employee
ORDER BY salary DESC
FETCH FIRST 4 ROWS ONLY;
```

The LIMIT and FETCH FIRST|NEXT clauses can be used in conjunction with **OFFSET n** to skip a desired number of rows first. This query with OFFSET will skip the first three rows and fetch four rows commencing with row 4:

```
SELECT employee_nr, surname, salary
FROM employee
ORDER BY salary DESC
LIMIT 4 OFFSET 3;
```

```
SELECT employee_nr, surname, salary
FROM employee
ORDER BY salary DESC
FETCH FIRST 4 ROWS ONLY OFFSET 3;
```

The LIMIT, FETCH FIRST|NEXT and OFFSET clauses can be used in conjunction with the SET operators as shown here. Note that each of the two queries has been parenthesized. In the example no more than six rows will be returned in total, however, with the data in the training database only four rows will be returned due to the removal of duplicates in the UNION.

```
(SELECT surname, initials, street, town, county, post_code AS postcode  
FROM prospect LIMIT 3 )  
UNION  
(SELECT surname, initials, street, town, county, post_code  
FROM customer LIMIT 3)
```



## WORKSHOP EXERCISES

Write SQL statements using subqueries, etc., to answer the following questions.

1. List details of the employee(s) with the lowest salary.

2. List orders where the quantity is greater than the average quantity of all orders.

3. List details of any suppliers who do not supply any products

4. List details of products not supplied by a London based company.

5. List details of all orders placed for the product 'Graphix Draw 2.0'.

6. List surnames, initials and postcodes from both the customer and prospect tables as a single integrated vertical list without displaying duplicates.

7. As question 6 but do not eliminate duplicates.

8. List the surnames which appear in both the employee and customer tables.

9. List the surnames which appear in the customer table but do not appear in the employee table.



## SAMPLE ANSWERS

```
1.  SELECT *  
    FROM employee  
    WHERE salary = ( SELECT MIN (salary)  
                     FROM   employee )
```

```
2   SELECT *  
    FROM orders  
    WHERE quantity > ( SELECT AVG (quantity)  
                       FROM   orders )
```

```
3.  SELECT *  
    FROM supplier  
    WHERE supplier_nr NOT IN ( SELECT supplier_nr  
                               FROM   product )
```

```
4.  SELECT product_code, description, sales_price, cost_price, instock  
    FROM product  
    WHERE supplier_nr IN ( SELECT supplier_nr  
                           FROM   supplier  
                           WHERE  town != 'London' )
```

```
5.  SELECT order_nr, customer_nr, product_code, quantity  
    FROM orders  
    WHERE product_code IN ( SELECT product_code  
                             FROM   product  
                             WHERE  description = 'Graphix Draw 2.0' )
```

```
6.  SELECT surname, initials, post_code
    FROM customer
    UNION
    SELECT surname, initials, post_code
    FROM prospect
    ORDER BY surname
```

```
7.  SELECT surname, initials, post_code
    FROM customer
    UNION ALL
    SELECT surname, initials, post_code
    FROM prospect
    ORDER BY surname
```

```
8.  SELECT surname
    FROM customer
    INTERSECT
    SELECT surname
    FROM employee
```

```
9.  SELECT surname
    FROM customer
    EXCEPT
    SELECT surname
    FROM employee
```



## **Section 7**

### **NUMERIC, CHARACTER AND DATE FUNCTIONS**



## NUMERIC, CHARACTER AND DATE FUNCTIONS

In this section the following topics relating to Functions will be covered:

- ♦ **Function Types**
- ♦ **Testing Functions**
- ♦ **Numeric Functions**
- ♦ **Character Functions**
- ♦ **Date and Arithmetic Functions**

## OVERVIEW OF FUNCTIONS

### FUNCTION TYPES

PostgreSQL provides a number of *single row* functions in addition to the standard aggregate functions described in Section 4. These may be categorised according to the type of data they operate on. The main categories are:

- ◆ Numeric functions.
- ◆ Character functions.
- ◆ Regular expression functions
- ◆ Date functions.
- ◆ Conversion functions (which allow various conversions between data types).

There are also a few other miscellaneous functions which do not readily fall into any of the above categories. This section does not list all the additional SQL functions, but describes the most useful and commonly used functions in each category.

### TESTING FUNCTIONS

If you simply wish to test a function in isolation from actual table data, you can do so by using a **SELECT** clause alone as these three examples show:

```
SELECT 'Hello World'
```

Performing a calculation:

```
SELECT SQRT ( 52 / 5 )
```

Testing the operation of a function:

```
SELECT TO_CHAR ( CURRENT_DATE, 'YYYY' )
```

## NUMERIC FUNCTIONS

Some of the more useful functions which operate on numbers, numeric expressions or number columns (indicated below by the characters *n*, *n1* and *n2*) include:

<b>ABS ( <i>n</i> )</b>	Returns the absolute value of <i>n</i> , i.e. ignoring the sign. @ can be used instead.
<b>MOD ( <i>n1</i>, <i>n2</i> )</b>	Returns the remainder resulting from dividing <i>n1</i> by <i>n2</i> . The arithmetical operator % can be used instead.
<b>POWER ( <i>n1</i>, <i>n2</i> )</b>	Returns <i>n1</i> raised to the power of <i>n2</i>
<b>ROUND ( <i>n1</i> [, <i>n2</i>] )</b>	Rounds <i>n1</i> to <i>n2</i> decimal places or 0 decimal places if <i>n2</i> is omitted
<b>FLOOR(<i>n</i>)</b>	Rounds <i>n</i> to the nearest integer down.
<b>CEIL(<i>n</i>)</b>	Rounds <i>n</i> to the nearest integer up. Or use <b>CEILING(<i>n</i>)</b>
<b>SIGN ( <i>n</i> )</b>	Returns 1 if <i>n</i> is positive, -1 if <i>n</i> is negative and 0 if <i>n</i> is zero
<b>SQRT ( <i>n</i> )</b>	Returns the square root of <i>n</i>
<b>TRUNC ( <i>n1</i> [, <i>n2</i>] )</b>	Truncates <i>n1</i> to <i>n2</i> decimal places or 0 decimal places if <i>n2</i> is omitted

Also many mathematical functions such as **SIN**, **COS**, **TAN**, **EXP**, **LN**, **LOG**

### Examples

```
SELECT POWER ( 3, 3 )
```

(Result = 27)

```
SELECT MOD ( 35, 8 )
Or:
SELECT 35 % 8
```

(Result = 3)

```
SELECT TRUNC ( ABS ( -58.00 / 6 ), 2 )
```

(Result = 9.66)

```
SELECT SIGN ( 100 - 150 )
```

(Result = -1)

The example below lists details of employees whose employee number is an even number:

```
SELECT employee_nr, surname, initials  
FROM employee  
WHERE MOD ( employee_nr, 2 ) = 0
```

The example here shows employees' weekly salaries with and without rounding and truncating to the nearest penny:

```
SELECT surname,  
        ( salary * 12 / 52 ) AS Weekly_Salary,  
        ROUND ( salary * 12 / 52, 2 ) AS Rounded,  
        TRUNC ( salary * 12 / 52, 2 ) AS Truncated  
FROM employee
```

The following example lists orders and their values rounded to the nearest £10.

```
SELECT o.order_nr, ROUND ( o.quantity * p.sales_price, -1 ) AS value  
FROM orders o JOIN product p  
ON o.product_code = p.product_code
```

## CHARACTER FUNCTIONS

Some of the more useful functions which operate on character (string) expressions and columns (indicated below by the characters **s**, **s1**, **s2** and **s3**) include:

<b>CONCAT(s1, s2..sn)</b>	Concatenate the text representations of all the arguments. NULL arguments are ignored.
<b>CONCAT_WS(s0, s1, s2.. sn)</b>	Concatenate all but the first argument with separators. The first argument (s0) is used as the separator string. NULL arguments are ignored.
<b>INITCAP ( col )</b>	Capitalises the first character of each word in <b>col</b>
<b>LENGTH ( string )</b>	Returns the length of <b>string</b> or column. You can also use <b>CHAR_LENGTH</b> (string).
<b>LOWER ( col )</b>	Returns value of <b>col</b> converted to lowercase
<b>LPAD ( col, width [, pad-char ] )</b>	Returns value of <b>col</b> padded to the left to the specified <b>width</b> with spaces or specified <b>pad-char</b> . The value will be truncated if the width is too short.
<b>LTRIM (col [,char])</b>	Trim character <b>char</b> from the start of string <b>col</b> or where <b>char</b> is omitted, trim <b>leading</b> spaces.
<b>POSITION ( string IN col )</b>	Returns the position of the start of the <b>string</b> found in the column <b>col</b> starting the search at the beginning looking for the first <b>occurrence</b> of the string. Returns zero if not found.
<b>REPLACE ( col, oldstring , newstring )</b>	Returns the value in the <b>column</b> with all occurrences of the <b>oldstring</b> replaced with the <b>newstring</b> .
<b>RPAD ( col, width [, pad-char ] )</b>	Works in the same way as <b>LPAD</b> but adds characters to the end up to the specified width.
<b>RTRIM (col [,char])</b>	Trim character <b>char</b> from the string <b>col</b> or where <b>char</b> is omitted, trim <b>trailing</b> spaces.
<b>SUBSTRING ( col from start-place [ for length ] )</b>	Returns the part of the value in the <b>column</b> which starts at position of the <b>start-place</b> and is <b>length</b> characters long. If <b>length</b> is omitted it returns the rest of the value

<b>TRANSLATE ( <i>col</i>, <i>charlist1</i>, <i>charlist2</i> )</b>	Returns the value of the <b>column</b> with each character in <b>charlist1</b> replaced with its equivalent in <b>charlist2</b> according to its position. If <b>charlist1</b> has more characters than <b>charlist2</b> those with no equivalent will be omitted. If any argument is null it returns null.
<b>TRIM (</b> <b>[leading   trailing   both]</b> <b>[<i>characters</i>]</b> <b>from <i>col</i>)</b>	The argument can be <b>LEADING</b> , <b>TRAILING</b> or <b>BOTH</b> . It returns the value of <b>col</b> with all occurrences of <b>character</b> removed from the beginning, end or both according to the chosen argument which defaults to <b>BOTH</b> . If <b>chars</b> is omitted it defaults to spaces.
<b>UPPER ( <i>col</i> )</b>	Returns value in <b>column</b> in uppercase



## STRING CONCATENATION

PostgreSQL concatenates (joins) string values using the `||` operator (two vertical bars). You can also concatenate non-strings this way, e.g.; numbers, provided at least one string is included.

### Concatenation Examples

```
SELECT 'ABC' || 'DEF'
```

(Result = ABCDEF)

```
SELECT initials || surname AS name FROM employee
```

Initials and surname are joined as a single column so the alias goes at the end. PostgreSQL does not automatically space-pad short values for CHAR datatype columns.

```
SELECT surname || ' ' || salary AS "Management"  
FROM employee  
WHERE job = 'Manager';
```

In the above example a non-string – salary – is concatenated with two strings.

Alternatively you can use **CONCAT** to concatenate multiple comma-separated arguments (data type does not matter); NULL values will be ignored.

```
SELECT surname, CONCAT( street, ' ', town, ' ', county)  
AS "Address"  
FROM prospect
```

A further option is **CONCAT\_WS** where the first argument is a character to insert between each of the subsequent, comma-separated, arguments:

```
SELECT surname, CONCAT_WS( ' ', street, town, county)  
AS "Address"  
FROM prospect
```

## CHARACTER FUNCTION EXAMPLES

Find all prospective customers with the surname Jones regardless of case.

```
SELECT prospect_nr, initials, UPPER ( surname ), LOWER ( post_code )
FROM    prospect
WHERE INITCAP ( surname ) = 'Jones'
```

Identify positions of the first space and full stop in product descriptions.

```
SELECT product_code, description, POSITION ( ' ' IN description) AS space,
POSITION ( '.' IN description) AS position_of_dot
FROM    product
```

Find the products with full stops in product descriptions. Zero means no full stop.

```
SELECT * FROM product
WHERE POSITION ( '.' IN description) > 0
```

This example formats surname and initials with hyphens and jobs with '\*' and concatenates the result into a single column. See Section 9 page 11 for the **COALESCE** function

```
SELECT RPAD ( surname, 20, '-' ) ||
RPAD ( COALESCE ( initials, 'n/a'), 10, '-' ) ||
      LPAD ( job, 9, '*' ) AS formatted_output
FROM    employee;
```

	formatted_output text
1	King-----J.R.-----*Chairman
2	Blake-----S.T.-----**Manager

This discards all occurrences of characters H,e and l from the start of a string.

```
SELECT TRIM ( LEADING 'Hel' FROM 'Hello World' );
```

(Result = o World)

This example produces the string 'Hello World' with leading '\*', trailing '\*' and both removed respectively:

```
SELECT TRIM ( LEADING '*' FROM '***** Hello World *****' ) as trailing,  
TRIM ( TRAILING '*' FROM '***** Hello World *****' ) as leading,  
TRIM ( '*' FROM '***** Hello World *****' ) as both;
```

This one obtains the product category from the first two letters of the product code:

```
SELECT SUBSTRING (product_code FROM 1 FOR 2) AS category,  
       product_code, description  
FROM   product
```

CATEGORY	PRODUCT_CODE	DESCRIPTION
AM	AM17	Activan 17" Monitor
AS	ASM2	Activan SuperMouse 2
DS	DS96	DataStore DBMS 96

This example includes salaries padded to the left with asterisks (as if for a printed cheque) for employees whose job starts with the letter "c" in any case. Note that it has been necessary to convert salary which is numeric to a string using the **CAST** function ( see Section 8 page 7):

```
SELECT UPPER( surname ) AS name, LPAD ( CAST(salary as varchar), 8, '*' )  
AS salary  
FROM   employee  
WHERE  LOWER ( job ) LIKE 'c%'
```

This example shows the length of the initials column and sorts by length of surname:

```
SELECT surname, initials, LENGTH ( initials ) AS length
FROM employee
ORDER BY LENGTH(surname) DESC
```

This example concatenates customers' initials and surname with a space between them:

```
SELECT RTRIM ( initials ) || ' ' || surname AS Customer_Name
FROM customer
```

The following example concatenates employees' first initial only and surname with a dot between the initial and the surname:

```
SELECT SUBSTRING ( initials FROM 1 FOR 1 ) || '.' || surname AS name,
job, salary
FROM employee
WHERE department_nr = 30
```

This example shows product descriptions from the *product* table with all occurrences of the word "Monitor" replaced with the word "Screen".

```
SELECT product_code, description AS Old_Description,
REPLACE ( description, 'Monitor', 'Screen' ) AS New_Description
FROM product
WHERE description LIKE '%Monitor%'
```

This example replaces the spaces in telephone numbers with hyphens and removes the commas from customers' addresses (there is no match for the comma):

```
SELECT customer_nr, surname, initials,
TRANSLATE ( street, 'x,', 'x' ) AS street, town, post_code,
TRANSLATE ( telephone, ',', '-') AS telephone
FROM customer
```

The example here retrieves names and postcodes from the *customer* table. It also finds just the first part of the postcode to give the postal district:

```
SELECT RTRIM ( initials ) || surname AS name, post_code AS postcode,  
SUBSTRING ( post_code FROM 1 FOR POSITION ( ' ' IN post_code ) - 1 ) AS  
area  
FROM    customer  
ORDER BY area
```

This is processed as follows:

- Step 1 Find out where a space occurs in the post\_code field using **POSITION**.
- Step 2 Subtract 1 from the number generated in step 1.
- Step 3 Extract the portion of the post\_code field starting from position 1 for the number of characters calculated in step 2.

## DATE ARITHMETIC AND FUNCTIONS

### DATE AND TIME COLUMN TYPES

These are the common data types involving date and time:

<b>TIME [ WITH TIME ZONE]</b>	Time in form 'HH:MI:SS.nnnnnnn'
<b>DATE</b>	Date in form 'YYYY-MM-DD'
<b>TIMESTAMP [ WITH TIME ZONE]</b>	Date and time in form 'YYYY-MM-DD HH:MI:SS.nnnnnnn'

### DATE AND TIME FORMATS

As well as the formats specified above, other delimiters can be used or no delimiters at all. The following three formats are all acceptable:

```
SELECT * FROM EMPLOYEE  
WHERE start_date > '2017-01-16'
```

```
SELECT * FROM EMPLOYEE  
WHERE start_date > '2017/01/16'
```

```
SELECT * FROM EMPLOYEE  
WHERE start_date > '20170116'
```

## DATESTYLE

Defaults for the display of dates and times can be set in the **postgresql.conf** configuration file of which an extract is shown here:

```
datestyle = 'iso, dmy'  
#intervalstyle = 'postgres'  
timezone = 'Europe/London'  
#timezone_abbreviations = 'Default'
```

A user could do the same for a session by using the **SET DATESTYLE** command. The *datestyle* parameter consists of two components to set the display format for date and time values, as well as the rules for interpreting ambiguous date input values.

There are only four output format date styles available of which **ISO** ('YYYY-MM-DD HH:MI:SS') is the default, and is the style required by the ANSI SQL standard. Alternative output formats can be achieved using functions, such as DATE\_PART() or EXTRACT(), which are covered later in this section, or TO\_CHAR() (see Section 8).

Date and time input is accepted in almost any reasonable format. The date ordering, **DMY**, **MDY** or **YMD**, is used to interpret ambiguous input of day, month, and year in the date input. The default is dependent on the locale when the database server is created.

## FUNCTIONS TO RETURN DATE TIME

The current date and time can be retrieved using the following functions:

<b>CURRENT_DATE</b>	Returns the date in the format yyyy-mm-dd
<b>CURRENT_TIME</b>	Returns the current time with fractional seconds and timezone offset in the format hh:mm:ss.ffffff tzh:tzm
<b>CURRENT_TIMESTAMP</b> <b>Or: NOW()</b>	Returns date and time to fractional seconds with timezone offset

Examples:

```
SELECT CURRENT_DATE
```

(Result = 2018-01-09 )

```
SELECT CURRENT_TIME
```

(Result = 15:48:32.137432 +00:00)

**LOCALTIME** does the same but without the Timezone.

```
SELECT CURRENT_TIMESTAMP
```

(Result = 2018-01-09 15:48:32.137432 +00:00)

**LOCALTIMESTAMP** does the same but without the Timezone.



## FUNCTIONS TO EXTRACT COMPONENTS FROM DATE TIME

These are the main functions to extract date and time components, *d* is the date and time in each case. Note that unit of time is quoted for **DATE\_PART** but not for **EXTRACT**.

<b>DATE_PART</b> ( ' <i>month</i> ' , <i>d</i> )	Returns the month of the year (January is 1)
<b>DATE_PART</b> ( ' <i>isodow</i> ' , <i>d</i> )	Returns the numeric day of the week (Monday is 1)
<b>DATE_PART</b> ( ' <i>doy</i> ' , <i>d</i> )	Returns the numeric day of the year.
<b>DATE_PART</b> ( ' <i>week</i> ' , <i>d</i> )	Returns the numeric week of the year.
<b>DATE_PART</b> ( ' <i>quarter</i> ' , <i>d</i> )	Returns the quarter of the year.
<b>DATE_PART</b> ( ' <i>hour</i> ' , <i>d</i> )	Returns the hour.
<b>DATE_PART</b> ( ' <i>minute</i> ' , <i>d</i> )	Returns the minute.
<b>DATE_PART</b> ( ' <i>second</i> ' , <i>d</i> )	Returns the second.
<b>DATE_PART</b> ( ' <i>milliseconds</i> ' , <i>d</i> )	Returns the millisecond.
<b>EXTRACT</b> (day FROM <i>d</i> )	Returns the numeric day of the month
<b>EXTRACT</b> (isodow FROM <i>d</i> )	Returns numeric day of the week (Monday is 1)
<b>EXTRACT</b> (month FROM <i>d</i> )	Returns numeric month of the year

See pages 7-17 to 7-18 for more information about **DATE\_PART** and **EXTRACT** functions.

This example lists details of all orders placed in October.

```
SELECT *
FROM orders
WHERE DATE_PART( 'month', order_date) = 10
```

Or:

```
SELECT *
FROM orders
WHERE EXTRACT( month FROM order_date) = 10
```

## DATE TIME ARITHMETIC

PostgreSQL does not currently support some of the functions to measure time elapsed as some other vendors of SQL. However basic arithmetic can be performed on dates, and the **DATE\_PART** function can also be leveraged for some problem solving as well.

### Examples

This example simply adds one week to the date of an order in the ORDERS table.

```
SELECT order_nr, order_date, order_date + 7 as "next week"  
FROM orders  
WHERE order_nr = 1007;
```

This example calculates days and weeks worked by reference to **CURRENT\_DATE**.

```
SELECT employee_nr, start_date, current_date - start_date as  
"days worked", (current_date - start_date)/7 as "weeks worked"  
FROM employee  
WHERE job = 'Manager';
```

This example calculates the number of months worked by each manager. Firstly **DATE\_PART** is used to establish years worked and multiply that by 12 to arrive at an equivalent number of years; secondly **DATE\_PART** is used to establish the number of months between start month and current month. The latter figure is then added to the former.

```
SELECT employee_nr, start_date,  
(DATE_PART ('year',current_date) – DATE_PART ('year', start_date))  
* 12 +  
DATE_PART ('month', current_date) – DATE_PART ('month', start_date)  
AS "Months Worked"  
FROM employee  
WHERE job = 'Manager';
```

## EXTRACT

This function can be used to extract from a Timestamp a specific unit of time. Data of the datatype DATE can also be used as this is cast automatically as TIMESTAMP.

**EXTRACT** ( field FROM timestamp )

Where **field** is the unit of measure of time, and **timestamp** is a column or valid date / time string. Valid values for **field** include:

SECOND	MINUTE	HOURL	DAY
WEEK	MONTH	QUARTER	YEAR

```
SELECT surname, start_date,
EXTRACT ( day FROM start_date) as day_started,
EXTRACT ( month FROM start_date) as month_started,
EXTRACT ( year FROM start_date) as year_started
FROM employee
```

Sample output would be:

	surname character varying (20)	start_date date	day_started double precision	month_started double precision	year_started double precision
1	Blake	2016-09-25	25	9	2016
2	Clark	2016-12-27	27	12	2016

You can also use **isodow** – day of week where Monday = 1 to Sunday = 7; or **dow** where Sunday = 0 to Saturday = 6. For example this query lists staff with a start date falling at a weekend:

```
SELECT surname, start_date
FROM employee
WHERE EXTRACT (isodow FROM start_date ) in ( 6, 7)
```

## DATE\_PART

**DATE\_PART** can also be used to extract a unit of time from a timestamp.

```
DATE_PART ( 'field' , timestamp )
```

Note that with this function, **field** must be quoted.

Valid values for **field** are the same as with the **EXTRACT** function and include:

SECOND	MINUTE	HOUR	DAY
WEEK	MONTH	QUARTER	YEAR

This example lists staff with a start date that fell at a weekend:

```
SELECT surname, start_date,  
DATE_PART( 'day', start_date) as day_started,  
DATE_PART ( 'month', start_date) as month_started  
FROM employee  
WHERE DATE_PART ( 'isodow', start_date) in ( 6, 7);
```

This example lists details of all orders placed during the year 2017.

```
SELECT *  
FROM orders  
WHERE DATE_PART ('YEAR', order_date) = 2017;
```

## INTERVAL

This function can be used to add or subtract a unit of measure of time to / from a date item.

Valid values for the unit of measure are:

YEAR  
MONTH  
DAY  
HOUR  
MINUTE  
SECOND

Each measure must be quoted.

```
SELECT start_date,  
start_date + INTERVAL '10' month as "Ten Months from now",  
start_date - INTERVAL '5' day as "Five days ago"  
FROM employee  
WHERE employee_nr = 1000;
```



## WORKSHOP EXERCISES

Write SQL statements or scripts to answer the following questions.

1. Select 5 raised to the power of 3.

2. Select the square root of 2.

3. Display the surname and salary from the employee table, rounding the salary to the nearest 100.

4. Find the average salary from the employee table rounded to 2 decimal places.

5. Select the town from the customer table in upper case

6. Select the first initial and following dot concatenated with surname of customers.

7. Select the surname right-padded to 10 characters, concatenated with the initials from the customer table.

8. Select today's date.

9. Select the date five days from now.

**10.** Select the date 4 months from now.

**OPTIONAL ADDITIONAL QUESTIONS (IF TIME PERMITS)**

**11.** Display the employee number, surname, job and weekly salary for each employee. Ensure that the weekly salary values are rounded to two decimal places on output.

**12.** Display the product code, description, sales price, cost price and instock values for each product. Order the output so that 'longest' product description is listed first.

**13.** Display the name (first initial only and surname combined (as in 'M.Knight' for example), street, town, and county of all customers.

**14.** Display the surname and street name (without the street number) for each customer. Ensure that the surnames are displayed in upper case. The last example on page 10 should help you with this one.

**15.** To the nearest £1000 what is the average sales value of orders placed within the last 6 months? (Order value to be calculated as quantity \* sales price.)



## SAMPLE ANSWERS

```
1.  SELECT POWER (5, 3)
```

```
2.  SELECT SQRT (2)
```

```
3.  SELECT surname, ROUND (salary,-2)
     FROM    employee
```

```
4.  SELECT ROUND (AVG (salary), 2)
     FROM    employee
```

```
5.  SELECT UPPER (town)
     FROM    customer
```

```
6.  SELECT SUBSTRING (initials FROM 1 FOR 2) || surname AS "name"
     FROM    customer
```

```
7.  SELECT RPAD (surname, 10) || initials as "custname"
     FROM    customer
```

```
8.  SELECT CURRENT_DATE
```

9a. **SELECT CURRENT\_DATE + 5**

9b. **SELECT CURRENT\_DATE + INTERVAL '5' DAY**

10. **SELECT CURRENT\_DATE + INTERVAL '4' MONTH**

11. **SELECT** employee\_nr, surname, job,  
          **ROUND** ( ( salary \* 12 ) / 52, 2 ) **AS** Weekly\_Salary  
**FROM** employee

12. **SELECT** product\_code, description, sales\_price, cost\_price, instock  
**FROM** product  
**ORDER BY LENGTH** ( description ) **DESC**

13. **SELECT SUBSTRING** ( initials FROM 1 FOR 1 ) || '. ' || surname **AS** Name,  
street, **RPAD** ( town , 15) **AS** Town , county  
**FROM** customer

14. **SELECT UPPER** ( surname ) **AS** surname,  
          **SUBSTRING** ( street **FROM** **POSITION** ( ' ' IN street ) + 1 ) **AS**  
Street\_Name  
**FROM** customer

15. **SELECT ROUND** ( **AVG** ( o.quantity \* p.sales\_price ), -3) **AS** Avg\_Value  
**FROM** product p **JOIN** orders o  
      **ON** o.product\_code = p.product\_code  
**WHERE** order\_date > **CURRENT\_DATE – INTERVAL '6' MONTH**

**Section 8**

**CONVERSION AND  
MISCELLANEOUS FUNCTIONS**



## CONVERSION AND MISCELLANEOUS FUNCTIONS

In this section the following topics relating to Conversion and Miscellaneous Functions will be covered:

- ♦ **Converting and Formatting Dates**
- ♦ **Converting and Formatting Numbers**
- ♦ **The COALESCE Function**
- ♦ **The CASE Statement**
- ♦ **The NULLIF Function**

## CONVERSION FUNCTIONS

The functions listed below allow conversion between different types of data as indicated by the characters ***n*** (number), ***s*** (string) and ***d*** (date):

<b>TO_CHAR( <i>d</i>, '<i>dformat</i>' )</b>	Converts the date <b><i>d</i></b> into a string expression formatted according to the date format mask ' <b><i>dformat</i></b> '
<b>TO_CHAR( <i>n</i>, '<i>nformat</i>' )</b>	Converts the number <b><i>n</i></b> into a string expression formatted according to the numeric format mask ' <b><i>nformat</i></b> '
<b>TO_DATE( '<i>s</i>', '<i>dformat</i>' )</b>	Converts the string <b><i>s</i></b> into a recognised PostgreSQL date value using the date format mask specified by ' <b><i>dformat</i></b> '
<b>TO_NUMBER( '<i>s</i>', '<i>nformat</i>' )</b>	Converts the string value <b><i>s</i></b> into a valid numeric item. The string or column must just consist of digits

## CONVERTING AND FORMATTING DATES

Listed below are some of the more useful date and time formatting elements. They can be combined to form a complete date format mask.

### BASIC DATE FORMATTING ELEMENTS

<b>YY</b>	2-digit year	<b>YYYY</b>	4-digit year
<b>MM</b>	Month number	<b>MONTH</b>	Full month name
<b>MON</b>	3-character month abbreviation	<b>RM</b>	Roman numeral month
<b>DD</b>	Day of month	<b>DAY</b>	Full day name
<b>DY</b>	3-character day abbreviation	<b>Q</b>	Quarter of year
<b>DDD</b>	Day of year	<b>D</b>	Day number of week (Sunday = 1)
<b>HH</b>	Hour of day (1-12). Or HH12	<b>HH24</b>	Hour of day (0-23)
<b>MI</b>	Minutes	<b>SS</b>	Seconds

## MORE DATE FORMATTING ELEMENTS

<b>WW</b>	Week of year
<b>W</b>	Week of month
<b>SSSS</b>	Seconds since midnight
<b>AM (or A.M.)</b>	Prints AM or PM depending on the time of day
<b>PM (or P.M.)</b>	Prints AM or PM depending on the time of day
<b>fm</b>	Prefix to a format element to suppress the padding of day and month names with spaces and of numbers with leading zeros.
<b>th</b>	Suffix to a number value to display that number with an ordinal suffix

### Examples

The following example displays the current (24-hour clock) time in hours and minutes separated by a colon:

```
SELECT TO_CHAR ( CURRENT_TIMESTAMP, 'HH24:MI' )
```

This lists employees with start dates in a format similar to Monday 7th March 2016:

```
SELECT surname, initials,
       TO_CHAR ( start_date, 'fmDay ddth Month YYYY' ) AS started
FROM   employee
ORDER BY start_date
```

### Note:

TO\_CHAR formats are case-sensitive. Words will be in the same case format as the format mask in the TO\_CHAR statement. The **th** modifier can be specified as **TH** for upper case ordinal number suffix.

The following example lists details of all employees who started on a weekend:

```
SELECT surname, start_date, TO_CHAR ( start_date, 'Dy' ) AS day
FROM employee
WHERE TO_CHAR ( start_date, 'Dy' ) IN ( 'Sat', 'Sun' )
```

This example lists details of all orders placed during the year 2017.

```
SELECT *
FROM orders
WHERE TO_CHAR ( order_date, 'YYYY' ) = '2017'
```

The remaining examples use the **TO\_DATE** function to convert a character string in a non-standard format to a **DATE** value:

Find all employees who started since 15/2/2017

```
SELECT surname, initials, start_date
FROM employee
WHERE start_date >= TO_DATE ( '17/2/2016', 'dd/mm/yyyy' )
```



## EXAMPLES WITH TO\_NUMBER

PostgreSQL does not carry out an implicit datatype conversion for numeric values which are held as strings. So this example would fail as both the numbers are contained within quotes:

```
SELECT '100' * '500'
```

**TO\_NUMBER** takes a string and applies a suitable quoted number format to it.

```
TO_NUMBER( 's', 'nformat')
```

**TO\_NUMBER** can be used to convert strings to valid numbers upon which arithmetical operations can then be performed.

```
SELECT TO_NUMBER( '100' , '999') * TO_NUMBER( '500', '999')  
AS "Result"
```

This will then return the numeric result (50000).

Another example but with extra formatting elements (see the following page) which will return 11000.

```
SELECT TO_NUMBER ( '£10,250.99', 'L99,999.99') + 749.01 AS  
"Numeric"
```

See also the **CAST** function on page 7 for an alternative.

## CONVERTING AND FORMATTING NUMBERS

Listed below are some of the more useful numeric format masks.

<b>9,999.99</b>	The number of 9's indicates the number of digits displayed; commas and decimal points will be displayed as indicated
<b>9,990.99</b>	Displays 0 before the decimal point if the value is less than 1
<b>09999</b>	Displays a number with leading zeros
<b>L9999</b>	Displays the (local) currency symbol in front of the number
<b>9999MI</b>	Displays a minus sign after the number (default is a leading minus sign)
<b>PL9999</b>	Displays plus sign before positive values
<b>9999PR</b>	Displays negative numbers in angle brackets
<b>SG9999</b>	Displays plus or minus sign before the number as applicable
<b>9999TH</b>	Appends an ordinal suffix e.g.: 1 <sup>st</sup> , 2 <sup>nd</sup>

### Examples

Display average salary (in pounds) formatted to 2 decimal places.

```
SELECT TO_CHAR( AVG(salary), 'L9,999.99' )
FROM    employee
```

This produces a result similar to this £2,101.79.

```
SELECT TO_CHAR( -8.2, '09.99MI' )
```

(Result = 08.20-)

```
SELECT TO_CHAR( 0.02, 'L990.99' )
```

(Result = £0.02)

### Note:

If the format mask is shorter than the number being converted then the output will show as hash (#) symbols indicating that insufficient digits are in the format mask.

## CAST FUNCTION

The Cast function and the cast operator :: (two colons) provide alternative methods of treating data held as one datatype as if of another datatype. A few examples will help illustrate this.

```
SELECT CAST( '100' AS integer) * 5 AS "Result";  
SELECT '100'::integer * 5 AS "Result";
```

Result = 500

Integer would give rise to an error with this example as the specification of the source string would be unsuitable. However decimal is an appropriate choice:

```
SELECT CAST ( '4.25' AS decimal(3,2) ) * 5 AS "Result";  
SELECT '4.25'::decimal(3,2) * 5 AS "Result";
```

Result = 21.25

This example converts a string to a valid date:

```
SELECT CAST( '2018-01-10' AS date) AS "The Date";  
SELECT '2018-01-10'::date AS "The Date";
```

This example converts an integer to a string:

```
SELECT CAST ( 2018 AS varchar) AS "String value";  
SELECT 2018::varchar AS "String value";
```

## MISCELLANEOUS FUNCTIONS

### CASE EXPRESSIONS

There are two types of **CASE** expression:

- ◆ the *simple* **CASE** expression which uses expressions to determine the returned value and
- ◆ the *searched* **CASE** expression which uses conditions to determine the returned value.

### THE SIMPLE CASE EXPRESSION

The syntax for the simple **CASE** expression is:

```
CASE expression
  WHEN expression1 THEN return_value1
  WHEN expression2 THEN return_value2
  .....
  WHEN expressionN THEN return_valueN
  [ ELSE return_default_value ]
END
```

The expression being evaluated and the test expressions *1* to *N* must all be of the same datatype. The return values, too, must all be of the same datatype, although the type may differ from that of the expressions.

When a match is found the **CASE** expression terminates.

The **ELSE** clause is optional and provides a default value in case no match is found. If it is omitted (and no match is found) the **CASE** expression returns NULL.

## THE SEARCHED CASE EXPRESSION

The syntax for the searched **CASE** expression is:

```
CASE
  WHEN condition1 THEN return_value1
  WHEN condition2 THEN return_value2
  .....
  WHEN conditionN THEN return_valueN
  [ ELSE return_default_value ]
END
```

The return values must all be of the same datatype.

When a match is found the **CASE** expression terminates.

The **ELSE** clause is optional and provides a default value in case no match is found. If it is omitted (and no match is found) the **CASE** expression returns NULL.

## EXAMPLES

## Example 1

A simple **CASE** expression:

```
SELECT department_nr,  
       CASE sex  
         WHEN 'M' THEN 'Mr. '  
         WHEN 'F' THEN 'Ms. '  
         ELSE '??.'  
       END AS title,  
       surname  
FROM employee
```

## Example 2

A searched **CASE** expression:

```
SELECT employee_nr, salary,  
       CASE  
         WHEN salary < 2000  
           THEN 'Not paid enough'  
         WHEN salary <= 3000  
           THEN 'Paid a reasonable amount'  
         ELSE 'Paid too much'  
       END AS Relative_pay  
FROM employee
```

## COALESCE FUNCTION

The COALESCE function returns the first not null value in a given list of values.

### Example

The following query would return the county or the town if there is no county or the street if there is neither town nor county

```
SELECT surname, COALESCE (county, town, street) AS which_address_field  
FROM customer
```

If a table has separate columns for the mobile, work and home telephone numbers, the following would give the first of the three for which a phone number was listed:

```
COALESCE ( mobile_phone, work_phone, home_phone ) AS telephone
```

and the following would give None if all were null:

```
COALESCE ( mobile_phone, work_phone, home_phone, 'None' ) AS telephone
```

## THE NULLIF FUNCTION

The NULLIF function compares two values and does the following:

- ◆ Returns null if the two values are the same.
- ◆ Returns the first value if the two values are different.

The general format for the NULLIF function is:

**NULLIF (expression1, expression2)**

### Example

Suppose we wish to display zero credit\_limits as NULL rather than zero.

```
SELECT NULLIF (credit_limit, 0)  
FROM customer
```



## WORKSHOP EXERCISES

Write SQL statements or scripts to answer the following questions.

1. Select today's date and time in a format of your own choice.

2. Select the cost price from the product table and display it with £ signs.

3. Select the surname and county from the prospect table displaying NULL counties as 'NONE'.

4. Select the surname and credit\_limit from the customer table followed by the words 'Small', 'Medium' or 'Large' according to whether the credit\_limit is less than 750, between 750 and 1000, or greater than 1000.

5. Show initials and surname (as one column) and the start date for all employees who started work on a Wednesday.

6. Display the number of order records for each day of the week on which orders have been placed.

7. Show title (Mr. if male, Ms. if female), initials and surname (in one column) and start date of all employees who started last year.



## SAMPLE ANSWERS

```
1.  SELECT TO_CHAR(CURRENT_TIMESTAMP,'HH24:MI:SS DAY DDth Month  
      YYYY')
```

```
2.  SELECT TO_CHAR(cost_price,'L999.99')  
      FROM product
```

```
3.  SELECT surname, COALESCE(county,'NONE')  
      FROM prospect
```

```
4.  SELECT surname, credit_limit,  
      CASE  
        WHEN credit_limit < 750 THEN  
          'Small'  
        WHEN credit_limit BETWEEN 750 AND 1000 THEN  
          'Medium'  
        WHEN credit_limit > 1000 THEN  
          'Large'  
      END AS "Scale of Credit"  
      FROM customer;
```

```
5.  SELECT RTRIM( initials ) || ' ' || surname AS Name,
      start_date AS Started
FROM  employee
WHERE TO_CHAR( start_date, 'Dy' ) = 'Wed'
```

```
6.  SELECT TO_CHAR( order_date, 'Day' ) AS Day, COUNT( * ) AS Orders
FROM  orders
GROUP BY TO_CHAR( order_date, 'Day' ), TO_CHAR( order_date, 'D' )
ORDER BY TO_CHAR( order_date, 'D' )
```

```
7.  SELECT CASE sex
      WHEN 'M' THEN 'Mr. '
      WHEN 'F' THEN 'Ms. '
      END || RTRIM( initials ) ||
      surname AS Name, start_date AS Started
FROM  employee
WHERE TO_NUMBER( TO_CHAR( start_date, 'YYYY' ), '9999' ) =
      TO_NUMBER( TO_CHAR( CURRENT_DATE, 'YYYY' ), '9999' ) - 1
```

**Section 9**  
**COMPLEX SUBQUERIES**



## COMPLEX SUBQUERIES

In this section the following topics relating to more complex subqueries will be covered:

- ♦ **Subquery Usages**
- ♦ **Subquery in a HAVING clause**
- ♦ **In-Line Views**
- ♦ **WITH Queries (Common Table Expressions)**
- ♦ **Correlated Subqueries**
- ♦ **Subqueries with joins**
- ♦ **MultiColumn Subqueries**
- ♦ **ANY, ALL and SOME Operators**
- ♦ **Subquery Rules**

## OVERVIEW

Basic subqueries were introduced in Section 6 and the examples showed a subquery as a `SELECT` statement enclosed within parentheses as part of the **WHERE** clause of an outer (or parent) `SELECT` statement. In this section, we will give more examples of how subqueries can be usefully defined within a containing `SELECT` statement.

## SUBQUERY USAGES

- ◆ In queries where an aggregate function result is required in a `WHERE` clause.
- ◆ In queries that identify non-matching records (an anti-join).
- ◆ In queries where subquery phrasing makes the query easier to understand than using a join.
- ◆ In queries where the data required is not held in a physical table.
- ◆ In DML statements that need to reference more than 1 table.



## SUBQUERY IN A HAVING CLAUSE

It is also possible to use a subquery in a **HAVING** clause.

This example shows the departmental average salary for each department where the departmental average is greater than the overall average.

The subquery itself is SCALAR, it calculates a single value.

```
SELECT department_nr, AVG( salary )  
FROM employee  
GROUP BY department_nr  
HAVING AVG( salary ) >  
        ( SELECT AVG( salary )  
          FROM employee )
```

## SUBQUERY IN A SELECT CLAUSE

Subqueries can be included in the SELECT clause of the parent query as the following example shows. The subquery is parenthesized and is SCALAR – it fetches a single value.

```
SELECT employee_nr, surname, salary,  
salary - round((SELECT avg(salary) FROM employee))  
AS diff_from_avg  
FROM employee
```

This will display each employee's salary and the difference between that figure and the overall average salary.

## SUBQUERY IN A FROM CLAUSE

FROM clause subqueries may be referred to as **INLINE VIEWS** but are sometimes also referred to more loosely as **DERIVED TABLES**.

**INLINE VIEWS** can help with complex queries. The following example has an Inline View in the FROM clause of the parent which contains information about total revenue for each product. This information is not available from a single base table. Also in the FROM clause of the parent is the base table **PRODUCT**.

Just as if there were two base tables in the FROM clause, a **JOIN** would be required, the same applies where the FROM clause contains one or more base tables and / or inline views. Here, **product\_code** is used to join the two.

```
SELECT p1.*  
FROM product p1 JOIN  
    (SELECT p2.product_code, SUM(p2.sales_price * o.quantity) AS sumsales  
    FROM product p2, orders o  
    WHERE p2.product_code = o.product_code  
    GROUP BY p2.product_code) s  
    ON p1.product_code = s.product_code  
WHERE s.sumsales > p1.instock * p1.cost_price
```

Note that the use of an alias is mandatory for an Inline View. In addition because a function has been used within it, it too has been aliased so that it can be referred to in the parent query.

## WITH QUERIES (COMMON TABLE EXPRESSIONS)

A Common Table Expression, or CTE, is a query whose result set you can reference in later sections of your query. They operate similarly to derived tables but allow referencing the result set repeatedly across the query.

Common Table Expressions can be helpful for simplifying your SQL, whilst also making it easier to read and maintain. There are also potential performance benefits.

The WITH clause can contain more than one statement, each being comma separated and having its own name and AS keyword at the beginning.

### Examples

This is the example derived table from the previous page, rewritten as a CTE for clarity.

```
WITH s AS
  (SELECT p2.product_code, SUM(p2.sales_price * o.quantity) AS sumsales
   FROM product p2, orders o
   WHERE p2.product_code = o.product_code
   GROUP BY p2.product_code)
SELECT p1.*
FROM product p1 JOIN s
  ON p1.product_code = s.product_code
WHERE s.sumsales > p1.instock * p1.cost_price
```

In this example the CTE calculates the monthly salary bill for each department. This is then referenced in both the parent query and its sub-query which follows, but without the need to recalculate the monthly totals for each reference. In this way, using a CTE may deliver a performance gain.

```
WITH summary AS
  (SELECT department_nr, SUM(salary) AS totalsal
   FROM employee
   GROUP BY department_nr)
SELECT department_nr, totalsal
FROM summary
WHERE totalsal > (SELECT SUM(totalsal)/3
  FROM summary)
```

## CORRELATED SUBQUERIES

A correlated subquery is one where the subquery references data retrieved by the parent query which may vary from row to row. For example:

```
SELECT *  
FROM employee e  
WHERE e.salary >  
  ( SELECT AVG(salary) FROM employee  
    WHERE department_nr = e.department_nr )
```

Note that a table alias in the parent SELECT statement is required, otherwise the WHERE clause in the subquery has no meaning. The column names from the tables used in a subquery cannot be referenced in the parent query.

The correlated subquery is executed once for each candidate row in the parent query. This can lead to poor performance. However, this technique may have better performance than other forms of subqueries, as each execution of the subquery may be fast. The performance is dependent on the data involved.

Correlated subqueries are frequently used in UPDATE and DELETE statements.

## SUBQUERIES WITH JOINS

Subqueries follow the same syntax rules as standard queries and can therefore include joins, sorts, groupings and subqueries.

In some circumstances, the subquery used to generate the desired criteria may need to refer to multiple tables.

The following example lists all customers who have not placed any orders for hard disk drives:

```
SELECT surname, initials, telephone
FROM    customer
WHERE customer_nr NOT IN
  ( SELECT o.customer_nr
    FROM    orders o JOIN product p ON o.product_code = p.product_code
    WHERE p.description LIKE '%Hard Disk Drive%' )
ORDER BY surname, initials
```

## MULTIPLE-COLUMN SUBQUERIES

A subquery is considered *multiple-column* when there is more than one column in the **SELECT** clause of the subquery and a comparison is to be made between the same number of columns in the parent query.

### Example

```
SELECT c.customer_nr, c.surname, o.order_date, o.quantity
FROM   customer c JOIN orders o ON c.customer_nr = o.customer_nr
WHERE (c.customer_nr, o.order_date) IN
      ( SELECT customer_nr, MAX( order_date )
        FROM orders
        GROUP BY customer_nr )
```

The above subquery retrieves the details of the most recent order placed by each customer.

The columns from the parent query being compared with the subquery must be enclosed within parentheses. The number of columns being compared from the parent query must match the number of columns being selected in the subquery.

## USING THE ANY, ALL AND SOME OPERATORS

The **ANY** and **ALL** operators, in conjunction with the main conditional operators, are used to test if an item value is less than, greater than or equal to *any value* or *all values* in a list of values respectively. The **SOME** operator means exactly the same as the **ANY** operator.

The first two examples below both return details of all employees who earn less than the highest paid manager. The third returns details of all employees who earn more than the highest paid manager:

```
SELECT surname, job, salary
FROM    employee
WHERE salary < ANY ( SELECT salary
                      FROM    employee
                      WHERE job = 'Manager' )
```

```
SELECT surname, job, salary
FROM    employee
WHERE salary < SOME ( SELECT salary
                      FROM    employee
                      WHERE job = 'Manager' )
```

```
SELECT surname, job, salary
FROM    employee
WHERE salary > ALL ( SELECT salary
                      FROM    employee
                      WHERE job = 'Manager' )
```

Be careful if the column used in the ALL comparison may contain nulls.



## SUBQUERY RULES

In a WHERE clause, the number of columns listed before the comparison operator must match the number of columns on the SELECT clause of the subquery given after the comparison operator (=, >, IN etc.).

The column listed before the comparison operator on the WHERE clause must be logically related to the column on the SELECT clause of the subquery given after the comparison operator.

Only columns referenced in the tables (or subqueries) in the FROM clause may be included in other parts of the outer SELECT statement.

FROM clause subqueries usually need joining clauses to other tables and subqueries on the FROM clause.



## WORKSHOP EXERCISES

Write SQL statements or scripts using subqueries to answer the following questions.

1. List the average credit\_limit grouped by town, but only showing towns where the average credit\_limit is greater than the overall average credit\_limit.

2. Use the ALL operator to list those customers whose credit limit exceeds the highest credit limit of London customers.

3. Who is the lowest earner in each department?

4. List orders which are worth more than the average of all orders.

5. List the employees who earn the three highest salary figures (n.b.: this may mean more than 3 employees)

## OPTIONAL ADDITIONAL QUESTIONS (IF TIME PERMITS)

6. Which department(s) has (have) the most employees?



## SAMPLE ANSWERS

```
1.  SELECT town, AVG(credit_limit)
    FROM customer
    GROUP BY town
    HAVING AVG(credit_limit) >
        ( SELECT AVG(credit_limit)
          FROM customer );
```

```
2.  SELECT surname, town, credit_limit
    FROM customer
    WHERE credit_limit > ALL ( SELECT credit_limit
                              FROM customer
                              WHERE town = 'London' );
```

```
3.  --Using a multi-column subquery:
    SELECT *
    FROM employee e
    WHERE (e.department_nr, e.salary)
           IN (SELECT e2.department_nr, MIN(salary)
              FROM employee e2
              GROUP BY e2.department_nr);

    --Using a correlated subquery:
    SELECT *
    FROM employee e
    WHERE e.salary = ( SELECT MIN(salary)
                      FROM employee e2
                      WHERE e.department_nr = e2.department_nr );

    --Using an in-line view:
    SELECT e.*
    FROM employee e,
         ( SELECT department_nr, MIN(salary) AS minsal
           FROM employee
           GROUP BY department_nr ) dm
    WHERE e.department_nr = dm.department_nr
    AND e.salary = dm.minsal;
```

```
4.  SELECT o.order_nr AS Ord_No,  
      p.description AS Product,  
      o.quantity * sales_price AS Value  
FROM orders o JOIN product p  
      ON o.product_code = p.product_code  
WHERE o.quantity * p.sales_price >  
      ( SELECT AVG( o.quantity * p.sales_price )  
        FROM orders o JOIN product p  
          ON o.product_code = p.product_code );
```

```
5.  SELECT surname, job, salary  
FROM employee  
WHERE salary IN (SELECT distinct salary  
                  FROM employee  
                  ORDER BY salary DESC  
                  LIMIT 3)  
ORDER BY salary DESC;
```

```
6.  SELECT department_nr, COUNT( * )  
FROM employee  
GROUP BY department_nr  
HAVING COUNT( * ) =  
      (SELECT MAX(a.headcount)  
       FROM  
         (SELECT COUNT(*) as headcount  
          FROM employee  
          GROUP BY department_nr) a);
```

Note that calls to aggregate functions cannot be nested in PostgreSQL so this subquery would return an error:

```
SELECT MAX ( COUNT(*) )  
FROM employee  
GROUP BY department_nr
```

This problem has been avoided in the workshop solution by having an extra level of subquery.

**Section 10**  
**MANAGING**  
**DATA**





## MANAGING DATA

In this section the following topics relating to Managing Data will be covered:

- ♦ **Inserting Rows**
- ♦ **Updating Rows**
- ♦ **The DELETE Statement**
- ♦ **The TRUNCATE Statement**
- ♦ **Verifying Updates**
- ♦ **Transaction Control**

## INSERTING ROWS

Using the SQL **INSERT** statement, rows may be inserted into tables in two ways. They may be inserted as lists of values, or they may be selected from another table using a query.

### INSERT USING A VALUE LIST

The following syntax is used to insert a single row:

```
INSERT INTO table [ ( column-list ) ]  
VALUES  
( value-list )
```

Where ***table*** is the name of the table to update, ***column-list*** is a list of columns in the specified table and ***value-list*** is the list of values to be inserted for the row. Further rows may be added by providing additional value lists, each in parenthesis and comma separated from the previous one.

There must be as many values in the value list as there are columns in the column list. If the column list is not specified, there must be a value in the value list for each column of the table.

Columns with a **NOT NULL** constraint and no default value *must* be specified in the column list and values must be specified for them in the value list. Where a column has a **DEFAULT** value, the key word DEFAULT can be used in the value list, or some alternative value provided.

## Examples

The examples below both insert a row into the *orders* table by specifying a value for each column in the table.

```
INSERT INTO orders ( order_nr, product_code, customer_nr,
                    order_date, quantity )
```

```
VALUES
```

```
( 3456, 'HD12', 1036, '2018-02-05', 24 )
```

```
INSERT INTO orders
```

```
VALUES
```

```
( 2431, 'AM17', 1019, NULL, 40 )
```

The following example lists only the **NOT NULL** columns of the *orders* table. The *order\_date* column will automatically acquire a **NULL** value.

```
INSERT INTO orders ( order_nr, product_code, customer_nr, quantity )
```

```
VALUES
```

```
( 3457, 'MW97', 1087, 24 )
```

In psql, the `\set` command can be used to declare a local variable and assign a value to it. The variable can then be referenced in one or more INSERT statements. Note the colon prefix to the variable name where it is referenced in the INSERT. This option is not available in pgAdmin 4.

```
\set custno 1317
```

```
INSERT INTO orders
```

```
VALUES
```

```
( 1026, 'MW97', :custno, '2018-02-05', 1000 )
```

An INSERT can use a **Sequence** to generate a value with the obvious example being to generate a Primary Key.

In this example the primary key value will be obtained by using the next sequential value from a Sequence called *orderseq*. Further details about defining Sequences can be found in a later section in this manual.

```
INSERT INTO orders
VALUES
( NEXTVAL('ORDERSEQ'), 'MW97', 1317 , '2018-02-05', 1000
)
```

Multi row inserts can be made by providing comma-separated value lists. This example will add three rows to the Orders table:

```
INSERT INTO orders
VALUES
( 2000, 'MW97', 1317 , '2018-02-05', 1000 ),
( 2001, 'HD12', 1443 , '2018-02-05', 1000 ),
( 2002, 'MW97', 1317 , '2018-02-05', 1000 );
```

#### Notes:

**NULL** values can be specified explicitly using either the keyword **NULL** or by using the keyword **DEFAULT**. With **DEFAULT**, any default value will be used or else **NULL** if there is no default.

Dates must be bounded by single quotes and must be in the default format expected for a date. If dates are to be given in another format they must be converted using the **TO\_DATE** function.

If it is necessary to store the time as well as the date an expression similar to the following example can be used:

```
TO_DATE( '15/03/12 12:35', 'DD/MM/YY HH24:MI' )
```

## UPDATING ROWS

The **UPDATE** statement allows columns to be updated with user-specified values or with values retrieved from another table via a SELECT statement.

### UPDATE WITH USER-SPECIFIED VALUES

The following syntax is used to update one or more rows with user-specified values:

```
UPDATE table
      SET column-1 = value-1 ,
          column-2 = value-2 ,
          . . . . .
          column-n = value-n
[ WHERE conditions ]
```

If the optional **WHERE** clause is omitted **all** rows in the specified table will be updated, otherwise only the rows matching the specified conditions will be affected.

### Examples

This statement changes the telephone number and credit limit of customer number 1317.

```
UPDATE customer
      SET telephone = '0161 935 1071' ,
          credit_limit = 2000
WHERE customer_nr = 1317
```

This statement changes the credit limit for all customers in Manchester.

```
UPDATE customer
      SET credit_limit = 1500
WHERE INITCAP( town ) = 'Manchester'
```

This statement gives every employee a 25% pay rise.

```
UPDATE employee
      SET salary = salary * 1.25
```

## UPSERT ROWS

With this option, new rows can be added or existing rows updated in the same statement, the obvious example being by checking for the existence of the Primary Key value for a row.

```
INSERT INTO table  
VALUES (value-list)  
ON CONFLICT ON CONSTRAINT constraint-name  
{DO NOTHING |  
DO UPDATE SET col1 = value, col2 = value.. }  
[RETURNING col-1, col-2..col-n ];
```

The **ON CONFLICT** clause states what to do rather than raise a Unique violation or exclusion constraint violation. The clauses **DO NOTHING** and **DO UPDATE** represent alternates. Optionally a **RETURNING** clause can be appended to display values for affected rows.

In this example, because there is already an order with Primary Key value 1000 in the orders table, its quantity will be updated to 200.

```
INSERT INTO orders  
VALUES  
(1000, 'MW97', 1317, CURRENT_DATE, 200)  
ON CONFLICT ON CONSTRAINT ord_pkey  
DO UPDATE SET QUANTITY = 200  
RETURNING order_nr, quantity;
```

## DELETING ROWS

### THE DELETE STATEMENT

Rows are deleted from a table using the following statement:

```
DELETE FROM table  
[ WHERE conditions ]
```

If the optional **WHERE** clause is omitted **all** rows in the specified table will be deleted, otherwise only the rows matching the specified conditions will be affected.

### Examples

The example below deletes all orders for a specific product.

```
DELETE FROM orders  
WHERE product_code = 'MW97';
```

The example below deletes all suppliers who do not supply any products.

```
DELETE FROM supplier  
WHERE supplier_nr NOT IN ( SELECT supplier_nr  
                           FROM product )
```

## THE TRUNCATE STATEMENT

The statement:

```
TRUNCATE TABLE name
```

Removes all rows from the specified table. It is faster than the **DELETE** command because it does not scan the table - space is reclaimed immediately on commit.



## TRANSACTION CONTROL

Within a Transaction, changes made by SQL statements (such as **INSERT**, **UPDATE**, **DELETE**, **TRUNCATE** and **DDL** statements) are not immediately reflected in the database. They are *posted*, but not yet *committed*. Only the user who issued the updates can view the effect of the changes; other users will only see the old information.

In addition, each row affected is locked; in other words, other users can read those rows (and their old values) but may themselves not update those rows.

The work done by these statements needs to be committed in order to affect the database permanently.

### THE COMMIT AND ROLLBACK COMMANDS

The SQL statement **COMMIT** will apply all changes made to the database since the last **COMMIT** was issued and release all locks retained on the affected rows.



**COMMIT**

When one or a series of complex inserts, updates or deletes have been made it is advisable to check the results before issuing a **COMMIT**. If any mistakes have been made then the changes can be reversed or *rolled back*.

The **ROLLBACK** statement will reverse all changes made since the last **COMMIT** was issued and release all row locks acquired. In the event of a database or system crash, any uncommitted work will automatically roll back.



**ROLLBACK**

Committed transactions cannot be rolled back.

## TRANSACTIONS

You can start a transaction in PostgreSQL with the keywords **BEGIN TRANSACTION**, or just **BEGIN**.

**BEGIN TRANSACTION**

Using a transaction means that you have the option to **ROLLBACK** fully, or partially by using Savepoints, until you complete the transaction by using **COMMIT** or **END TRANSACTION** commands.

## SAVEPOINTS

A logical transaction may consist of a large number of SQL statements. Greater control over recovery actions can be gained by separating each stage (comprising one or more SQL statements) with a **savepoint**.

Rollback is then permitted to any particular savepoint in the current transaction.

Savepoints within a transaction must have unique names. A transaction is ended when either a commit or an unconditional rollback occurs, explicitly or implicitly.

A savepoint is created using the following syntax:

**SAVEPOINT *name***

Rollback to a specific savepoint is achieved using the following command:

**ROLLBACK TO [ SAVEPOINT ] *name***

**Example:**

A new order is added to the orders table.

A Savepoint is added so that the subsequent update can be rolled back if necessary. The order is updated.

Rolling back to the savepoint undoes the update whilst retaining the insert.

Finally the insert is committed meaning that the opportunity to rollback ceases.

```
BEGIN TRANSACTION;  -- OR BEGIN
```

```
INSERT INTO ORDERS  
VALUES (1101, 'MW97',1317,DEFAULT,500);
```

```
SAVEPOINT ORD1;
```

```
UPDATE ORDERS  
SET QUANTITY = QUANTITY * 1.1  
WHERE ORDER_NR = 1101;
```

```
SELECT *  
FROM ORDERS  
WHERE ORDER_NR = 1101;
```

```
ROLLBACK TO SAVEPOINT ORD1;
```

```
COMMIT; -- OR END TRANSACTION
```

**Note:**

When using **psql**, an error in a transaction block will normally throw an exception causing the transaction to be aborted - requiring the transaction to be explicitly rolled back to before the error. If you try to run any SQL statement other than a ROLLBACK, you will receive the following error message:

**ERROR: current transaction is aborted, commands ignored until end of transaction block**

This is expected behaviour, but it can be very annoying if it happens when you are in the middle of a large transaction and mistype something!

The **ON\_ERROR\_ROLLBACK** control variable (default value: off) allows changing the behaviour of psql to automatically rollback just the statement causing the error, allowing the transaction to continue:

**\set ON\_ERROR\_ROLLBACK on**

View the current setting using:

**\echo :ON\_ERROR\_ROLLBACK**

## IMPLICIT COMMITS

If the **AUTOCOMMIT** setting is *on* (its default is *on*) then work will be committed automatically on completion of each **INSERT**, **UPDATE** and **DELETE** statement.

In psql, the value of the control variable **AUTOCOMMIT** may be set using the command

**\set AUTOCOMMIT on** (or off)

To view the current setting for **AUTOCOMMIT** use

**\echo :AUTOCOMMIT**

In this example a record is added to the Department table but because the **AUTOCOMMIT** setting has been switched off, the transaction can be rolled back.

```
sat=# \set AUTOCOMMIT off
sat=# insert into department values
sat-# (50, 'HR', 'Croydon');
INSERT 0 1
sat=# rollback;
ROLLBACK
```

This setting can be set on a more permanent basis by adding the desired value to the user's **~/.psqlrc** start-up file. Note that this is a client-side option and the name **AUTOCOMMIT** is case-sensitive. Setting this to off is equivalent to prefixing your transaction with the word **BEGIN**.

### Note:

Regardless of the **AUTOCOMMIT** setting, if you explicitly **BEGIN** a transaction, you have the ability to **ROLLBACK**, fully or partially by using savepoints, until you terminate the transaction with either a **COMMIT** or **END TRANSACTION** command.

## VERIFY UPDATES

Column values updated or changed as a result of issuing an **INSERT**, **UPDATE** or **DELETE** statement (but not of a **TRUNCATE** statement) may be verified using the **RETURNING** clause of the issued statement.

After issuing an **INSERT**, **UPDATE** or **DELETE** statement a **RETURNING** clause can be added to display the values of rows affected.

### Examples:

The example below returns the surname and new salary of each of the managers in the employee table.

```
UPDATE employee
SET salary = salary * 1.05
WHERE job = 'Manager'
RETURNING surname as "Employee", salary as "New Salary";
```

This next example returns the new annual salary (after the update) and the name of the employee affected:

```
UPDATE employee
SET salary = salary * 1.05
WHERE employee_nr = 1024
RETURNING salary * 12 as "New Annual Salary",
Surname as "Empname" ;
```

This example below returns the name of the employee deleted:

```
DELETE FROM employee  
WHERE employee_nr = 1024  
RETURNING RTRIM( initials ) || surname as "Full name" ;
```

The **RETURNING** clause may not be used to return aggregate values. They are not allowed in this clause.

## IMPORT DATA USING CSV FILES

Data can be imported to a table from a suitably structured csv file using the copy command.

```
COPY <tablename>  
FROM '<path_to_filename>' DELIMITER 'delimiter character'  
FILETYPE [ HEADER ];
```

The **COPY** clause names the table into which data is being imported. A parenthesised and comma separated column list can be specified if the source file does not provide values for all columns. The column list must reflect the order in which columns are listed in the source file.

The **FROM** clause provides the path to filename, within single quotes.

The **DELIMITER** clause specifies the field separator character, within single quotes.

**CSV** denotes the file type. Binary or Text files can also be used. **HEADER** indicates that the first row in the source data contains column header names so can be omitted if this is not the case.

```
COPY employee  
FROM '\\home\postgresql\extdata.csv' DELIMITER ',' CSV HEADER;
```



## EXPORTING DATA

There are two options for exporting data using **psql**.

1. The **COPY** command involves the server writing the export file, which will be created on the database server, not on the requester's local machine.
2. The second option is to use the **\copy** command where the destination file can be created on the requester's own local machine.

### COPY COMMAND

Exporting uses the same style of syntax as importing, but using **COPY .. TO** (rather than **COPY .. FROM**).

The **COPY** clause specifies the source table. You can provide a parenthesised and comma separate list of column names if you do not wish to export values from all columns.

The **TO** clause specifies the path to the export filename, within single quotes.

The **DELIMITER** clause specifies the delimiter character within single quotes.

**FILETYPE** can be CSV, TEXT or BINARY.

The **HEADER** clause where specified denotes that the column names from the name will be used to populate the first row of the export file.

```
COPY <tablename> [ ( column list ) ]  
TO '<path_to_filename>' DELIMITER 'delimiter character' FILETYPE [  
HEADER ];
```

In this example the data from the employee table is copied to a csv file using psql:

```
sat=#  
sat=# copy employee  
sat=# to 'c:\jc\allstaff.csv'  
sat=# delimiter ','  
sat=# csv  
sat=# header;  
COPY 31
```

The final line shows the number of rows copied when the statement was executed.

	A	B	C	D	E	F	G	H	I
1	employee	surname	initials	sex	start_date	job	manager	salary	department_nr
2	1035	Martin	R.J.	M	27/01/2017	Salesman	1036	1562.5	30
3	1033	Allen	P.G.	F	25/09/2016	Salesman	1023	2500	10
4	1007	Turner	J.	M	03/12/2017	Salesman	1036	1875	30
5	1019	Jameson	N.H.B.	F	26/10/2016	Clerk	1036	1187.5	30
6	1013	Ward	K.T.	M	31/05/2017	Salesman	1024	1562.5	20
7	1026	Ford		M	25/07/2016	Analyst	1023	3750	10

Above is an extract from the exported data.

## COPYING USING THE \COPY OPTION

The \copy option allows for the creation of a local file on the requester's own machine. It also gives the flexibility to specify the required data as a SELECT statement.

**\copy (SELECT \* FROM tablename) to '<path to filename.csv' with csv**

For example:

```
sat=# \copy (SELECT * FROM EMPLOYEE WHERE JOB = 'Manager')
TO 'c:\jc\managers.csv' WITH CSV;
COPY 3
sat=#
```

This will generate a list of managers from the employee table. The final line indicates that three rows were selected when the statement was executed. The output generated is shown here:

	A	B	C	D	E	F	G	H	I
1	1036	Blake	S.T.	M	25/09/2016	Manager	1000	3740.63	30
2	1023	Clark	M.	F	27/12/2016	Manager	1000	3215.63	10
3	1024	Jones	D.S.	M	30/03/2017	Manager	1000	4099.92	20

There is no option to generate column headings with the \copy option.

## WORKSHOP EXERCISES

Write SQL statements or scripts to achieve the following objectives.

**Guidance:**

Answer each question within a transaction, i.e.: the first line will be:

**BEGIN TRANSACTION;**

The final line will be either:

To commit, either: **END TRANSACTION** or **COMMIT;**

To rollback: **ROLLBACK;**

1. Insert a new department into the department table. Commit your change.

2. Insert an order into the orders table. Commit your change.

3. Give all employees in department 10 a pay increase of £200. Commit your change.

4. Delete prospect number 11. Commit your change.

5. Count all current prospects. Delete prospects with a credit rating less than 10 who are based in London. How many prospects are left? Create a savepoint.

6. Delete prospects with a credit rating less than 10 who are based in Truro. How many prospects are left? Rollback to your savepoint. How many prospects are left now? Do a full rollback. How many prospects are left now?

*Optional additional questions (if time permits) on next page*

**OPTIONAL ADDITIONAL QUESTIONS (IF TIME PERMITS)**

7. For any orders that have been placed on a weekend move the order date to the following Monday.

8. Write a single **UPDATE** statement to increase the credit limit of London customers by 10%, Manchester customers by 6% and Birmingham customers by 4%. No other customer credit limits should be affected.

## SAMPLE ANSWERS

```
1.  INSERT INTO department
    VALUES
    ( 60, 'IT', 'Cellar');

    COMMIT;
```

```
2.  INSERT INTO orders
    VALUES(2000, 'MW97', 1317, CURRENT_DATE, 1);

    COMMIT;
```

```
3.  UPDATE employee
    SET salary = salary + 200
    WHERE department_nr = 10;

    COMMIT;
```

```
4.  DELETE FROM prospect
    WHERE prospect_nr = 11;

    COMMIT;
```

```
5.  SELECT COUNT( * ) FROM prospect;

    DELETE FROM prospect
    WHERE credit_rating < 10
    AND town = 'London';

    SELECT COUNT( * ) FROM prospect;

    SAVEPOINT s1;
```

```
6.  DELETE FROM prospect
    WHERE credit_rating < 10
    AND town = 'Truro';

    SELECT COUNT( * ) FROM prospect;

    ROLLBACK TO s1;

    SELECT COUNT( * ) FROM prospect;

    ROLLBACK;

    SELECT COUNT( * ) FROM prospect;
```

```
7a. UPDATE orders
    SET order_date =
    CASE EXTRACT (isodow FROM order_date)
    WHEN 6 then order_date + INTERVAL '2' DAY
    ELSE order_date + INTERVAL '1' DAY
    END
    WHERE EXTRACT (isodow from order_date) IN ( 6, 7);
```

```
7b. UPDATE orders
    SET order_date =
    CASE WHEN TO_CHAR(order_date, 'Dy') = 'Sat'
    THEN order_date + 2
    ELSE order_date + 1
    END
    WHERE TO_CHAR(order_date, 'Dy') IN ('Sat', 'Sun');
```

```
8.  UPDATE customer
    SET credit_limit = credit_limit * CASE LOWER( town)
                                     WHEN 'london' THEN 1.1
                                     WHEN 'manchester' THEN 1.06
                                     ELSE 1.04
    END
    WHERE LOWER(town) IN ( 'london', 'manchester', 'birmingham' );
```

**Section 11**  
**MANAGING**  
**TABLES**





## MANAGING TABLES

In this section the following topics relating to Managing Tables will be covered:

- ♦ **Creating Tables**
- ♦ **Datatypes**
- ♦ **Column Level Constraints**
- ♦ **Add A Column to a Table**
- ♦ **Change Column Widths**
- ♦ **Rename a Column**
- ♦ **Drop a Column**
- ♦ **Add a Constraint**
- ♦ **List Constraints**
- ♦ **Copy a Table**
- ♦ **Drop a Table**

## INTRODUCTION

Database design is a specialised task, a discussion of which is beyond the scope of this course. It involves the application of a number of design techniques and methodologies.

Once the design process is complete the database designer should be able to establish:

- ◆ What tables are required
- ◆ The relationships between those tables
- ◆ What columns should appear in those tables
- ◆ The type of data each column contains
- ◆ Which columns are mandatory and may not contain Null values
- ◆ Which columns must contain unique values
- ◆ What other constraints should be placed on column data
- ◆ Which column or set of columns represents the primary key for each table
- ◆ Which columns represent foreign keys
- ◆ What indexes should be created for each table

The following information should also be established before the database is implemented:

- ◆ Who is allowed to access the database
- ◆ Who is allowed to create and change tables, columns and indexes
- ◆ Who is allowed to retrieve or update data
- ◆ What views (logical representations of table data) should be created

This section is concerned primarily with the creation and maintenance of tables.

Section 10 describes how rows may be inserted, updated and deleted.

## CREATING TABLES

A table is created using the following SQL statement:

```
CREATE TABLE name (  
    column-1 DATATYPE [ DEFAULT value ] [ constraints ],  
    column-2 DATATYPE [ DEFAULT value ] [ constraints ],  
    .....  
    .....  
    column-n DATATYPE [ DEFAULT value ] [ constraints ]  
    [ , table-constraints ] )  
    [ TABLESPACE name ]  
    [ other possible clauses ] ;
```

Where ***name*** is the table name and ***column-1*** to ***column-n*** are the column names.

These names must start with an alphabetic character and consist of no more than 30 letters, numbers and underscores. Other characters should be avoided.

The **DATATYPE** is mandatory; the most commonly used datatypes are listed overleaf.

An optional default value (associated with the keyword **DEFAULT**) may be specified. This value is used if no other value is specified when inserting a row into the table.

Constraints (of which **NOT NULL** is the most common) can be specified at column level or at table level; they are discussed in more detail later in this section.

If the **TABLESPACE** clause is omitted, the table is created in the tablespace specified by the configuration parameter **default\_tablespace**. If no value is set for **default\_tablespace**, the location of the database system catalogs is used.

Tablespaces, created by the DBA, are used to map logical storage to physical (disk) storage.

Other clauses are available but beyond the scope of this course.

## DATATYPES

The most frequently used datatypes are listed below:

<b>CHAR( <i>n</i> )</b>	<p>A fixed-length character string <i>n</i> characters long. Default length is 1 character if no size is specified. The maximum allowed size is 10485760.</p> <p>If the length of the value is less than the specified column width then the value is space-padded to the right.</p> <p>Example: national_ins_nr <b>CHAR( 9 )</b></p>
<b>VARCHAR( <i>n</i> )</b>	<p>A variable length character string up to <i>n</i> characters long. The maximum allowed size is 10485760.</p> <p>Example: surname <b>VARCHAR( 30 )</b></p>
<b>TEXT</b>	Store strings of any length, up to a maximum size of about 1Gb.
<b>SMALLINT</b>	A 2 byte signed integer in the range -32,768 to 32,767
<b>INT</b>	A 4 byte signed integer in the range -2,147,483,648 to 2,147,483,647
<b>SERIAL</b>	The same as INT except that PostgreSQL will automatically generate and populate values into a Serial column. Equivalent to Auto Increment in other vendor versions of SQL.
<b>NUMERIC( <i>p</i>, <i>s</i> )</b>	<p>A signed number <i>p</i> digits long (in total) including <i>s</i> decimal places, with <math>p \leq 1000</math> and <math>s \leq p</math>. DECIMAL(<i>p</i>,<i>s</i>) is equivalent.</p> <p>Example: salary <b>NUMERIC( 6,2 )</b></p>
<b>NUMERIC</b>	Numeric values of any precision and scale can be stored.
<b>FLOAT(<i>n</i>)</b>	Floating point values, implemented as IEEE standard <i>real</i> (for $1 \leq n \leq 24$ ) or <i>double precision</i> (for $25 \leq n \leq 53$ ) datatypes, which have platform specific precisions of at least 6 digits and 15 digits respectively.
<b>DATE</b>	<p>A valid date</p> <p>Example: start_date <b>DATE</b></p>
<b>TIME</b>	A valid time
<b>TIMESTAMP</b>	Date and time
<b>INTERVAL</b>	To store a unit of time

## CONSTRAINTS

The purpose of a constraint is to maintain data integrity. The more constraints that are specified for a table, the fewer validation routines will be needed in applications which access that table. However, more constraints also mean slower updates.

Constraints are stored as separate objects in the database.

There are essentially five types of constraint:

- ◆ Not Null constraints
- ◆ Unique (value) constraints
- ◆ Primary key constraints
- ◆ Foreign key constraints
- ◆ Check constraints

The syntax of these constraints varies according to whether they are specified at column level or at table level.

## BASIC CONSTRAINT SYNTAX

The basic syntax for a constraint is:

**[ CONSTRAINT *name* ] CONSTRAINT-TYPE [ *attributes* ]**

Where ***name*** is a constraint name (which must be unique within the user's schema), **CONSTRAINT-TYPE** is the type of constraint and the ***attributes*** depend upon the type of constraint specified.

If the constraint name is omitted a default name will be assigned for it in the database. For example, a foreign key constraint will acquire a name with the style *table\_column\_fkey*.

Only Not Null and Primary Key constraints can subsequently be dropped or disabled without reference to their names.

**CONSTRAINT-TYPE** syntax is defined overleaf for both column and table level constraints.

## COLUMN LEVEL CONSTRAINT TYPES

All constraints (except **NOT NULL**) may be specified at either the column level (using the syntax below) or at the table level. The syntax for each of the column level constraints is:

<b>NOT NULL</b>	Ensures that all rows contain a value for the column.
<b>UNIQUE</b>	Ensures that all rows contain a unique value for the column.
<b>PRIMARY KEY</b>	Indicates the primary key for the table and implies <b>UNIQUE</b> and <b>NOT NULL</b> .
<b>REFERENCES <i>table</i> ( <i>key</i> )</b>	Indicates a foreign key for the specified table. <b><i>key</i></b> is the name of the primary key column in the specified table.
<b>CHECK ( <i>condition</i> )</b>	Data in the column must meet the specified <b><i>condition</i></b> . A single column can be specified with more than one check constraint.

## TABLE LEVEL CONSTRAINT TYPES

Constraints which apply to multiple columns can only be defined as table level constraints. Table level constraints are specified following the definition of all the columns in the table. The corresponding table level constraints are:

**UNIQUE ( *column-list* )**

**PRIMARY KEY ( *column-list* )**

**FOREIGN KEY ( *column-list* ) REFERENCES *table* ( *column-list* )**

**CHECK ( *condition* )**

## CONSTRAINT EXAMPLES

### COLUMN LEVEL CONSTRAINTS

```
customer_nr SMALLINT CONSTRAINT customer_pk PRIMARY KEY  
  
surname VARCHAR( 20 ) NOT NULL  
  
national_ins_nr CHAR( 9 ) CONSTRAINT ninsnr UNIQUE  
  
sex CHAR CONSTRAINT checksex CHECK ( LOWER( sex ) IN ( 'm', 'f' ) )  
  
department_nr SMALLINT CONSTRAINT emp_dept_fk REFERENCES  
                department ( department_nr )
```

### TABLE LEVEL CONSTRAINTS

```
CONSTRAINT pkey PRIMARY KEY ( order_nr, line_nr )  
  
CONSTRAINT nat_ins_unq UNIQUE ( national_ins_nr )  
  
CONSTRAINT pricecheck CHECK ( sales_price > cost_price )  
  
CONSTRAINT fkey FOREIGN KEY ( order_nr, line_nr )  
                REFERENCES order_details ( order_nr, line_nr )
```

## NOTES ON KEY CONSTRAINTS

### PRIMARY KEY CONSTRAINTS

All tables should have a **primary key**. PostgreSQL ensures that the value of the primary key is unique from row to row. **NULL** values are not allowed.

Any column (or set of columns) which have the **NOT NULL** and **UNIQUE** constraints applied have the same attributes as a primary key.

PostgreSQL automatically generates unique B-tree indexes for **PRIMARY KEY** and **UNIQUE** constraints. If there is already an index in place on the column(s) then the constraint uses the existing index. Indexes are further discussed later in this Section.

### FOREIGN KEY CONSTRAINTS

The purpose of a **foreign key** is to ensure the maintenance of *referential integrity*.

Taking the customer and order tables as an example, if no foreign key constraint referencing the customer table was applied to the orders table it would be possible, using standard SQL statements, to delete customers who still have outstanding orders. In addition, orders could be inserted for non-existent customers. The results would be that some orders in the orders table would apply to no known customer.

The following optional clauses may be applied to foreign key constraints:

**ON DELETE RESTRICT** a parent record cannot be deleted until all referencing child records have been deleted.

**ON DELETE NO ACTION** essentially the same as ON DELETE RESTRICT but the timing of the check may be deferred to a later point within a transaction. This is the default action.

**ON DELETE CASCADE** which causes referencing child records to be deleted automatically whenever a parent record is deleted.

**ON DELETE SET NULL** sets the referencing columns in child records to null.

**ON DELETE SET DEFAULT** sets the referencing columns in child records to their default values. There must be a parent record matching the default values if they are not null.

These options may also be set separately for **ON UPDATE**.



# TABLE CREATION EXAMPLE

The example **CREATE TABLE** statement below creates an employee table (similar to that described in Appendix A) in the tablespace *pg\_default*.

```
CREATE TABLE employee2 (
  employee_nr SMALLINT CONSTRAINT emp2_pk PRIMARY KEY ,
  surname     VARCHAR( 20 )    NOT NULL ,
  initials    VARCHAR( 6 ) ,
  sex        CHAR           CONSTRAINT checksex
                                CHECK ( LOWER( sex ) IN ( 'm', 'f' ) ) ,
  nat_ins_nr  CHAR( 9 )    CONSTRAINT ninsnr UNIQUE ,
  start_date  DATE ,
  job        VARCHAR( 15 )    NOT NULL ,
  dept_nr    SMALLINT      DEFAULT 10 ,
  salary     NUMERIC( 6,2 )  CONSTRAINT checksal
                                CHECK ( salary <= 9000 ) ,
  manager    SMALLINT ,
  CONSTRAINT emp_fkey FOREIGN KEY ( dept_nr ) REFERENCES
                                department ( department_nr )
)
TABLESPACE pg_default
```

The foreign key constraint (*emp\_fkey*) is specified above as a table level constraint. As it only references a single column it could have been specified at the column level instead using a different format:

```
dept_nr    SMALLINT  DEFAULT 10
            CONSTRAINT emp_fkey REFERENCES
            department ( department_nr)
```

## MANAGING TABLES

### COLUMN OPERATIONS

Tables can be changed in a number of ways using the SQL statement **ALTER TABLE**. Its syntax is essentially the same as for the **CREATE TABLE** statement. Certain restrictions apply and these are noted in the examples given below.

#### ADDING COLUMNS

The first example adds two columns to the customer table, the first of which has a named Check Constraint.

```
ALTER TABLE customer  
ADD COLUMN title char(4)  
CONSTRAINT cust_title_chk CHECK (title IN ('Mr.','Mrs.','Ms.')),  
ADD COLUMN bank varchar(20);
```

It is not possible to specify the position of the column within the table.

PostgreSQL allows a NOT NULL column with a default value to be added to a table in one step.

#### Example

```
ALTER TABLE customer ADD COLUMN  
sales_manager_code VARCHAR(4) DEFAULT 'XXXX' NOT NULL;
```

When the above statement is executed, the database does not issue an update to all the records of the table. When a user selects the column for an existing record, the default value is obtained from the data dictionary and returned to the user.

## CHANGING COLUMN WIDTHS

The example below increases the width of the department name in the department table.

```
ALTER TABLE department  
ALTER COLUMN name [ SET DATA ] TYPE varchar(50);
```

The width of a character or numeric column may be increased at any time. The number of decimal places can be increased if the overall length is also increased and provided no data in the column would contravene the new definition.

Other reductions in size specification are possible, once again provided no data in the column concerned would breach the new definition

Additional ALTER COLUMN clauses can be added to the same ALTER TABLE statement if required, each comma separated from the previous.

## RENAMING COLUMNS

A column can be renamed.

### Example

```
ALTER TABLE department  
RENAME COLUMN name TO dept_name
```

## DROPPING COLUMNS

One or more columns can be dropped using the syntax show here.

```
ALTER TABLE table_name DROP COLUMN [IF EXISTS] column_name [CASCADE]
```

Adding **IF EXISTS** means that a notice will be returned rather than an error is an attempt is made to drop a non-existent column.

Adding **CASCADE** will cause referencing objects such as Views and Triggers to be dropped if the referenced column is dropped.

Multiple columns can be dropped in a single statement by adding extra **DROP COLUMN** clauses, each comma separated from the previous.

The example below drops a column immediately:

```
ALTER TABLE prospect DROP COLUMN credit_rating
```

This statement would drop two columns:

```
ALTER TABLE customer  
DROP COLUMN sales_manager_code,  
DROP COLUMN title;
```

## ADDING CONSTRAINTS

Table level syntax is used to add constraints to existing tables. For example:

```
ALTER TABLE product ADD CONSTRAINT psfkey  
FOREIGN KEY ( supplier_nr )  
REFERENCES supplier ( supplier_nr )  
ON DELETE CASCADE ;
```

Use **ALTER TABLE .. ALTER COLUMN** to add a **NOT NULL** constraint to an existing column as follows:

```
ALTER TABLE supplier  
ALTER COLUMN telephone SET NOT NULL ;
```

Remove a Not Null Constraint

To remove a NOT NULL constraint and render the column nullable use the following syntax:

```
ALTER TABLE supplier  
ALTER COLUMN telephone DROP NOT NULL ;
```

## LISTING CONSTRAINTS

A list of constraints can be displayed by querying the table **table\_constraints** in the **information\_schema** schema.

A useful query is indicated in the example below.

```
SELECT constraint_name, constraint_type
FROM information_schema.table_constraints
WHERE table_schema = 'public' AND
table_name = 'orders';
```

The column **constraint\_type** contains one of the following values:

- PRIMARY KEY
- UNIQUE
- FOREIGN KEY
- CHECK – for both NOT NULL and condition-based Check Constraints

The column *check\_clause* in the table **information\_schema.check\_constraints** specifies the check condition for a check constraint as shown in the example below.

```
SELECT *
FROM information_schema.check_constraints
WHERE constraint_schema = 'public'
AND constraint_name = 'employee_salary_check';
```

## COPYING TABLES IN POSTGRESQL

Tables can be copied or created by combining a **CREATE TABLE** statement with a **SELECT** statement.

The example below creates a duplicate customer table including all its rows:

```
CREATE TABLE customer2 AS  
SELECT * FROM customer
```

Note that **no** constraints are copied.

The next example creates a table containing a subset of the rows and columns in the supplier table:

```
CREATE TABLE supplier2 AS  
SELECT supplier_nr, name, telephone  
FROM supplier  
WHERE LOWER( town ) = 'london'
```

The final example creates an empty table from some of the columns of the product table:

```
CREATE TABLE product2 AS  
SELECT product_code, description, cost_price  
FROM product  
WHERE 0 = 1
```

## DROPPING TABLES

A table (and all its rows, columns and indexes) can be dropped using the **DROP TABLE** command as in the following example:

```
DROP TABLE [ IF EXISTS ] product2
```

Adding the IF EXISTS clause means that if an attempt is made to drop a table that does exist, no error will be raised but a notice message will be displayed instead.

The optional clause **CASCADE CONSTRAINTS** may be appended to this statement to drop all referential integrity constraints which refer to keys in the dropped table. A message will display details of constraints dropped due to the use of this clause.

Multiple tables can be dropped in one statement by providing a comma separated list of table names.

PostgreSQL does not offer any functions to recover dropped tables.



## WORKSHOP EXERCISES

Write SQL statements or scripts to achieve the following objectives.

1. Create a table called *book*. Specify the following columns:

- `book_id` (the primary key)
- `title`
- `author`

2. Create an employee dependants table (called *dependants*). Specify the following columns:

- `dependant number`
- `first name`
- `last name`
- `relationship to employee`
- `sex`
- `date of birth`
- `employee number`

Apply appropriate column (or table level) constraints. In particular ensure that the employee number is specified as a foreign key of the *employee* table.

*Optional additional questions (if time permits) on next page*

**OPTIONAL ADDITIONAL QUESTIONS (IF TIME PERMITS)**

3. Make a copy of the orders table (called *orders2*).  
Add a numeric column called *discount\_percent* to the table. The column must allow for discount rates up to 50% and must not accept **NULL** values.

4. Alter the table *orders2* so that the discount column is now nullable.

## SAMPLE ANSWERS

```
1.  CREATE TABLE book (
    book_id SMALLINT NOT NULL CONSTRAINT book_pkey PRIMARY KEY ,
    title   VARCHAR(100),
    author  VARCHAR(80) );
```

```
2.  CREATE TABLE dependants (
    dependant_nr SMALLINT CONSTRAINT depkey PRIMARY KEY ,
    first_name   VARCHAR(10) NOT NULL ,
    last_name    VARCHAR(20) NOT NULL ,
    relationship  VARCHAR(15) ,
    sex          CHAR CONSTRAINT depsex
                CHECK ( sex IN ( 'M', 'F', '?' ) ) ,
    birthdate    DATE ,
    employee_nr  SMALLINT CONSTRAINT fkey_emp
                REFERENCES employee ( employee_nr )
    );
```

```
3.  CREATE TABLE orders2 AS
      SELECT *
      FROM orders ;

ALTER TABLE orders2 ADD COLUMN
discount_percent NUMERIC( 4,2 ) DEFAULT 0
CONSTRAINT not_over_50 CHECK (discount_percent < 50) NOT NULL ;
```

```
4.  ALTER TABLE orders2 ALTER COLUMN discount_percent DROP NOT NULL ;
```



**Section 12**  
**MANAGING INDEXES**  
**AND VIEWS**



## MANAGING INDEXES AND VIEWS

In this section the following topics relating to Managing Indexes and Views will be covered:

- ◆ **Index Overview**
- ◆ **Create an Index**
- ◆ **Drop an Index**
- ◆ **List Indexes**
- ◆ **View Overview**
- ◆ **Create a View**
- ◆ **Alter a View**
- ◆ **List Views**

## INDEXES

### WHAT IS AN INDEX?

An index at the end of a book is an alphabetic sequence of keywords or topics and their associated page numbers. It is quicker and more efficient to refer to the index for a particular topic than to scan the content pages from beginning to end looking for the required information.

In similar fashion, an index on a table is a list of column values in alphabetic or numeric sequence together with the physical locations on disk of the rows which contain those column values.

While it is not necessary for a table to be indexed, an index can improve performance when retrieving data from the table. When executing a query any relevant indexes which exist for the tables on which the query is based will be used, if the optimizer estimates that it would be quicker to do so.

Indexes should normally be created for foreign keys and any other columns which may be used as join columns.

They may also be created on other columns frequently used in queries. For example, a customer table may often be queried by postcode (rather than customer number) so consideration could be given to creating an index on this column.

Indexes are automatically created for **PRIMARY KEY** columns and for columns defined with a **UNIQUE** constraint.



## SOME PERFORMANCE CONSIDERATIONS

PostgreSQL provides several index types: B-tree, Hash, GiST, SP-GiST, GIN and BRIN. There are also extensions that offer further index types. Each index type uses a different algorithm that is best suited to different types of data and queries.

By default, the `CREATE INDEX` command creates B-tree indexes, which fit the most common situations.

The information given below is not intended to be a full discussion of performance issues; it mainly concentrates on those situations where B-tree indexes are **not** used.

A B-tree index might not be used if:

- ◆ The optimizer estimates that a full table scan would be faster.
- ◆ The **WHERE** clause contains calculations or functions involving the indexed column (unless the index is a corresponding function-based index).
- ◆ The **!=** operator is used in the **WHERE** clause for the indexed column.
- ◆ The **LIKE** operator is used in the **WHERE** clause and the associated value starts with a wildcard character.

## MANAGING B-TREE INDEXES

B-Tree indexes are the most commonly used type and are the default type of index supported by PostgreSQL. The 'B' is for 'Balanced', indicating that index values will be balanced as evenly as possible across the tree so that the number of levels to be searched for a given index value is kept as low as possible.

### CREATING AN INDEX

The following SQL statement is used to create a B-Tree index:

```
CREATE [ UNIQUE ] INDEX name ON table-name ( column-list )  
      [ TABLESPACE tbname ]  
      [ other clauses possible ]
```

Where *name* is a user-defined name for the index, *table-name* is the name of the table on which the index is being created and *column-list* is a list of column names from which the index entries are to be constructed.

The keyword **UNIQUE** enforces unique values for each index entry.

If the **TABLESPACE** clause is omitted, the index is created in the tablespace specified by the configuration parameter **default\_tablespace**. If no value is set for **default\_tablespace**, the location of the database system catalogs is used.

Other clauses (such as the **STORAGE** clause) are also supported.

The index values are stored, by default, in ascending order according to the collation sequence defined for the column.

## Examples

The example below creates an index on the *department\_nr* column of the *employee* table:

```
CREATE INDEX deptno_idx ON employee ( department_nr )
```

The next example creates a compound (or composite) index on a combination of *surname* and *initials* for the *customer* table:

```
CREATE INDEX custname_idx ON customer ( surname, initials )
```

Index values can be held in descending sequence as shown in the example here:

```
CREATE INDEX empdate_idx ON employee ( start_date DESC )
```

Different sort orders can be specified for columns in a composite index. For example:

```
CREATE INDEX ord_qty_idx ON orders ( order_date DESC, quantity )
```

Indexes may be renamed using the following syntax:

```
ALTER INDEX ord_qty_idx RENAME TO sales_qty_idx;
```

## DROPPING INDEXES

Once created, indexes cannot be changed; they must first be dropped and then re-created. The SQL command used to drop an index is simply:

```
DROP INDEX name
```

Where ***name*** is the name of the index to be dropped; no table name needs to be specified.

## LISTING INDEXES

Metadata about user-defined indexes can be found in the system catalogs **pg\_class** and **pg\_index**.

```
SELECT    pc.relname, pc.relfilenode, pc.reltuples,
          pi.INDNATTS, pi.INDKEY
FROM      pg_class pc JOIN pg_index pi
ON        pi.indexrelid = pc.relfilenode
WHERE     pc.relname LIKE '%idx%'
OR        pc.relname LIKE '%pkey%';
```

This generates the following output:

relname	relfilenode	reltuples	indnatts	indkey
emp_pkey	16735	23	1	1
cus_pkey	16745	0	1	1
supp_pkey	16750	0	1	1
prod_pkey	16755	0	1	1
ord_pkey	16765	0	1	1
pros_pkey	16781	0	1	1
dependants_pkey	16941	0	1	1

Reltuples denotes the number of index values, Indnatts is the number of columns indexed, and indkey is the column number / s of the indexed column / s. These all relate to the index examples on the previous page.

In **psql**, you can issue the **\d+ tablename** command to see full details of a table, including information about its indexes:

```
Indexes:
    "emp_pkey" PRIMARY KEY, btree (employee_nr)
    "deptno_idx" btree (department_nr)
    "empdate_idx" btree (start_date DESC)
    "empyearsal_idx" btree ((salary * 12::numeric))
```

## FUNCTION-BASED INDEXES

Function-based indexes are designed to improve the performance of queries by defining an index to be used when the **WHERE** clause contains an equivalent operation.

For example, if the following index was created (n.b.: both sets of parentheses are needed)

```
CREATE INDEX empyearsal_idx ON employee (( salary * 12 ))
```

Then, in the SQL statement below, the *empyearsal\_idx* index may be used:

```
SELECT *  
FROM employee  
WHERE salary * 12 >= 30000
```

This index might be used where the WHERE clause applied the LOWER function to a town name:

```
CREATE INDEX prospect_town_idx ON prospect ( lower ( town ));
```

Certain privileges are required and certain system or session parameters need to be set in order to make use of function-based indexes. These are beyond the scope of this course and not discussed here.

## VIEWS

### WHAT IS A VIEW?

A view is a customised representation of one or more underlying (base) tables or other views (or a mixture of both).

Views are created for a number of reasons:

<b>Privacy</b>	It is not possible for the end-user to access columns and/or tables that are not included in the view.
<b>Readability</b>	Columns can be renamed as part of the view definition to make them more readily recognisable to the end-user.
<b>Ease of Use</b>	Frequently joined tables, in which all the join criteria are already specified, can be used as the basis of a view; hence simplifying subsequent query writing.
<b>Performance</b>	Using a standardised SQL statement can have performance benefits although the details of this are beyond the scope of this course.

Views are held on the database as **SELECT** statements on which they are based. They can be used for queries in exactly the same way as tables. On executing a query which refers to a view, the **SELECT** statement which forms the view is automatically executed.

View names must differ from other object names (table names, synonyms etc.) in a user's schema. Comments may be associated with views and their columns using the **COMMENT** statement described earlier. For example:

```
COMMENT ON VIEW stock_view IS 'This view is not updateable';
```

A comment but be removed by using **IS NULL**.

Views can be described using the **psql \d+** command.

## CREATING VIEWS

A view is created using the following SQL statement:

```
CREATE [ OR REPLACE ]  
VIEW view-name [ ( alias-list ) ]  
AS  
SELECT statement  
[ WITH CHECK OPTION ]
```

where ***view-name*** is a user-defined name for the view and ***alias-list*** is a comma-separated list of column names for the view.

If the alias list is omitted, the column names for the view are inherited from the column names (or aliases) specified in the **SELECT** statement.

If an alias list is specified then the number of names in the list must match the number of columns specified in the **SELECT** statement.

The clause **OR REPLACE** allows an existing view to be replaced.

The clause **WITH CHECK OPTION** specifies that inserts and updates done through the view should satisfy the **WHERE** clause of the **SELECT** statement.

## RESTRICTIONS

If the query uses an asterisk to select all columns of a table and new columns are later added to the table, then the view will not contain those columns until the view is recreated via the **CREATE OR REPLACE VIEW** statement.

## UPDATABLE VIEWS

An updatable view is one which may be used to insert, update or delete base table rows. *Only a single table at a time may be updated via an updatable view.*

A view is updateable if the query on which it is based derives data from a single table or another updateable view. For insert operations all the **NOT NULL** columns of the target table must be specified.

An attempt to insert via a View that has more than one table will result in this message:

```
ERROR:  cannot insert into view "stock_view"  
DETAIL:  Views that do not select from a single table or view are not automatically updatable.
```

In addition, the following constructs cannot be used for an updatable view:

- ◆ A **FROM** clause with **more than** one table (or other Updateable View)
- ◆ Any of the set operators **UNION**, **INTERSECT** or **EXCEPT**
- ◆ The **DISTINCT**, **LIMIT** or **OFFSET** operators
- ◆ An aggregate function such as **COUNT**, **AVG**, **MAX** or **SUM**
- ◆ A **GROUP BY** or **HAVING** clause
- ◆ Clauses associated with hierarchical queries (beyond the scope of this course)



## Examples

The example below creates a view based on the employee table. It is updateable but only for female employees so that the Check Option clause is met.

```
CREATE OR REPLACE VIEW fememps  
AS SELECT * from employee  
WHERE LOWER (sex) = 'f'  
WITH CHECK OPTION;
```

If an attempt to add a record in breach of the Check Option was made the following message is returned:

```
ERROR:  new row violates check option for view "fememps"  
DETAIL:  Failing row contains (3457, King, A.B. , M, 2018-02-06, Clerk, 1036, 1234.00, 20).  
SQL state: 44000
```

The view here includes values derived from an aggregate function, and has the associated **GROUP BY** clause and thus is not updatable:

```
CREATE OR REPLACE VIEW prodsales AS  
  SELECT p.product_code, p.description,  
         SUM( o.quantity ) AS total_sold  
  FROM   orders o, product p  
  WHERE  o.product_code = p.product_code  
  GROUP BY p.product_code, p.description
```

## DROPPING VIEWS

A view is deleted using the following SQL statement:

```
DROP VIEW view-name
```

If any of the underlying tables for a view are dropped the view itself should also be dropped.

## LISTING VIEWS

A view is stored in the Data Dictionary as the SELECT statement specified when it was created. To list your own views or those on which you have permissions, query the table ***information\_schema.views***.

```
SELECT is_updatable, view_definition
FROM information_schema.views
WHERE table_name = 'stock_view';
```

The output is easier to read using psql:

```
sat=# select is_updatable, view_definition
sat=# from information_schema.views
sat=# where table_name = 'stock_view';
 is_updatable | view_definition
-----+-----
NO            | SELECT s.name AS supplier,
              |     p.product_code,
              |     p.description,
              |     p.supplier_nr AS suppno,
              |     ((p.instock)::numeric * p.cost_price) AS stockval
              | FROM (product p
              | JOIN supplier s ON ((p.supplier_nr = s.supplier_nr)));
(1 row)
```

In psql, you can issue the \dv+ command to list views.

## WORKSHOP EXERCISES

Write SQL statements or scripts to achieve the following objectives.

1. Create an index on the `order_date` column of the `orders` table

2. Create an index on the `surname` and `initials` columns of the `customer` table

3. Create a view (called *emp\_view*) based upon the following data from the `employee` table:

- `employee_nr`
- `surname`
- `initials`
- `department_nr`

After creating the view, use a select statement to retrieve data from the view where the `department_nr` is 10

4. Create a view (called *sales\_revenue*) based upon the following data from existing tables:

- customer number
- customer surname
- total spend per customer

After creating the view, use a select statement to retrieve data from the view where the total spend for a customer exceeds £20000.



## SAMPLE ANSWERS

```
1.  CREATE INDEX orders_date_idx ON orders(order_date)
```

```
2.  CREATE INDEX customer_name_idx ON customer(surname,initials)
```

```
3.  CREATE OR REPLACE VIEW emp_view AS
      SELECT employee_nr, surname, initials, department_nr
      FROM   employee;

      SELECT *
      FROM   emp_view
      WHERE  department_nr = 10 ;
```

```
4.  CREATE OR REPLACE VIEW sales_revenue (custno, name, spend)
      AS
      SELECT c.customer_nr, c.surname,
      sum(o.quantity * p.sales_price)
      FROM   customer c JOIN orders o
      ON     c.customer_nr = o.customer_nr
      JOIN   product p
      ON     o.product_code = p.product_code
      GROUP BY c.customer_nr, c.surname
      ORDER BY 3 DESC;
```



**Section 13**  
**MANAGING SEQUENCES**





## MANAGING SEQUENCES

In this section the following topics relating to Managing Sequences will be covered:

- ♦ **Sequence Overview**
- ♦ **Create a Sequence**
- ♦ **Drop a Sequence**
- ♦ **List Sequences**

## SEQUENCES

A *sequence* is a database object used primarily for assigning incrementally unique numbers to columns and most often used with **INSERT** statements to generate unique numeric primary keys for a table.

A sequence is created using the following SQL statement:

```
CREATE [ TEMPORARY | TEMP ] SEQUENCE [ IF NOT EXISTS ] name
[ AS data_type ]
[ INCREMENT [ BY ] increment ]
[ MINVALUE minvalue | NO MINVALUE ] [ MAXVALUE maxvalue | NO
MAXVALUE ]
[ START [ WITH ] start ] [ CACHE cache ] [ [ NO ] CYCLE ]
[ OWNED BY { table_name.column_name | NONE } ]
```

**TEMP** or **TEMPORARY** mean that the sequence only endures for the current session. To refer to some other permanent sequence, its name would need to be prefixed with the relevant schema name.

**IF NOT EXISTS** means that if an attempt is made to create a sequence with the same name as some existing sequence, a notice will be displayed to that effect rather than an error being returned.

**AS datatype** allows the specification of a data type for the sequence. Valid types are smallint, integer and bigint. Bigint is the default.

The default **INCREMENT** is 1. A positive value will increment the sequence; a negative value will decrement the sequence.

The default **START WITH** value is **MINVALUE** (for incrementing sequences) or **MAXVALUE** (for decrementing sequences).

The default **MINVALUE** is 1. The default **MAXVALUE** is dictated by the datatype.

**CYCLE** allows the sequence to continue cycling; for an incrementing sequence, the cycle begins again at **MINVALUE** once **MAXVALUE** has been reached; and vice versa for a decrementing sequence. The default is **NOCYCLE**.

The **CACHE** value specifies how many sequence numbers must be kept in memory at any one time. The default is 1 i.e.: no values are cached.

**OWNED BY { table\_name.column\_name | NONE }** creates an association between the sequence and a specific column of a table. If the column in the table is subsequently dropped, then the sequence will also be dropped. However this association does not mean that the sequence cannot be used to generate a primary key value in some other table. The default for this clause is **OWNED BY NONE**.

## USING SEQUENCES

In SQL a reference to **NEXTVAL('sequence\_name')** retrieves the next value from the sequence and also updates the sequence.

The first time **NEXTVAL('sequence\_name')** is referenced after the sequence is created, the **START WITH** value is retrieved.

A reference to **CURRVAL('sequence\_name')** retrieves the current value of the sequence. If **CURRVAL('sequence\_name')** is referenced immediately after the sequence is created, an error is raised.

```
ERROR: currval of sequence "custnrseq" is not yet defined in this session
SQL state: 55000
```

PostgreSQL also has a **SETVAL** function that can be used to reset the next available sequence number.

```
SELECT SETVAL('custnrseq',3190);
```

This would mean that when the sequence was next referenced using **NEXTVAL**, the next available sequence number following 3190 would be retrieved, i.e.: 3191 assuming an increment of 1.

## Example

The following command creates a sequence (called *custnrseq*) which starts with 3000 and increments by 10 to a maximum value of 5000.

```
CREATE SEQUENCE custnrseq  
INCREMENT BY 10  
MINVALUE 3000 MAXVALUE 5000  
START WITH 3000 CACHE 20 NO CYCLE  
OWNED BY customer.customer_nr;
```

An initial reference to **NEXTVAL('custnrseq')** retrieves the value 3000.

The next reference to **NEXTVAL('custnrseq')** retrieves the value 3010 and so on.

An error results when the sequence value is 5000 and a further reference to **NEXTVAL('custnrseq')** is made.

## Example of Use

```
INSERT INTO customer (customer_nr, surname, initials)  
VALUES (NEXTVAL('custnrseq'), 'Fitzsimons', 'K');
```

Creates customer number 3000 on initial use of the sequence created above.

## DROPPING SEQUENCES

A sequence is dropped using the SQL statement:

```
DROP SEQUENCE name
```

## LISTING SEQUENCES

You can use the data dictionary table **information\_schema.sequences** to obtain information about a sequence.

```
SELECT *  
FROM information_schema.sequences  
WHERE sequence_name = 'custseq';
```

In psql, you can issue the \ds+ command to list sequences.

## PRACTICAL EXERCISES

1. Create a Sequence starting at 2000 and incrementing by 5, belonging to the customer\_nr column of the customer table.
2. Insert a new customer into the customer table using the sequence to generate a value for the primary key.
3. Check the value returned by the sequence using the CURRVAL function.
4. Insert a new prospect into the prospect table using the sequence to generate a value for prospect\_nr.
5. Reset the sequence to 2095. Insert a few new customer to the customer table. What primary key is assigned to the new customer?





## SAMPLE ANSWERS

```
1.  CREATE SEQUENCE custseq
      INCREMENT BY 5
      MINVALUE 2000 MAXVALUE 3000
      START WITH 2000
      OWNED BY customer.customer_nr;
```

```
2.  INSERT INTO customer (customer_nr, surname, initials)
      VALUES (NEXTVAL('custseq'), 'Jones', 'K');
```

```
3.  SELECT CURRVAL('custseq'); -- Returns 2000
```

```
4.  INSERT INTO prospect (prospect_nr, surname, initials, credit_rating)
      VALUES (NEXTVAL('custseq'), 'Singh', 'A', 12);
```

```
5.  SELECT SETVAL('custseq', 2095);

      INSERT INTO customer (customer_nr, surname, initials)
      VALUES (NEXTVAL('custseq'), 'Patel', 'G');

      SELECT CURRVAL('custseq'); -- returns 2100
```



**Appendix A**  
**DEMONSTRATION**  
**TABLES**



# TABLES USED ON THIS COURSE

## EMPLOYEE TABLE (employee)

employee_nr	<b>SMALLINT</b>	<b>NOT NULL</b>	Primary key
surname	<b>VARCHAR( 20 )</b>	<b>NOT NULL</b>	
initials	<b>CHAR( 6 )</b>		
sex	<b>CHAR</b>	<b>NOT NULL</b>	
start_date	<b>DATE</b>	<b>NOT NULL</b>	
job	<b>VARCHAR( 15 )</b>	<b>NOT NULL</b>	
manager	<b>SMALLINT</b>		
salary	<b>NUMERIC( 6,2 )</b>	<b>NOT NULL</b>	
department_nr	<b>SMALLINT</b>	<b>NOT NULL</b>	Foreign key of <b>department</b>

EMPLOYEE_NR	SURNAME	INITIALS	SEX	START_DATE	MANAGER	SALARY	DEPARTMENT_NR	JOB
1000	King	J.R.	M			5000	10	Chairman
1036	Blake	S.T.	M		1000	2850	30	Manager
1023	Clark	M.	F		1000	2450	10	Manager
1024	Jones	D.S.	M		1000	2975	20	Manager
1035	Martin	R.J.	M		1036	1250	30	Salesman
1033	Allen	P.G.	F		1023	2000	10	Salesman
1007	Turner	J.	M		1036	1500	30	Salesman
1019	Jameson	N.H.B.	F		1036	950	30	Clerk
1013	Ward	K.T.	M		1024	1250	20	Salesman
1026	Ford		M		1023	3000	10	Analyst
1029	Smith	J.	M		1036	800	30	Clerk
1022	Scott	B.L.	F		1024	3000	20	Analyst
1017	Adams	G.D.C.	F		1023	1100	10	Clerk
1030	Miller		F		1024	1300	20	Clerk

Note: Dates are calculated at installation but will be in the format 'yyyy-mm-dd'

**SALARY GRADE TABLE (salgrade)**

grade	<b>SMALLINT</b>	<b>NOT NULL</b>
low_salary	<b>NUMERIC( 6,2 )</b>	<b>NOT NULL</b>
high_salary	<b>NUMERIC( 6,2 )</b>	<b>NOT NULL</b>

<b>GRADE</b>	<b>LOW_SALARY</b>	<b>HIGH_SALARY</b>
1	700	1200
2	1201	1400
3	1401	2000
4	2001	3000
5	3001	9999

**DEPARTMENT TABLE (department)**

department_nr	<b>SMALLINT</b>	<b>NOT NULL</b>	Primary Key
name	<b>VARCHAR( 10 )</b>	<b>NOT NULL</b>	
location	<b>VARCHAR( 15 )</b>	<b>NOT NULL</b>	

<b>DEPARTMENT_NR</b>	<b>NAME</b>	<b>LOCATION</b>
10	Accounts	Birmingham
20	Research	Glasgow
30	Sales	Manchester
40	Operations	Birmingham

**CUSTOMER TABLE (customer)**

customer_nr	<b>SMALLINT</b>	<b>NOT NULL</b>	Primary key
surname	<b>VARCHAR( 20 )</b>	<b>NOT NULL</b>	
initials	<b>CHAR( 6 )</b>	<b>NOT NULL</b>	
street	<b>VARCHAR( 25 )</b>		
town	<b>VARCHAR( 25 )</b>		
county	<b>VARCHAR( 15 )</b>		
post_code	<b>CHAR( 8 )</b>		
telephone	<b>VARCHAR( 15 )</b>		
credit_limit	<b>NUMERIC( 6,2 )</b>		

CUSTOMER_NR	SURNAME	INITIALS	STREET	TOWN	COUNTY	POSTCODE	TELEPHONE	CREDIT_LIMIT
1317	Williams	R.	15, London Road	Manchester		M13 6TR	0161 254 1094	1000
1087	Knight	M.B.	23, Highbury Street	London		NW12 7FC	0181 905 6127	500
1019	Jameson	N.H.B.	12-14, Turner Road	Dudley	W.Midlands	B13 4DE	0161 912 3305	1250
1543	McDonald	S.W.	102, Feltham Court	Croydon	Surrey	CR8 7GH	0134 267209	1500
1822	Jones	W.J.	16, Alloa Crescent	Birmingham		B5 6HH	0121 234 8123	1000
1036	Blake	S.T.	67, Victoria Place	London		SW4 9TR	0171 210 3156	750
1443	Smith	T.	22, Gresham Court	Wivenhoe	Essex	CO15 7VD	01206 872090	750
1980	Green	L.M.	45, Leeds Road	London		W6 9HG	0171 710 1027	1250
1674	Jones	W.J.	90, Carter Street	Manchester		M15 8FD	0161 205 2206	1500
1223	Brown	K.	5, Windmill Place	Truro	Cornwall	TR4 2VX	01857 511089	500



## ORDERS TABLE (orders)

order_nr	<b>SMALLINT</b>	<b>NOT NULL</b>	Primary key
product_code	<b>CHAR( 10 )</b>	<b>NOT NULL</b>	Foreign key of <b>product</b>
customer_nr	<b>SMALLINT</b>	<b>NOT NULL</b>	Foreign key of <b>customer</b>
order_date	<b>DATE</b>		
quantity	<b>SMALLINT</b>	<b>NOT NULL</b>	

ORDER_NR	PRODUCT_CODE	CUSTOMER_NR	ORDER_DATE	QUANTITY
1000	MW97	1317		100
1001	HD12	1674		5
1002	DS96	1443		150
1003	MW97	1674		75
1004	HD12	1317		5
1005	MW97	1543		100
1006	TX10	1087		25
1007	AM17	1019		5
1008	XM21	1674		10
1009	HD12	1223		125
1010	GD2	1036		100
1011	DS96	1674		50
1012	TX10	1036		5
1013	MW97	1223		10
1014	HD12	1036		100
1015	AM17	1443		50
1016	DS96	1317		25
1017	HD12	1543		10
1018	GD2	1443		5
1019	MW97	1019		25
1020	XM21	1223		5
1021	GD2	1087		125
1022	DS96	1674		100
1023	HD12	1443		10

Note: Dates are calculated at installation but will be in the format 'yyyy-mm-dd'.

**PRODUCT TABLE (product)**

product_code	<b>CHAR( 10 )</b>	<b>NOT NULL</b>	Primary Key
description	<b>VARCHAR( 30 )</b>	<b>NOT NULL</b>	
supplier_nr	<b>SMALLINT</b>	<b>NOT NULL</b>	Foreign key of <b>supplier</b>
cost_price	<b>NUMERIC( 6,2 )</b>		
sales_price	<b>NUMERIC( 6,2 )</b>		
instock	<b>SMALLINT</b>		

<b>PRODUCT_CODE</b>	<b>DESCRIPTION</b>	<b>SUPPLIER_NR</b>	<b>COST_PRICE</b>	<b>SALES_PRICE</b>	<b>INSTOCK</b>
MW97	MicroWare Office 97	1034	230	305	450
FD35	PCK 1.44Mb Floppy Drive	1060	25	35	206
TX10	TrailBlaser Printer X10	1196	440	510	124
AM17	Activan 17" Monitor	1196	450	605	20
HD12	1.2GB Hard Disk Drive	1005	198	204	230
DS96	DataStore DBMS 96	1060	420	525	100
GD2	Graphix Draw 2.0	1034	120	165	203
XM21	Xvision 21" Monitor	1196	512	712	26
ASM2	Activan SuperMouse 2	1005	24	35	105

## SUPPLIER TABLE (supplier)

supplier_nr	<b>SMALLINT</b>	<b>NOT NULL</b>	Primary key
name	<b>VARCHAR( 30 )</b>	<b>NOT NULL</b>	
street	<b>VARCHAR( 25 )</b>		
town	<b>VARCHAR( 25 )</b>		
county	<b>VARCHAR( 15 )</b>		
post_code	<b>CHAR( 8 )</b>		
telephone	<b>VARCHAR( 15 )</b>		

SUPPLIER_NR	NAME	STREET	TOWN	COUNTY	POST_CODE	TELEPHONE
1034	Ace Software Ltd.	24, Dudley Road	Birmingham		B10 4JK	0121 356 7885
1060	Computer City Ltd	9, Highfield Court	London		N19 2RD	0181 556 2045
1005	The Hardware Shop	23-25 , High Street	Colchester	Essex	CO7 5LM	01206 308985
1196	London Computers Ltd.	143 Merchants Road	London		W4 7GH	
1078	PC Universe Ltd	27, Coventry Road	Bristol	Avon	BR2 6HS	01222 108873
1045	K.Brown	Compuhouse,High Street	Brighton	Sussex	BT5 9UK	01239 235666

**PROSPECT TABLE (prospect)**

prospect_nr	<b>SMALLINT</b>	<b>NOT NULL</b>	Primary key
surname	<b>VARCHAR( 20 )</b>	<b>NOT NULL</b>	
initials	<b>CHAR( 6 )</b>	<b>NOT NULL</b>	
street	<b>VARCHAR( 25 )</b>		
town	<b>VARCHAR( 25 )</b>		
county	<b>VARCHAR( 15 )</b>		
post_code	<b>CHAR( 8 )</b>		
telephone	<b>VARCHAR( 15 )</b>		
credit_rating	<b>SMALLINT</b>	<b>NOT NULL</b>	

PROSPECT_NR	SURNAME	INITIALS	STREET	TOWN	COUNTY	POST_CODE	TELEPHONE	CREDIT_RATING
1	Lamb	R.J.	15,London Road	Manchester		M13 6TR	0161 254 1094	10
2	Knight	M.B.	23,Highbury Street	London		NW12 7FC	0181 905 6127	5
3	Jameson	N.H.B.	12-14,Turner Road	Dudley	W.Midlands	B13 4DE	0161 912 3305	10
4	Hendrix	J.	102,Feltham Court	Croydon	Surrey	CR8 7GH	01314 267209	15
5	Jones	L.R.	16, Alloa Crescent	Birmingham		B5 6HH	0121 234 8123	10
6	Smith	M.	67,Victoria Place	London		SW4 9TR	0171 210 3156	7
7	Smith	T.	22 Gresham Court	Wivenhoe	Essex	CO15 7VD	01206 872090	7
8	White	K.R.	45,Leeds Road	London		W6 9HG	0171 710 1027	10
9	Jones	W.J.	90,Carter Street	Manchester		M15 8FD	0161 205 2206	15
10	Botham	S.P.	5,Windmill Place	Truro	Cornwall	TR4 2VX	01857 511089	5
11	Williams	R.	15,London Road	Manchester		M13 6TR	0161 254 1094	10
12	Watkins	D.R.L.	23,Highbury Street	London		NW12 7FC	0181 905 6127	5
13	Fford	H.C.C.	12-14,Turner Road	Dudley	W.Midlands	B13 4DE	0161 912 3305	10
14	McDonald	S.W.	102,Feltham Court	Croydon	Surrey	CR8 7GH	01314 267209	15
15	Jones	P.	16,Alloa Crescent	Birmingham		B5 6HH	0121 234 8123	10
16	Blake	S.T.	67,Victoria Place	London		SW4 9TR	0171 210 3156	7
17	Allen	B.C.	22,Gresham Court	Wivenhoe	Essex	CO15 7VD	01206 872090	7
18	Green	L.M.	45 Leeds Road	London		W6 9HG	0171 710 1027	12
19	Middleton	S.	90 Carter Street	Manchester		M15 8FD	0161 205 2206	15
20	Brown	K.	5,Windmill Place	Truro	Cornwall	TR4 2VX	01857 511089	5
21	Channon	M.	124 Causton Road	Ipswich	Suffolk	IP3 6FD	01420 340912	7

**Appendix B**

**QUICK  
REFERENCE**



## SELECT STATEMENT

```
SELECT [ DISTINCT ] * | comma-separated-column/expression-list
      FROM comma-separated-table-or-view-list
      [ WHERE conditions-linked-with-AND/OR ]
      [ GROUP BY comma-separated-column-list ]
      [ HAVING conditions-linked-with-AND/OR ]
      [ ORDER BY comma-separated-column-list-with-optional-ASC-or-DESC]
```

## DML STATEMENTS

```
INSERT INTO table-or-view-name
      [ (comma-separated-column-list) ]
      VALUES (comma-separated-value-list)

UPDATE table-or-view-name
      SET column-name = value
          [ column-name = value ... ]
      [ WHERE conditions-linked-with-AND/OR ]

DELETE FROM table-or-view-name
      [ WHERE conditions-linked-with-AND/OR ]
```

## AGGREGATE FUNCTIONS

```
SUM(column-name)
MIN(column-name)
MAX(column-name)
AVG(column-name)
COUNT(*)
COUNT(column-name)
```

## CONDITIONS

```
=
<
>
<=
>=
<>
[ NOT ] BETWEEN value1 AND value2
[ NOT ] IN ( comma-separated-list-of-values)
[ NOT ] LIKE value-with-wildcards
IS [ NOT ] NULL
```

## DDL STATEMENTS

```
CREATE TABLE table-name
    ( column-name data-type [ column-constraint ] [ DEFAULT value ]
    [, further-column-definitions ...]
    [, constraint-definitions ]
    )
```

```
CREATE TABLE table-name
AS select-statement
```

```
DROP TABLE table-name
```

```
CREATE [ UNIQUE ] INDEX index-name
    ON table-name ( comma-separated-column-list )
```

```
DROP INDEX index-name
```

```
CREATE VIEW view-name [ comma-separated-list-of-column-names ]
AS select-statement
```

## CONSTRAINTS

```
CONSTRAINT constraint-name PRIMARY KEY (comma-separated-column-list)
```

```
CONSTRAINT constraint-name UNIQUE ( comma-separated-column-list )
```

```
CONSTRAINT constraint-name FOREIGN KEY (comma-separated-column-list)
    REFERENCES parent-table-name [( comma-separated-column-list )]
    ON DELETE RESTRICT | CASCADE | SET NULL
```







**StayAhead Training Limited**

**020 7600 6116**

**[www.stayahead.com](http://www.stayahead.com)**