实验 1　PostgreSQL 数据库 SQL 语句练习实验

实验类别：验证性　　实验级别：必做

开课单位：计算机与软件学院　　实验时数：12 学时

一、实验目的：

1、 了解 DBMS 系统的功能、软件组成；

2、掌握利用 SQL 语句定义、操纵数据库的方法。

二、实验要求：

1、在课外安装相关软件并浏览软件自带的帮助文件和功能菜单，了解 DBMS 的功能、结构；

2、创建一个有两个关系表的数据库;

3、数据库、关系表定义；

4、学习定义关系表的约束(主键、外键、自定义)；

5、了解 SQL 的数据定义功能；

6、了解 SQL 的操纵功能；

7、 掌握典型的 SQL 语句的功能；

8、 了解视图的概念；

三、实验设备：

计算机、数据库管理系统等软件。

四、建议的实验步骤：

0、安装 PostgreSQL 软件。见附件 1

1、使用 SQL 语句建立关系数据库模式及数据如下；（注：**数据要自己输入**）

EMP:

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	17-Dec-90	13750		20
7499	ALLEN	SALESMAN	7698	20-FEB-89	19000	6400	30
7521	WARD	SALESMAN	7698	22-FEB-93	18500	4250	30
7566	JONES	MANAGER	7839	02-APR-89	26850		20
7654	MARTIN	SALESMAN	7698	28-SEP-97	15675	3500	30
7698	BLAKE	MANAGER	7839	01-MAY-90	24000		30
7782	CLARK	MANAGER	7839	09-JUN-88	27500		10
7788	SCOTT	ANALYST	7566	19-APR-87	19500		20
7839	KING	PRESIDENT		17-NOV-83	82500		10
7844	TURNER	SALESMAN	7698	08-SEP-92	18500	6250	30
7876	ADAMS	CLERK	7788	23-MAY-96	11900		20
7900	JAMES	CLERK	7698	03-DEC-95	12500		30
7902	FORD	ANALYST	7566	03-DEC-91	21500		20
7934	MILLER	CLERK	7782	23-JAN-95	13250		10
3258	GREEN	SALESMAN	4422	24-Jul-95	18500	2750	50
4422	STEVENS	MANAGER	7839	14-Jan-94	24750		50
6548	BARNES	CLERK	4422	16-Jan-95	11950		50

DEPT:

DEPTNO	DNAME	LOC
10	ACCOUNTING	LONDON
20	RESEARCH	PRESTON
30	SALES	LIVERPOOL
40	OPERATIONS	STAFFORD
50	MARKETING	LUTON

2、用 SQL 定义数据库的关系表；

注：（每位同学在各自创建的图表名字后面添加自己学号以示区分，如 EMP20170000112 等）

3、定义各个关系的字段和自定义的数据完整性约束；

4、确定关系表的主键、外键；

5、对照帮助文件和教材理解主键和外键的约束规则；

6、分别为关系表添加记录；

7、理解 SQL 语句和关系运算的关系；

8、练习典型的 SQL 语句，对第 6 步实验中已建立的表做查询、插入、更新、删除等操作；完成练习题。

9、检查同学作业的思考题是否存在问题。

注：以上具体步骤可参见帮助文件 SQL handbook 或相关书籍。

五、 实验 1 实验报告要求格式：

1、实验目的：

2、实验时间：

3、完成的实验内容：

4、实验设备和实验环境：

5、实验结果和结论（运行的结果）：

6、练习题：SQL Handbook 练习题 **EX1—EX7**，**老师会在实验课上选取难度较高的一些题目写入试验报告**。

提交报告的详细内容和模板，请见试验报告模板。

附件一、CentOS7 内 PostgreSQL 安装

rpm 安装

目前的 Linux 7 的发行版本已经集成了 PostgreSQL 的相关套件，只需要配置好 yum 源既可以使用 yum 软件安装命令安装，系统自带的 postgresql 的版本为 9.2 版本。

1 配置本地 yum

[root@sdedu ~]# cat /etc/yum.repos.d/local.repo

[local]

name=local

baseurl=file:///mnt

enabled=1

gpgcheck=0

2 更新 yum 源

[root@sdedu ~]# yum clean all && yum repolist all

3 检查 PostgreSQL 相关软件套件

[root@sdedu yum.repos.d]# yum list all | grep ^postgresql

4 执行 yum 安装

[root@sdedu ~]# yum install -y postgresql-server.x86_64

使用 yum 安装后，默认的命令路径位于/usr/bin 下

5 初始化数据目录

#创建初始化数据目录

[root@sdedu ~]# mkdir -p /pg92/pgdata/data/

#为数据目录授权 postgres 用户所属主和所属组

[root@sdedu ~]# chown postgres.postgres -R /pg92

#切换至 postgres 用户

[root@sdedu ~]# su – postgres

#执行 初始化数据库命令

[postgres@sdedu ~]$ /usr/bin/initdb -D /pg92/pgdata/data/

6 启动数据库

[postgres@sdedu ~]$ /usr/bin/pg_ctl start -D /pg92/pgdata/data/ -l /tmp/logfile

server starting

7 配置服务开机启动

[root@sdedu ~]# systemctl enable postgresql.service

Yum 安装配置相对于简单，可以作为学习入门

8 拓展

除了使用操作系统本身集成的 postgresql 软件套件外，也可以通过官方网站获取官方集成的 rpm 软件套件，下去可自行尝试，在 Linux 下下载必须要保证网络畅通。

具体下载地址：

https://download.postgresql.org/pub/repos/yum/reporpms/EL-7-x86_64/pgdg-redhat-repo-latest.noarch.rpm

下载官方提供的 rpm 源配置包

然后使用 rpm 安装该包

[root@sdedu ~]# rpm -ivh pgdg-redhat-repo-latest.noarch.rpm

或者直接使用 yum 进行安装

[root@sdedu ~]# yum install https://download.postgresql.org/pub/repos/yum/reporpms/EL-7-x86_64/pgdg-redhat-repo-latest.noarch.rpm

然后执行 yum 安装客户端

[root@sdedu ~]# yum install postgresql12

执行 yum 安装数据库服务端

[root@sdedu ~]# yum install postgresql12-server

初始化数据目录

[root@sdedu ~]# /usr/pgsql-12/bin/postgresql-12-setup initdb

通过 systemctl 启动数据库和配置开机启动

[root@sdedu ~]# systemctl start postgresql-12

[root@sdedu ~]# systemctl enable postgresql-12

启动数据库

su – postgres

用“Postgres”用户登录

-bash-4.2$ psql -U postgres

#进入数据库后修改密码；

postgres=# alter user postgres with password '这里填密码'

附件二、SQLhandbook

CONTENTS

This book has been designed to help you learn SQL as it has to be learnt by doing, not by teaching.

It is therefore in your best interest to work your way through it (10-15 hours work) systematically.

You may be asked to submit some of the answers to the exercises, for an assignment and may be asked specific details in an exam.

SQL (Structured Query Language) is a relational database language. Amongst other things the language consists of statements to insert, update, query and protect data. Although SQL is not a DBMS, for simplicity in this manual SQL will be considered as a DBMS as well as a language. Of course, in the places where it is necessary, a distinction will be drawn.

There are a few things to note about SQL as a database language, because it is a relational database language, SQL may be grouped with the non-procedural database languages. By non-procedural it is meant that users (with the help of the various statements) have only to specify which data they want and not how this data must be found. C++, Java and VB are examples of Procedural languages. It also means that there are no variables, IF statements or loop constructs. Because it is non procedural, it is very difficult to teach, and the only way to learn it is by working through this book and picking up how certain results can be achieved.

SQL can be used in two ways. First, interpretively: an SQL statement is entered at a terminal or PC and immediately processed or interpreted. The result is also visible immediately. This is known as interactive SQL. The second way is known as embedded SQL. The SQL statements are embedded in a program written in another, procedural language. Results of these statements are not immediately visible to the user, but are processed by the 'enveloping' program. In this module we shall be

assuming the interpretive use of SQL.

SQL has already been implemented by many manufacturers as the database language for their DBMS. It is not the case, therefore, that SQL is the name of a particular manufacturer's product available on the market today. However, it is the market standard and you will find many career opportunities within the general SQL field. Currently it is number one in the Jobs market.

Some manufacturers are now providing SQL-server machines. These machines can be connected to a DBMS, and they carry out all the database functions defined in SQL. Thus SQL is now a data interchange language between any systems that can 'speak' SQL. Typically, an SQL-server is placed on a LAN where it processes all database operations for clients on the LAN.

Please note that, although SQL is an ISO standard, each manufacturer have their own add-ons.

SELECTING DATA FROM TABLES.

The SELECT Command is the basis of all queries on tables, therefore its full description is given to show its power. Examples of the various formats are provided after the description.

```
SELECT
   column_1, column_2, ...
FROM
   table_1
[INNER | LEFT |RIGHT] JOIN table_2 ON conditions
WHERE
   conditions
GROUP BY column_1
HAVING group_conditions
ORDER BY column_1
LIMIT offset, length;
```

The `SELECT` statement consists of several clauses as explained in the following list:

- SELECT followed by a list of comma-separated columns or an asterisk (*) to indicate that you want to return all columns.
- FROM specifies the table or view where you want to query the data.
- JOIN gets related data from other tables based on specific join conditions.
- WHERE clause filters row in the result set.
- GROUP BY clause groups a set of rows into groups and applies aggregate functions on each group.
- HAVING clause filters group based on groups defined by GROUP BY clause.
- ORDER BY clause specifies a list of columns for sorting.
- LIMIT constrains the number of returned rows.

The `SELECT` and `FROM` clauses are required in the statement.

Description: Selects rows and columns from one or more tables. May be used as a command, or (with certain restrictions on Clauses) as a subquery in another **SELECT**, and **UPDATE,** or other **SQL** command.

Don't worry too much about this generic syntax list as you will see all kinds of examples throughout this book.

PARAMETERS AND CLAUSES.

ALL makes **SELECT** display all rows produced by the query. Since this is the default, it is generally not needed

DISTINCT makes it omit duplicate rows.

***** makes **SELECT** display all columns of the table(s) specified by **FROM,** in the order they were defined when the table(s) were created.

i.e. **SELECT * FROM EMP;**

Alternatively, each expression becomes one column in the display.

i.e. **SELECT EMPNO, ENAME FROM EMP;**
displays only the named columns in the expression.

Each alias, if specified, is used to label the preceding expression in the displayed table.

e.g. SELECT ENAME "Name", SAL "Salary" from EMP;

Note the use of double quotes here. character strings are delimited by single quotes.

FROM table specifies the table or view to be drawn on. More than one table implies a join. **Alias,** if specified, may be used as an alias for the preceding table through the rest of the **SELECT** command.

SELECTED EXAMPLES AND WORKSHEETS.

The examples in this book should be worked through carefully to ensure that you understand what the commands are doing. Your assignment work will assume that knowledge.

You will need the following tables, EMP and DEPT.

These can be created with the following commands:

EPT;

The data in them is currently as shown on the next page:

Note - if the table contents become corrupted (particularly after the Update example in Exercise 7), you can always delete the tables and start again. This can be achieved by:

```
DROP TABLE EMP;
DROP TABLE DEPT;
```

To list all the tables in your Oracle area:

```
SHOW TABLES;
```

THE DATA USED IN THESE EXERCISES:

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	17-DEC-90	13750		20
7499	ALLEN	SALESMAN	7698	20-FEB-89	19000	6400	30
7521	WARD	SALESMAN	7698	22-FEB-93	18500	4250	30
7566	JONES	MANAGER	7839	02-APR-89	26850		20
7654	MARTIN	SALESMAN	7698	28-SEP-97	15675	3500	30
7698	BLAKE	MANAGER	7839	01-MAY-90	24000		30
7782	CLARK	MANAGER	7839	09-JUN-88	27500		10
7788	SCOTT	ANALYST	7566	19-APR-87	19500		20
7839	KING	PRESIDENT		17-NOV-83	82500		10
7844	TURNER	SALESMAN	7698	08-SEP-92	18500	6250	30
7876	ADAMS	CLERK	7788	23-MAY-96	11900		20
7900	JAMES	CLERK	7698	03-DEC-95	12500		30
7902	FORD	ANALYST	7566	03-DEC-91	21500		20
7934	MILLER	CLERK	7782	23-JAN-95	13250		10
3258	GREEN	SALESMAN	4422	24-JUL-95	18500	2750	50
4422	STEVENS	MANAGER	7839	14-JAN-94	24750		50
6548	BARNES	CLERK	4422	16-JAN-95	11950		50

DEPTNO	DNAME	LOC
10	ACCOUNTING	LONDON
20	RESEARCH	PRESTON
30	SALES	LIVERPOOL
40	OPERATIONS	STAFFORD
50	MARKETING	LUTON

THE SELECT STATEMENT

The **SELECT** statement is the workhorse of query processing the basic statement is:-

SELECT COLUMN(S) **FROM** TABLENAME;

This is the minimum amount of detail which must be entered for a **SELECT** statement to work.

Try the following examples:-

SELECT * FROM emp;

Provides a listing of all the data (all columns) in the EMP table.

SELECT ename FROM emp;

Gives a list of all the employee names found in the emp table.

SELECT dname, loc FROM dept;

gives department names and locations.

SELECT job FROM emp; (with duplicates)

Lists all the jobs in the emp table even if they appear more than once.

SELECT DISTINCT job FROM emp; (without duplicates)

List all the jobs in the EMP table eliminating duplicates.

SELECT job, deptno FROM emp; (with duplicates)

Lists the combination of jobs and departments for every row of the emp table.

SELECT DISTINCT job, deptno FROM emp; (without duplicates)

List all the combinations of job and department in the EMP table eliminating duplicates.

THE WHERE CLAUSE

A **WHERE** clause causes a 'search' to be made and only those rows that meet the search condition are retrieved.
A **WHERE** clause condition can use any of the following comparison operators:-

= equal to

```
SELECT * FROM emp
    WHERE ename = 'JONES';
```

(Again note that the data is case sensitive, this would not find Jones)

!= not equal to

^= not equal to

<> not equal to

```
SELECT * FROM emp
    WHERE ename != 'FORD';
```

> greater than

```
SELECT * FROM emp
    WHERE sal > 15000;
```

>= greater than or equal to

```
SELECT * FROM    emp
    WHERE sal >= 12000;
```

< less than

```
SELECT * FROM emp
    WHERE sal < 15000;
```

<= less than or equal to

```
SELECT * FROM emp
    WHERE sal <= 12000;
```

or special SQL operators

BETWEEN low AND high (values are inclusive)

```
SELECT * FROM emp
    WHERE sal BETWEEN 10000 AND 15000;
```

IN (VALUE1, VALUE2, VALUE3.....) character strings must be enclosed in quotes

```
SELECT * FROM emp
    WHERE job IN ('CLERK', 'ANALYST');
```

Selects all employees who are

Clerks or analysts

LIKE 'string picture' use '%' and '_' as wildcards within a string picture. Each _ acts for one character.

```
SELECT * FROM emp
    WHERE ename LIKE '%A%';
```

% is for any number of characters

Selects all employees with an 'A'

in their name.

IS NULL IS may only be used with NULL's (this means the variable has no value)

and also **NOT** any of the above expressions (used for negation purposes).

Try the following:-

```
SELECT ename, empno, deptno
    FROM emp
    WHERE job = 'CLERK';
```

List the names, numbers and departments of all the Clerks.

```
SELECT ename, sal, comm FROM emp
    WHERE  comm > sal;
```

Find the employees whose commission is greater than their salary.

SELECT ename, job, sal FROM emp
WHERE sal BETWEEN 12000 AND 14000;

Finds all employees who earn between 12,000 and 14,000

Selecting rows within a range, the WHERE clause can have a low-value and a high-value associated with it, these values represent the bottom and top of the required range.

NOT BETWEEN means that only rows that are outside the range will be selected.

SELECT ename FROM emp
WHERE job IN ('CLERK', 'ANALYST', 'SALESMAN');

Finds the employees who are clerks, analysts or salesmen.
NOT IN would list those employees whose jobs are not in the list.

SELECT ename FROM emp
WHERE job NOT IN ('CLERK', 'ANALYST', 'SALESMAN');

SELECT ename, deptno FROM emp
WHERE ename = 'FORD';

Finds the departments that employees called Ford work in.

SELECT ename, deptno FROM emp
WHERE ename LIKE '__A%';

Finds employee names that have an A as the 3rd letter i.e. Blake, Clark etc. (Note - there are 2 underscores before the A)

SELECT ename FROM emp
WHERE comm IS NULL;

Finds all employees that do not have any commission

Multiple search conditions may be used in a select statement, linked by either **AND** (both statements must be true for a row to be selected) or **OR** (only one condition

must be true for a row to be selected) **AND** and **OR** may be combined to produce complex search conditions and for clarity and reliability should be parenthesised to force precedence. Otherwise normal computing rules apply.

```
SELECT * FROM emp
    WHERE job = 'MANAGER'
    OR job = 'CLERK'
    AND deptno = 10;
```

Find everyone whose job title is manager, and all the clerks in department 10

```
SELECT * FROM emp
    WHERE job = 'MANAGER'
    OR( job = 'CLERK'
    AND deptno = 10);          (use of parentheses to clarify.)
```

```
SELECT * FROM emp
    WHERE (job = 'MANAGER'
    OR job = 'CLERK')
    AND deptno = 10;
```

Find all the managers or clerks in department 10.

Any group of search conditions can be negated by enclosing the statement in parentheses and preceding them with NOT.

```
SELECT * FROM emp
    WHERE NOT (job = 'MANAGER'
    OR job = 'CLERK')
    AND deptno = 10;
```

Find anyone who is neither a manager nor a clerk but is in department 10.

THE ORDER BY CLAUSE

By default Oracle will display rows of data in a totally unordered way. The **ORDER BY** clause should be used to impose an ordering of the rows retrieved by a query and should always be placed last in the query (or query block).

The use of **ORDER BY** causes data to be sorted (by default) as follows:-

NUMERICS ascending order by value

DATES chronological order

CHAR alphabetically

The keyword **DESC** causes the sort to be reversed.
NULL values in a sorted column will always be sorted high, i.e. they will be first when values are sorted in descending order and last when sorted in ascending order.

```
SELECT empno, ename, hiredate FROM emp
    ORDER BY hiredate;
```

Shows details of employees with earliest hiredates first.

```
SELECT job, sal, ename FROM emp
    ORDER BY job, sal DESC;
```

To order all employees by job, and within job, put them in descending salary order;

```
SELECT ename, job, sal, comm, deptno FROM emp
    ORDER BY 3;
```

Lists employees in salary order (salary is the 3rd item in the SELECT list)

EXERCISES. 1 SIMPLE COMMANDS

1 List all information about the employees.

2 List all information about the departments

3 List only the following information from the EMP table (Employee name, employee number, salary, department number)

4 List details of employees in departments 10 and 30.

5 List all the jobs in the EMP table eliminating duplicates.

6. What are the names of the employees who earn less than £20,000?

7. What is the name, job title and employee number of the person in department 20 who earns more than £25000?

8. Find all employees whose job is either Clerk or Salesman.

9. Find any Clerk who is not in department 10.

10. Find everyone whose job is Salesman and all the Analysts in department 20.

11. Find all the employees who earn between £15,000 and £20,000. Show the employee name, department and salary.

12 Find the name of the President.

13 Find all the employees whose last names end with S

14 List the employees whose names have TH or LL in them

15 List only those employees who receive commission.

16 Find the name, job, salary, hiredate, and department number of all employees by alphabetical order of name.

17. Find the name, job, salary, hiredate and department number of all employees in ascending order by their salaries.

18. List all salesmen in descending order by commission divided by their

salary.

19. Order employees in department 30 who receive commision, in ascending order by commission

20 Find the names, jobs, salaries and commissions of all employees who do not have managers.

21 Find all the salesmen in department 30 who have a salary greater than or equal to £18000.

JOINING TABLES

It is necessary to join two or more tables for some queries. This takes place by establishing a relationship (usually equality) between a column (domain) present in two tables known as a foreign key. Simple joins are usually called **equi-joins.** A join is automatically performed when a reference is made to more than one table in the **FROM** clause.

```
SELECT ename, sal, loc FROM emp, dept
    WHERE ename = 'ALLEN'       (search condition)
    AND emp.deptno = dept.deptno;       ((join condition)
```

Find Allen's name and salary from the EMP table and location of Allen's department from the DEPT table.

N.B. because we are now referencing two tables which each have a column with the same name (deptno), we must always qualify deptno with its table name in order to prevent confusion, this qualification must be used whenever ambiguous column names are used within an SQL statement.

```
SELECT  ename, dname FROM emp, dept
    WHERE emp.deptno = dept.deptno
    ORDER BY ename;
```

List the name and department of all employees in name order. (This joins the two tables over DEPTNO and projects out ENAME and DNAME)

Abbreviating Table Names.

Table names can be abbreviated in order to simplify what is typed in with the query. In this example E and D are abbreviated names for emp and dept.

List the department name and all employee data for employees that work in Chicago;

```
SELECT dname, E.*  FROM emp E, dept D
    WHERE E.deptno = D.deptno AND loc = 'LUTON'
        ORDER BY E.deptno;
```

Note – if we didn't have ORDER BY E.deptno, but had ORDER BY deptno We would get a syntax error because it would know whether to sort on the deptno in Emp or Dept.

Joining a Table to Itself

A table label can be used for more than just abbreviating a table name in a query. It also allows a join of a table to itself as though it were two separate tables. This can be very useful because a single SELECT will only go through a table once. By having two copies of the same table, you can find a specific record in the first copy and then search the second copy for comparisons.

```
SELECT WORKER.ename, WORKER.sal
    FROM emp WORKER, emp MANAGER
    WHERE WORKER.mgr = MANAGER.empno
    AND WORKER.sal > MANAGER.sal;
```

In the query the **emp** table is treated as if it were two separate tables named **WORKER** and **MANAGER.**

First all the **WORKERS** are joined to their **MANAGERS** using the **WORKER**'s manager's employee number (**WORKER.mgr**) and the **MANAGER's** employee number (**MANAGER.empno).**

The **WHERE** clause eliminates all **WORKER MANAGER** pairs except those where the **WORKER** earns more than the manager (**WORKER.SAL >MANAGER.SAL).**

Find all employees that earn more than Jones.

```
SELECT X.ename, X.sal, X.job, Y.job, Y.ename, Y.sal
    FROM emp X, emp Y
        WHERE X.sal > Y.sal
        AND Y.ename = 'JONES';
```

i.e. find JONES, and then go through the table again comparing.

Selecting all possible combinations of rows.

If the **WHERE** clause contains no join condition, then all possible combinations of rows from tables listed in the from clause are displayed. The result (Cartesian product) is normally not desired so a join condition is usually specified.
This is a common error, and to be avoided because if table A has 20 rows and table B has 30 rows then not using a join would result in 600 output lines.

Join the Allen row from the EMP table with all the rows in the Dept table

SELECT ename, loc FROM emp, dept
WHERE ename = 'ALLEN';

OUTER JOINS

When processing joins between emp and dept you will notice that details of department 40 never appear in the output. This is because department 40 has no corresponding rows in the emp table and therefore cannot take part in the join. If it is required to include records which are outside of the relationship between tables an ***outer join*** must be used.

SELECT dept.deptno, dname, ename, sal from dept left outer join emp on dept.deptno = emp.deptno

The **left 【outer】 join** effectively adds a dummy row to the emp table for each department record which has no corresponding employees. The department record is then joined with this dummy row and appears once in the output, having nulls in any columns from the emp table.

EXERCISES 2 JOINS

1. Find the name and salary of employees in Luton.

2. Join the DEPT table to the EMP table and show in department number order.

3. List the names of all salesmen who work in SALES

4. List all departments that do not have any employees.

5 For each employee whose salary exceeds his manager's salary, list the employee's name and salary and the manager's name and salary.

6. List the employees who have BLAKE as their manager.

SQL FUNCTIONS

SQL*PLUS has a wide range of functions which may be applied to Oracle data. There are four classes of functions:-

string functions for searching and manipulating strings.
arithmetic functions for performing calculations on numeric values
date functions for reformatting and performing data arithmetic
aggregate functions for calculations on groups of data.

Useful string functions

NOTE - when you wish to Select something, but the data is not in a table (as the examples below), you can use a dummy table name called DUAL. This table is only recognised by Mysql as a dummy table, and will never appear as an actual structure. MySQL may ignore the clauses. MySQL does not require FROM DUAL if no tables are referenced.

LOWER(string) converts upper case alphabetic characters to lower case. Other characters are not affected

SELECT LOWER ('MR. SAMUEL HILLHOUSE')
FROM DUAL;
gives mr samuel hillhouse

UPPER(string) converts lowercase letters in a string to uppercase.

SELECT UPPER ('Mr . Rodgers') FROM DUAL;

SUBSTR(string,startposition,length) shows a part of the string starting at the start position of the specified length

SELECT SUBSTR('ABCDEF',2,3) FROM dual;
gives BCD

INSTR(string1,string2) finds the start position of one string inside another string

SELECT INSTR('ABCDEF', 'DEF') FROM dual;

gives 4

str_to_date (string[,format]) converts the string to a date. A format may optionally be specified (see later)

LPAD(str,len,padstr) left pads the string with the specified fill characters to the specified length.

SELECT LPAD('hi',4,'??');

gives '??hi'

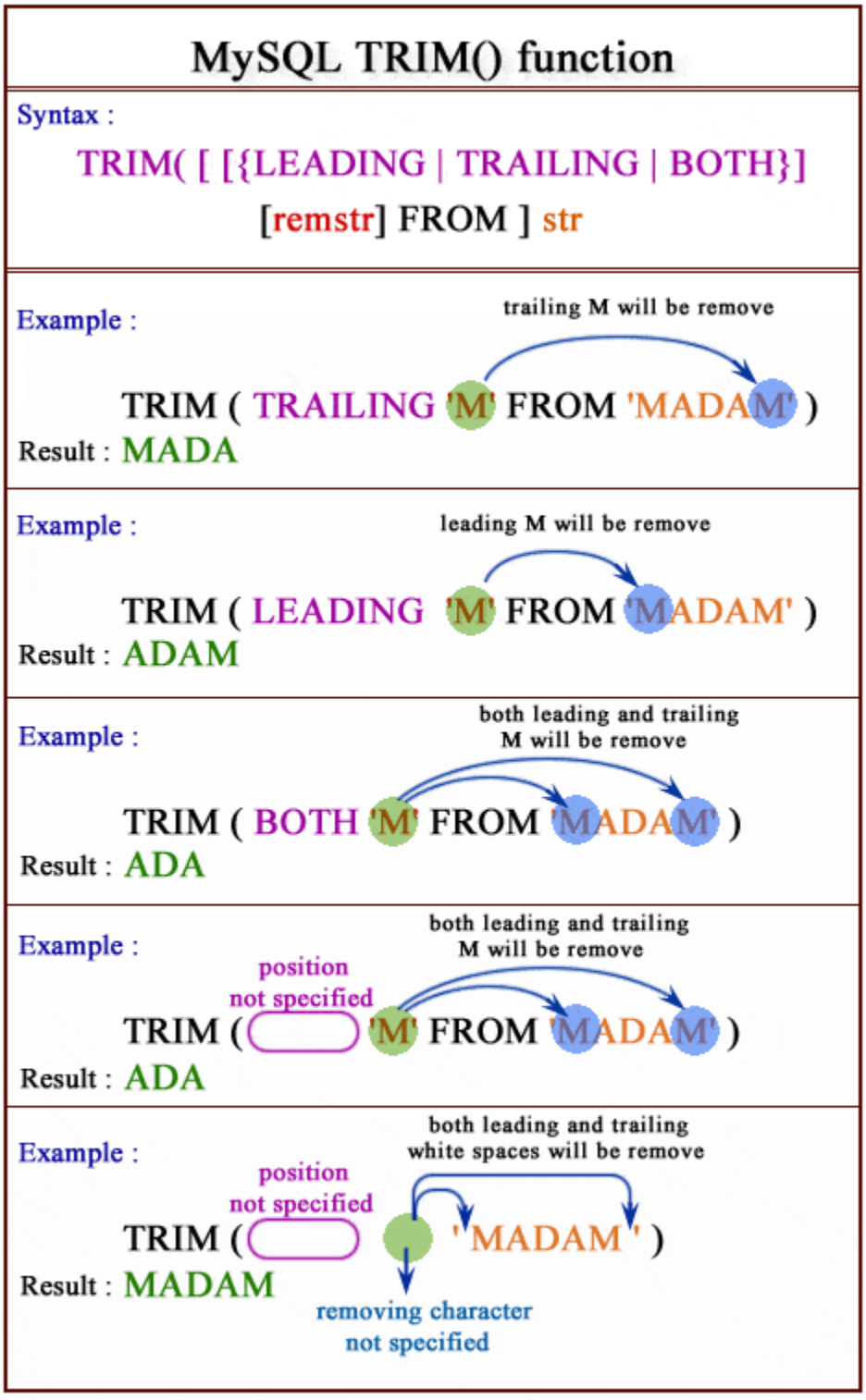
RPAD(str,len,padstr) right pads the string with the specified fill characters to the specified length.

LTRIM(string) Returns the string str with leading space characters removed

SELECT LTRIM(' barbar');
WOULD GIVE barbar

RTRIM(string) Returns the string str with trailing space characters removed

TRIM([{BOTH | LEADING | TRAILING} [remstr] FROM] str)
Returns a string after removing all prefixes or suffixes from the given string.



IFNULL(expression1, expression2); takes two expressions and if the first expression is not NULL, it returns the first expression. Otherwise, it returns the second expression.

e.g. **SELECT *, IFNULL(Comm, 0) FROM EMP;**

LENGTH(char) length in characters of specified string

Remember -- you must put single quotes round all data items which are strings.

EXAMPLES OF STRING FUNCTIONS

SELECT SUBSTR(ename,1,4) FROM emp;

UPPER(dname)

SELECT UPPER('helen campbell') FROM dual;

LOWER(ename)

SELECT LOWER('Mr Donald Briffet') FROM dual;

STR_TO_DATE('12-12-92', '%d-%m-%Y ')

SELECT STR_TO_DATE('12-06-1996',
' %d-%m-%Y ') FROM dual;

LPAD(ename, 10,' ')

SELECT LPAD(ename,10,' ') FROM emp;
(pads the name out to 10 chars with spaces before)

RPAD(ename, 10,' ')

SELECT RPAD(ename,10, ' ') FROM emp;
(as above but with spaces after)

LTRIM(ename,' ')

SELECT LTRIM(ename,' ') FROM emp;
(removes spaces from before the name)
RTRIM(ename,' ')

SELECT RTRIM(ename,' ') FROM emp;

IFNULL(comm, 0)

SELECT IFNULL (comm,0) FROM emp;

(if an employee has no commission then 0 is displayed)

LENGTH(ename)

```
SELECT LENGTH ('Anderson') FROM dual;
```

NOTE You can also rename columns within the SQL statement

```
SELECT ename Employee FROM emp;
```

will output the present ename values with the heading Employee. Note that if the new name is a single word then double quotes are not needed.

ARITHMETIC FUNCTIONS

ABS(numeric) absolute value of the number

SELECT ABS(-15) "Absolute" FROM DUAL;

MOD(num1, num2) returns the remainder when num1 is divided by num2

SELECT MOD (7,5) "modulo" FROM DUAL;

ROUND(numeric[,d]) rounds the number to d decimal places, the rounding can occur to either side of the decimal point.

SELECT ROUND (15.193,1) "round" FROM DUAL;

TRUNCATE(numeric[,d]]) truncates to d decimal places,

SELECT TRUNCATE(15.79,1) "truncate" FROM DUAL;

CEIL(numeric) rounds the number up to the nearest integer

SELECT CEIL(10.6) FROM dual;

FLOOR(numeric) truncates the number to the nearest integer

SELECT FLOOR(10.6) FROM dual;

SQRT(numeric) returns the square root of the number (returns NULL if the number is negative)

SELECT SQRT(25) FROM dual;

TO_CHAR(numeric[,format]) converts a number to a character string in the specified format

SELECT TO_CHAR(sysdate(),'DY') FROM dual;

SELECT TO_CHAR(sysdate(),'MONTH') FROM dual;

DATE_FORMAT(date,format) **Cut and Paste date_format strings for MySQL**

SELECT DATE_FORMAT(NOW(),'%Y-%m-%d %H:%i:%s')

SELECT DATE_FORMAT(NOW(),'%m')

Some more examples

SIGN(sal - comm)

```
SELECT SIGN(sal – 4 * comm) FROM emp;
```

ABS(sal - comm)

```
SELECT ABS(sal – 4*comm) FROM emp;
```

ROUND(sal,2)

```
SELECT ROUND(1234.5678,2) FROM dual;
```

TRUNCATE(comm, 3)

```
SELECT TRUNCATE(comm, 3) FROM emp;
```

GREATEST(sal, comm)

```
SELECT GREATEST(sal, comm) FROM emp;
```

DATE_FORMAT(date,format) **Cut and Paste date_format strings for MySQL**

```
SELECT DATE_FORMAT(NOW(),'%Y-%m-%d %H:%i:%s')
```

IT IS VERY IMPORTANT TO NOTE THAT IF ANY VARIABLE CONTAINS A NULL VALUE THEN ANY SQL STATEMENT INVOLVING ARITHMETIC WILL IGNORE IT
E.G.
SELECT ABS(SAL-COMM) FROM EMP;
will only produce results for employees who have a non-null commission (or salary)

AGGREGATE OR GROUPING FUNCTIONS

AVG **AVG ([DISTINCT] Column)**

This function returns the average of the values in the argument.
The data type of the argument must be numeric, date/time or character. The data type of the result is the same as the input argument.
DISTINCT eliminates duplicates.

e.g. Find the total salary budget for each department, the average salary, the number of people in each department.

```
SELECT emp.deptno, dname, SUM(sal),
AVG(sal), COUNT(empno)
    FROM emp, dept
    WHERE emp.deptno = dept.deptno
    GROUP BY emp.deptno, dname;

SELECT AVG(sal) “average” FROM emp;
```

COUNT **COUNT (*)**
(DISTINCT expression)

This function returns a count of items.
` **COUNT(*)** always returns the number of rows in the table, rows that contain null values are included.
COUNT(column-name) returns the number of column values.
COUNT(DISTINCT column-name) filters out duplicate column values.

e.g. How many employees are in each department of the EMP table?

```
SELECT COUNT(*) FROM emp
    GROUP BY deptno;

SELECT COUNT(DISTINCT job) "Jobs" FROM emp;
```

NOTE - the “” round Jobs is superfluous here, but must be used if the column heading is more than one word.

MAX **SELECT MAX(sal) FROM emp;**

MIN **SELECT MIN(sal) FROM emp;**

These functions returns the maximum or minimum value in the argument, which is a set of column values.

e.g. Find the highest and lowest salary in department 10.

```
SELECT MAX(sal), MIN(sal)
    FROM emp
    WHERE deptno = 10;
```

SUM **SUM(ALL expression)**

This function returns the sum of the values in the argument.

```
SELECT SUM(sal + comm) FROM emp
    WHERE job = 'SALESMAN';
```

NOTE - because Comm can contain a NULL value, be warned that if it does, SQL cannot evaluate it as an arithmetic expression and will ignore that record. Thus the above will obtain a sum for all Salesmen because they all get commission.
SELECT SUM(sal + Comm) FROM emp;
will try to do the same sum for all staff but since only salesmen get commission, all other employees will have null commission, their records will be ignored, and the end result will be the same.

NB these functions work down the columns they DO NOT act across the rows.

IF YOU INCLUDE GROUP FUNCTIONS IN A SELECT COMMAND YOU MAY NOT SELECT INDIVIDUAL RESULTS AS WELL.

For example, a command that begins SELECT ENAME, AVG(sal) is invalid.

ENAME has a value for each row selected while AVG(sal) has a single value for the whole query. If you use such a command SQL will display an error message.

There are two exceptions to this rule.:-
You can display individual results based on a group function in a subquery, or group results based on individual selections in a subquery.

NOTE - If an arithmetic expression encounters a Null value then that record is

ignored

e.g. SELECT SUM(SAL+COMM)
can only do the addition where COMM is not null. Similarly

SELECT AVG(SAL+COMM) will only divide the total by the number of times it has been able to do the addition (and not a count of all employees).

Summarising Several Groups of Rows.

Suppose you want to know the average salary of the employees in each department, you could enter several separate AVG(SAL) queries, one per department, but you can get the same information with a single query by using the GROUP BY clause. The GROUP BY clause divides a table into groups of rows so that the rows in each group have the same value in a specified column. (See later notes for more details about GROUP BY clause.)

To list the average salary in each department.

SELECT DEPTNO, AVG(sal) FROM emp
GROUP BY deptno;

In this example, the **GROUP BY deptno** clause divides all the employees into groups on their department number, the group function **AVG(sal**) is then applied to the rows in each group. This is a powerful function and would take many lines of code in a normal procedural language.

EXERCISES 3 FUNCTIONS

1 Find how many employees have a title of manager without listing them.

2 Compute the average annual salary plus commission for all salesmen

3 Find the highest and lowest salaries and the difference between them (single SELECT statement)

4 Find the number of characters in the longest department name

5 Count the number of people in department 30 who receive a salary and the number of people who receive a commission (single statement).

6 List the average commission of employees who receive a commission, and the average commission of all employees (assume employees who do not receive a commission attract zero commission)

7 List the average salary of employees that receive a salary, the average commission of employees that receive a commission, the average salary plus commission of only those employees that receive a commission and the average salary plus commission of all employees including those that do not receive a commission. (single statement)

8 Compute the daily and hourly salary for employees in department 30, round to the nearest penny. Assume there are 22 working days in a month and 8 working hours in a day.

9 Issue the same query as the previous one except that this time truncate (TRUNC) to the nearest penny rather than round.

DATE FUNCTIONS

Some important Date funcitons are listed below:

1. DATE_FORMAT (date,format)
It presents a date in the specified format

SELECT DATE_FORMAT(now(), '%d-%m-%Y');
See detailed instruction of DATE_FORMAT attached in the following pages

2. DATE_ADD (start_date, INTERVAL expr unit)
or start_date + INTERVAL expr unit

It adds an interval to a DATE or DATETIME. Specifically, start_date is a starting DATE or DATETIME value; INTERVAL expr unit is an interval value to be added to the starting date value.

SELECT DATE_ADD(now(), INTERVAL 1 day);

Unit	Expression
DAY	DAYS
DAY_HOUR	'DAYS HOURS'
DAY_MICROSECOND	'DAYS HOURS:MINUTES:SECONDS.MICROSECONDS'
DAY_MINUTE	'DAYS HOURS:MINUTES'
DAY_SECOND	'DAYS HOURS:MINUTES:SECONDS'
HOUR	HOURS
HOUR_MICROSECOND	'HOURS:MINUTES:SECONDS.MICROSECONDS'
HOUR_MINUTE	'HOURS:MINUTES'
HOUR_SECOND	'HOURS:MINUTES:SECONDS'
MICROSECOND	MICROSECONDS
MINUTE	MINUTES
MINUTE_MICROSECOND	'MINUTES:SECONDS.MICROSECONDS'
MINUTE_SECOND	'MINUTES:SECONDS'
MONTH	MONTHS
QUARTER	QUARTERS
SECOND	SECONDS
SECOND_MICROSECOND	'SECONDS.MICROSECONDS'
WEEK	WEEKS
YEAR	YEARS
YEAR_MONTH	'YEARS-MONTHS'

3. TIMESTAMPDIFF(unit,datetime_expr1,datetime_expr2)
It returns expr1 − expr2 expressed as a value in unit from one date to the other. expr1 and expr2 are date or date-and-time expressions.

```
select TIMESTAMPDIFF(Day, now(), HIREDATE) from emp;
select TIMESTAMPDIFF(Month, now(), HIREDATE) from emp;
select TIMESTAMPDIFF(Year, now(), HIREDATE) from emp;
```

4. LAST_DAY(date)
It takes a date or datetime value and returns the corresponding value for the last day of the month. Returns NULL if the argument is invalid.

```
SELECT LAST_DAY(NOW());
```

5. DATE(expr)
It extracts the date part of the date or datetime expression expr.

```
SELECT DATE('2003-12-31 01:02:03');
```

Similar functions include Time(), Day(), Month(), Year(), etc.

6. DATE_FORMAT(date,format)
The DATE_FORMAT function accepts two arguments:
date : is a valid date value that you want to format
format : is a format string that consists of predefined specifiers. Each specifier is preceded by a percentage character (%). See the table below for a list of predefined specifiers.
The following are some commonly used date format strings:

DATE_FORMAT string	Formatted date
%Y-%m-%d	7/4/2019
%e/%c/%Y	4/7/2019
%c/%e/%Y	7/4/2019
%d/%m/%Y	4/7/2019
%m/%d/%Y	7/4/2019
%e/%c/%Y %H:%i	4/7/2019 11:20
%c/%e/%Y %H:%i	7/4/2019 11:20
%d/%m/%Y %H:%i	4/7/2019 11:20
%m/%d/%Y %H:%i	7/4/2019 11:20

DATE_FORMAT string	Formatted date
%e/%c/%Y %T	4/7/2019 11:20
%c/%e/%Y %T	7/4/2019 11:20
%d/%m/%Y %T	4/7/2019 11:20
%m/%d/%Y %T	7/4/2019 11:20
%a %D %b %Y	Thu 4th Jul 2019
%a %D %b %Y %H:%i	Thu 4th Jul 2019 11:20
%a %D %b %Y %T	Thu 4th Jul 2019 11:20:05
%a %b %e %Y	Thu Jul 4 2019
%a %b %e %Y %H:%i	Thu Jul 4 2019 11:20
%a %b %e %Y %T	Thu Jul 4 2019 11:20:05
%W %D %M %Y	Thursday 4th July 2019
%W %D %M %Y %H:%i	Thursday 4th July 2019 11:20
%W %D %M %Y %T	Thursday 4th July 2019 11:20:05
%l:%i %p %b %e, %Y	7/4/2019 11:20
%M %e, %Y	4-Jul-19
%a, %d %b %Y %T	Thu, 04 Jul 2019 11:20:05

EXERCISES 4 DATES

1 Select the name, job, and date of hire of the employees in department 20. (Format the HIREDATE column to MM/DD/YY)

2 Then format the HIREDATE column into DoW (day of the week), Day (day of the month), MONTH (name of the month) and YYYY(year)

3 Which employees were hired in April?

4 Which employees were hired on a Tuesday?

5 Are there any employees who have worked more than 30 years for the company?

6 Show the weekday of the first day of the month in which each employee was hired. (plus their names)

7 Show details of employee hiredates and the date of their first payday. (Paydays occur on the last Friday of each month) (plus their names)

8 Refine your answer to 7 such that it works even if an employee is hired after the last Friday of the month (cf Martin)

THE GROUP BY CLAUSE

The **GROUP BY** clause is used to split rows in a table into groups or subsets. Summary calculations may then be performed on those groups of records. The grouping is performed on the basis of matching values within a column (or set of columns)

Only one line of output is presented for each group.

SELECT deptno, AVG(sal) FROM emp
GROUP BY deptno;

This will present average salaries for each deptno group along with the value of deptno within each group. Note it is important to **SELECT** the column by which you are grouping, in order to 'label' your calculated values. Whenever Oracle performs a GROUP BY it also sorts the groups on the basis of the grouping column.

RULES FOR GROUP BY

1 The **SELECT** list may contain only aggregate functions (e.g MAX(sal), COUNT(empno)) and items appearing in the group by clause.

2 The **GROUP BY** clause must be specified after any **WHERE** clause.

3 It is usual to **SELECT** columns which are specified in the **GROUP BY** clause

4 The default 'group' is the whole set of records in the table. Thus any aggregate functions will apply to the whole table if no **GROUP BY** clause is specified

SELECT MAX(sal) FROM emp;

will output one value for the maximum salary over all the employees but

SELECT ename, MAX(sal) FROM emp;

will cause an error because the table is not being grouped by ename (if you did group by ename you would see a maximum salary for each employee

SELECT job, MIN(sal) FROM emp
GROUP BY job;

will show a minimum salary for each job.

If it has a **WHERE** clause place the **GROUP BY** clause after the **WHERE** clause.

To find the average annual salary of the non-managerial staff in each department.

```
SELECT deptno,    AVG(sal) FROM emp
      WHERE job NOT IN ('MANAGER', 'PRESIDENT')
      GROUP BY deptno;
```

You may divide the rows of a table into groups based on values in more than one column, for example, to divide all employees into groups by department and job, specify both **DEPTNO** and **JOB** in the **GROUP BY** clause.

To count the employees and calculate the average annual salary for each job group in each department.

```
SELECT deptno, job, COUNT(*),    AVG(sal)
      FROM emp
      GROUP BY deptno, job;

SELECT deptno,MAX(sal) FROM emp
      WHERE job != 'PRESIDENT'
      GROUP BY deptno;
```

This shows the departmental maximums involving all employees excluding the president.

THE HAVING CLAUSE

Just as you can select individual rows to display with a **WHERE** clause you can select groups to display with a **HAVING** clause. Place the **HAVING** clause in your query after the **GROUP BY** clause.

A **HAVING** clause compares some property of the group with a constant value. If a group satisfies the condition in the **HAVING** clause it is included in the query result.

You want to list the average annual salary for all job groups with more than two employees.

```
SELECT job, COUNT(*),    AVG(sal) FROM emp
GROUP BY job
HAVING COUNT(*) > 2;
```

The **HAVING** clause compares COUNT(*) , a property of the group, to the constant value 2.

The **HAVING** clause must be specified after the **WHERE** clause and before any **ORDER BY** clause in the SQL statement. It may appear either before or after its associated **GROUP BY** clause, but it is normal to place it after the **GROUP BY**.

```
SELECT deptno, job, COUNT(empno), SUM(sal)
    FROM emp
    WHERE hiredate > '01-JAN-90'
    GROUP BY deptno, job
    HAVING COUNT(empno)>2
    ORDER BY deptno DESC, JOB;
```

Note the use of the aggregate function in the having clause. It is important to realize that aggregate functions are not allowed in **WHERE** clauses, because **WHERE** applies only to individual records - not groups of records. The **HAVING** clause is designed to work with grouped sets of records and hence can accommodate conditions based on aggregated values.

You may include both a **WHERE** clause and a **HAVING** clause in a query, if you do SQL proceeds in this order:

1. It applies the **WHERE** clause to select rows.

2. It forms the groups and calculates group functions.

3. It applies the **HAVING** clause to select groups.

To list all the departments with at least two clerks.

```
SELECT deptno FROM emp
    WHERE job = 'CLERK'
    GROUP BY deptno
    HAVING COUNT(*) >= 2;
```

To select groups based on comparisons with another group, include a subquery in the **HAVING** clause.

To list job groups whose average salary exceeds that of all the managers

```
SELECT job, AVG(sal) FROM emp
    GROUP BY job
    HAVING AVG(sal) >
            (SELECT AVG(sal) FROM emp
                WHERE job = 'MANAGER');
```

EXERCISES 5 GROUP BY & HAVING

1 List the department number and average salary of each department.

2 Divide all employees into groups by department and by job within department. Count the employees in each group and compute each group's average annual salary.

3 Issue the same query as above except list the department name rather than the department number.

4 List the average annual salary for all job groups having more than 2 employees in the group.

5 Find all departments with an average commission greater than 25% of average salary.

6 Find each department's average annual salary for all its employees except the managers and the president.

SUBQUERIES AND NESTED SUBQUERIES

A **SELECT** command may be incorporated into another **SQL** command such as **SELECT** or **UPDATE.** Such a **SELECT** command is called a subquery. The rows selected by the subquery are not displayed; instead they are fed back into the surrounding **SQL** command in one of the following ways:

If the subquery is used on the right side of a logical expression or a set expression, it must return a single value or a single column of values. The value(s) are compared to the value(s) on the left side of the expression in the manner specified by the operator connecting the two sides.

If the subquery is used to specify the values in a **CREATE, INSERT**, or **UPDATE** command, it must return one value for each column to be updated. The value(s) are used to update the specified row(s).

The **ORDER BY** and **FOR UPDATE** clauses may not be used in a subquery.

The **WHERE** clause of one query may contain another query (called a nested subquery).

```
SELECT ename FROM emp
    WHERE job = (SELECT job FROM    emp
                 WHERE ename = 'JONES')
                 AND ename != 'JONES';
```

The subquery must be enclosed in brackets and the values which are compared across the outer and subquery must be of the same datatype.

Subqueries are often used to perform stepwise processing.
Finding the person with the highest salary may be done in two steps.

1. Find the maximum salary
2. Find the person whose salary is equal to the maximum salary

```
SELECT ename, sal FROM emp
    WHERE sal = (SELECT MAX(sal) FROM emp);
```

The nested query is performed in step 1 and its result is used in step 2 as the outer query is processed.

Subqueries can be nested to any number of levels, but in practical terms 3 is usually the maximum used.

```
SELECT.......
      FROM....
            WHERE.....(SELECT.....
                        FROM......
                              WHERE.....(SELECT....
                                          FROM...
                                                WHERE....
```

Generally the subquery is executed first, and SQL compares select-fields in the 'outer' query with the results produced by the subquery.

It is possible to have 16 sub-queries at each level of nesting.

```
SELECT    select-list
      FROM .........
      WHERE    (select-field1, select-field2,.........)
            comparison operator
                  (SELECT    select-list2
                        FROM........
                        WHERE (..........))
                              comparison operator
                                    (SELECT select-list
                                          FROM ......
                                          WHERE (.........))
```

The comparisons can be any of the usual comparisons :--

= != > >= < <= LIKE (see previous notes).

If the subquery returns more than one value, one of the following words should follow the comparison operator:

ALL -- the comparison must be true for all returned values.

ANY -- The comparison need only be true for one returned value.

IN may be used in place of = ANY.

NOT IN may be used in place of != ALL.

More than one select-field may be used, but, in this case, only one test for equality may be used. Parentheses must be used to enclose the SELECT list of a subquery when it contains more than

one column.

Finding the department which has the highest total salary bill could be done in two steps.

1 Find the highest total salary paid by a department.

2 Find the department which has a salary bill which matches the value given in part 1.

```
SELECT deptno, SUM(sal) FROM emp
    GROUP BY deptno
    HAVING SUM(sal) = (select MAX(sum_sal)
                       from (SELECT deptno, SUM(sal) as sum_sal
                             FROM emp GROUP BY deptno)
                             as sum_sal_t);
```

or

```
SELECT deptno, SUM(sal) FROM emp
    GROUP BY deptno
    HAVING SUM(sal) >= All(SELECT SUM(sal)
                           FROM emp GROUP BY deptno);
```

Find the employees that earn more than at least one employee in department 30

```
SELECT sal, job, ename, deptno    FROM emp
    WHERE sal > ANY
        ( SELECT sal FROM emp
                WHERE deptno = 30)
            ORDER BY sal DESC;
```

The '=' operator should not be used if the subquery may return more than one value.
An error message will be produced.

' Subquery returns more than 1 row'.

To avoid this problem use the keyword **'IN'.**

```
SELECT ename, sal, deptno, job FROM emp
    WHERE empno IN (SELECT mgr FROM    emp);
```

Exists Operator.

In order to ensure that the subquery returns at least one row, the **EXISTS** operator can be used. The conditional expression **EXISTS** (i.e. the subquery) is TRUE if the subquery returns at least one row, and false otherwise.

Display data about employees who have at least one other employee reporting to them

```
SELECT job, ename, empno, deptno FROM    emp X
    WHERE    EXISTS
        (SELECT * FROM    emp
            WHERE X.empno = mgr)
                ORDER BY empno;
```

This is not the same as asking for all the managers, since some of the rows returned are not managers, but do have employees working for them

Multiple Conditions.

In the following example we compare both the department number and the salary. Where multiple columns are being compared and they must be enclosed in parentheses. The columns should be specified in the same order as their counterparts in the subquery.

```
SELECT ename, sal, deptno, job FROM    emp
    WHERE (deptno, sal) IN ( SELECT deptno, MIN(sal)
                                FROM emp
                                    GROUP BY deptno);
```

Synchronising a repeating subquery with a main query

Depending on the structure of a subquery, it can operate in different ways. In the previous examples, the subquery was executed once and the resulting value was substituted into the WHERE clause of the main query.

In some cases, the result of the subquery should be dependent on values in the outer query. This is termed a 'synchronised' or 'correlated' subquery.

```
SELECT    select-list    FROM    table1 label1 [.....]
    WHERE (select-field[.....] )
          comparison operator
    (SELECT select-list2    FROM    .....
        WHERE select-field comparison operator label1.select-field)
```

The important point is the use of the 'table label' in the outer query. This allows the execution of the subquery for each 'candidate row' (row that may be selected) in the outer query, and produce a subquery result depending on the data in the outer query.

Find the department number, name and salary of the employees who earn more than the

average salary in their department.

```
SELECT deptno, ename, sal FROM emp X
    WHERE sal >
        (SELECT AVG(sal) FROM emp
            WHERE X.deptno = deptno)
            ORDER BY deptno;
```

EXERCISES 6 SUB QUERIES.

1 List the name and job of employees who have the same job as Jones.

2 Find all the employees in Department 10 that have a job that is the same as anyone in department 30.

3 List the name, job, and department of employees who have the same job as Jones or a salary greater than or equal to Ford.

4 Find all employees in department 10 that have a job that is the same as anyone in the Sales department

5 Find the employees located in Liverpool who have the same job as Allen. Return the results in alphabetical order by employee name.

6 Find all the employees that earn more than the average salary of employees in their department.

7 Find all the employees that earn more than JONES, using temporary labels to abbreviate table names.

THE DATA MANIPULATION LANGUAGE

This is the section of SQL which handles data manipulation i.e. inserting, updating and deleting rows in tables It consists of three basic statements

INSERT **allows insertion of records into a table**

UPDATE **updates existing rows in a table**

DELETE **removes unwanted rows from a table**

These statements are incorporated into what is known as a transaction.

INSERTing rows into a table:-

INSERT INTO <tablename> (fieldname1, fieldname2, ...)
VALUES (value1, value2,.....);

This format allows insertion of ONE complete row into the table. The values in the list **must** be in the same order as the columns in the table and there must be a value for each column
i.e.

INSERT INTO emp (empno, ename, job, mgr, hiredate, sal, comm, deptno)
VALUES(7500, 'CAMPBELL', 'ANALYST', 7566, '30-OCT-1992', 24500, 0, 40);

If you do not have values for all the columns, a list of columns may be specified and values provided in the same order as the specified columns

INSERT INTO emp(empno, ename, hiredate, deptno)
VALUES(7888,'PITT','30-MAR-92', 30);

All unspecified columns will be set to NULL.

NB only one row at a time can be inserted using the above forms of the insert statement.

UPDATING TABLES

The general form is to update one or more rows of a table where a condition (possibly a subquery) is true.

```
UPDATE emp SET comm = 0;
```

will give all employees zero commission.

To give a 15% raise to all Analysts and Clerks in department 20 could use;

```
UPDATE emp
    SET sal = sal* 1.15
    WHERE (job = 'ANALYST' OR job = 'CLERK')
    AND deptno = 20;
```

The following **SELECT** shows how the two forms of **SET**, may be mixed in a single command, **SET deptno**... sets the updated rows' **deptno** to the value of **deptno** in the row of the table **dept** where the value of **loc** is Dallas. (**sal, comm**) .... sets **sal** and **comm** to values returned by group expressions in a subquery. **WHERE.**.. states that the updated rows are to be those whose **deptno** has a value found in the set of rows where the value of **loc** is Dallas or Detroit.

```
UPDATE emp A
   SET deptno = (SELECT deptno FROM dept
                 WHERE loc = 'PRESTON'),
                 (sal, comm) =
                 (SELECT 1.1*AVG(sal),1.5*AVG(comm)
                 FROM emp B
                 WHERE A.deptno = B.deptno)
                 WHERE deptno IN
                       (SELECT deptno FROM dept
                 WHERE loc = 'LIVERPOOL'
                 OR loc = 'LONDON');
```

If a WHERE clause is not used to limit the number of rows updated every row in the table will be updated to your specified value.

DELETING FROM TABLES

The general form is:-

DELETE FROM <tablename> WHERE [conditional statement];

If **WHERE** condition is specified all rows for which the condition is true are deleted.

i.e. To remove from EMP all sales staff who made less than 100 commission last month enter:

```
DELETE FROM emp
WHERE job = 'SALESMAN'
AND comm < 100;
```

To delete everything in a table :

```
DELETE FROM <tablename>;
```

This command does not ask for confirmation! Always make sure you use a WHERE clause in any DELETE statement unless you really want to wipe the entire table!

When a table is wiped, no space is freed up in the Oracle database for use by other tables. The space used by a table does not dynamically shrink when data is deleted from the table.

THE DATA DEFINITION LANGUAGE

DDL statements change the structure of the database. There are three basic commands:-

CREATE used to create new objects (tables, views, etc.) in the database
ALTER used to change the structure of an existing object
DROP used to remove the object from the database, (all its data plus any reference to it in the data dictionary)

TO CREATE A NEW TABLE use the CREATE TABLE statement

CREATE TABLE <tablename>
(fieldname data type,
fieldname data type,
.
);

Fieldname may be any alphanumeric name starting with an alphabetic character. The name may also contain '$' '-_' '#' '@' (maximum 30 characters)

Valid datatypes are listed after this section.

E.G. :-

```
CREATE TABLE emp
(empno NUMERIC NOT NULL,
ename CHAR(10) ,
job CHAR(9),
mgr NUMERIC (4),
hiredate DATE ,
sal NUMERIC (10,2),
comm NUMERIC (9,0) ,
deptno NUMERIC (4) NOT NULL);
```

Note that NOT NULL is specified for the empno column. This will be used as the primary key.

When a table is created you may specify criteria for its storage (such as initial space allocation). If the storage clause is not used SQL*PLUS will use the current defaults

SQL allows far more complex create statements

```
CREATE TABLE emp
 (empno NUMERIC NOT NULL,
 ename    CHAR(10) NOT NULL,
 job    CHAR(9),
 mgr    NUMERIC REFERENCES emp(empno),
 sal    NUMERIC(10,2),
 comm    NUMERIC(9,0) DEFAULT NULL,
 deptno      NUMERIC(2) NOT NULL REFERENCES dept(deptno),
Primary key(empno),
    CHECK(sal > 500)
)
```

Here the primary key is specified as are some integrity checks and simple validation. i.e.
mgr number must exist as an empno
deptno must exist in the dept table

Note this means that you must be very careful about the order in which you create the tables.

Can create **UNIQUE INDEX** to ensure the **PRIMARY KEY** does not contain duplicate values.

CREATE [UNIQUE] INDEX <indexname> ON <tablename> (index key);

every index must follow the standard ORACLE naming rules and must have a distinct name with respect to all other objects owned by a single user.

Include the name s of the tables and columns that comprise the index within the index name.

Preface the index name with I

Separate the table and column names with punctuation.

e.g. **CREATE UNIQUE INDEX I_EMP$EMPNO ON EMP (EMPNO);**

CREATE UNIQUE INDEX I_EMP$ENAME ON EMP (ENAME);

you could not have a unique index on this field as more than one person could have the same name.
this format will also work on multiple keys.

CREATE INDEX <indexname> ON <tablename>(fieldname1, fieldname2);

NB All index names must be unique!

DROP INDEX <indexname>; allows you to remove an unwanted index you can only drop indexes that you have created

SQL DATA TYPES

CHAR (size) consists of upper and lower case letters, numbers and special characters (+,-,%,$,&, etc.) size the maximum length, in characters, of the column. May not be larger than 255.

VARCHAR(size) Variable length character string data type. Only stores the actual length of the data field, does not space fill to size specified. size must be specified.

LONG Character data of variable length up to 65,535 characters. Only one long column per table.

NUMERIC Number values consisting of digits 0 -- 9, with an Values may be 38 digits wide.

NUMERIC(w,d) Number values with decimal places specified

DECIMAL(w,d) Stores numbers with up to 22 decimal digits.
W - specifies the total number of digits
d - specifies the number of decimal places.
(i.e. decimal (4,2) will allow a max. number of 99.99 to be inserted.)

INTEGER Stores numbers with 10 or fewer digits, digits to the right of the decimal point are truncated.

SMALLINT Stores digits with 5 or fewer digits.

DATE Date values - usual form dd-mmm-yy i.e. 30-OCT-98

ALTERing the table structure

To modify the definition or structure of a table, use **ALTER TABLE** command

To add a new column

ALTER TABLE <tablename>
ADD (column_name datatype);

ALTER TABLE emp
ADD gender CHAR(1);

To change the definition of an existing column.

ALTER TABLE <tablename>
MODIFY (column_name datatype);

ALTER TABLE emp
MODIFY deptno NUMERIC(6);

You cannot 'drop' or 'delete' a column using ALTER.
There is no direct support in SQL for removing columns from a table!

You may not rename a column using ALTER.
Again there is no direct support for this in SQL

The use of the ALTER TABLE statement to change column definitions is restricted to the following:-

1 If the table does not contain any data, you may
- add extra NULL or NOT NULL columns
- change the datatypes of an existing column
- alter an existing column to be NULL or NOT NULL
- make the width of the column smaller or larger

2 If the table contains rows, but there are no values in the column in question
- make the column width smaller or larger
- change the datatype

3 If the column already has data values
- make the column width larger(not smaller)
- force the column to be NOT NULL if there are no NULLs already present in the column.

DROPping objects from the database

The DROP statement may be used to remove entire unwanted objects from the database. It frees up any space they were occupying and removes all references to them from the data dictionary. (Only the creator of the table can DROP it)

DROP TABLE <tablename>;

will remove the specified table (with its contents) from the database.
When you drop a table, SQL automatically drops indexes for the table, synonyms for the table's name and privileges granted on the table. Views that refer to the table are not dropped, but become invalid. You should drop them or redefine them, or (re)define other tables in such a way that the views become valid again.

INDEX It is possible for two indexes from different tables to have the same name. In that case, when you drop one of the indexes you must specify ON table to identify the index you want to drop.

DROP INDEX ind1;

Remember that this statement is very severe and should be used with extreme caution.

No confirmation is requested by Oracle in order to perform this operation
Oracle issues an implicit command both before and after it processes to DROP command. This means you cannot roll it back.

You cannot use the DROP command to DROP columns, deleting columns from tables can not be done directly, the following can be used

To delete the column loc from the dept table

1 Create a new table which is the image of the dept table excluding the LOC column

CREATE TABLE newdept
AS SELECT deptno, dname FROM dept;

this produces a table containing all the data for the deptno and dname which already exists in the dept table. Column headings will default to those in the dept table.

2 Now drop the old table

DROP TABLE dept

3 Rename the new table to the old table name

RENAME newdept TO dept

```
EXEC sp_rename 'emp', 'emp10'
```

VIEWS

Views can be regarded as windows through which users may see data stored in database tables. They have a number of attractive features:-

(i) they do not own any data of their own

hence they take up virtually no space in the database (only that required for their definition in the relevant data dictionary)

(ii) they are automatically activated when the user references them in an SQL statement

this means they will always reflect the current state of the database

(iii) they may be simple or arbitrarily complex

views may be based on single or multiple tables and may also reference other views

views may be tailored to suit user requirements and make the users task easier (e.g. avoid specification of complex joins)

simple views are based on a single table and only contain columns which are directly stored in the table in question

(iv) views are merely stored SQL statements

hence they can be defined using familiar SQL constructs

(v) may be treated as tables in SQL queries

almost 100% compatible with table usage

(vi) can be used to implement row level security within SQL*PLUS

the GRANT statement does not provide this functionality

(vii) may be used to implement integrity (including referential) checks

SQL uses constraints to perform this function

(viii) useful in providing a level of data independence for application programs

their use allows the structure of the database to change with minimal effect on users and application programs.

VIEW MANAGEMENT

View definitions may be seen using the following dictionary views

USER_VIEWS ALL_VIEWS DBA_VIEWS

these show the viewnames along with the full view definitions.

Views may be created using the **CREATE VIEW<viewname>** command (no storage definition is required) They may be dropped with the **DROP VIEW<viewname>** command

Views may not be altered, they are essentially stored SQL statements, so for complex views it is advisable to save their definitions in a command file so that they can be changed more easily if needed.

When a table is dropped any views built on it become inaccessible.

The use of views can present a performance overhead, mainly in increased parse times.

CREATEing Views

The **CREATE VIEW** command allows you to create a view by specifying a standard SQL query

```
CREATE VIEW <VIEWNAME> [(col1, col2,...)]
    AS SELECT <some statement to present the required data>
        [WITH CHECK OPTION];
```

The specification of column headings in the view is normally optional and they will obviously correspond on a one to one basis with items in the SELECT list of the query.

To create a simple view on the emp table:-

```
CREATE VIEW dept30
    AS SELECT ename, sal, comm FROM emp
        WHERE deptno = 30;
```

At this point Mysql will reply with 'View Created'.

The view can be used and referenced as you would a normal mysql table-

```
SELECT ename, sal FROM dept30
    WHERE comm IS NOT NULL;
```

Complex views can be used to make life easier for the user and also to prevent virtual columns (columns that do not exist in the base table)

```
CREATE VIEW total_comp (employee, job, salary, commission, annual_sal, total)
    AS SELECT ename, job, sal, NVL(comm,0), sal*12, sal*12+NVL(comm,0)
        FROM emp;
```

Views which contain virtual columns MUST have their own column headings specified. In the above example annual_sal represents sal*12 from the base table.

Views cannot contain an **ORDER BY** clause this must be specified in the normal **SELECT** statement

```
SELECT * FROM dept30
    ORDER BY sal;
```

DATA MANIPULATION of Views

It is tempting to make heavy use of views, however some serious problems are likely to be encountered when attempting to manipulate data through views.

Consider the following view definition:-

```
CREATE VIEW summary
    AS SELECT deptno AVG(sal)
        FROM    emp
            GROUP    BY deptno;
```

This view contains an aggregate function which makes all data seen through the view non updatable.

Views on more than one table suffer similar restrictions.

```
CREATE VIEW deptemp
    AS SELECT empno, ename, hiredate, sal, comm, deptno, dname, loc
        FROM    emp
            WHERE emp.deptno = dept.deptno;
```

The output from this view would look like :-

EMPNO	ENAME	HIREDATE	SAL	COMM	DEPTNO	DNAME	LOC
7777	COX	11-APR-92	2000	500	30	SALES	LIVERPOOL

If you tried to delete Cox's details from the view, Oracle would try to translate the delete command into two separate deletes on the two base tables, this could cause problems as we could find the remaining employees in Cox's department would have no corresponding department record in the department table.
All DML operations on views based on more than one table are disabled.

UPDATEing Views

The Rules

1 Views containing GROUPed sets of data:

no DML is allowed on any column in the view

2 Views based on the join of one or more tables (or views)

no DML is allowed on any column in the view

3 Views containing virtual columns

updates allowed on all but the virtual columns
delete operations are unrestricted
inserts are allowed if ALL not null columns are specified and no attempt is made to insert a value in any of the virtual column(s)

E.G. given the following view

```
CREATE VIEW virtualcols
    AS SELECT empno, ename, sal, comm, sal + comm, total
        FROM    emp;
```

The following statements are **legal**

UPDATE virtualcols SET sal =9999 WHERE empno = 7934;

INSERT INTO virtualcols VALUES(7777,'COX',8888,1111);

The following statements are **illegal**

UPDATE virtualcols SET total = 9999 WHERE empno = 7934;

INSERT INTO virtualcols VALUES(7777,'COX',8888,1111,9999);

4 Simple views which do not possess ALL of the NOT NULL columns

no INSERTion of rows is possible (only updates and deletes)

5 Views containing the WITH CHECK OPTION

Updates are restricted to those which result in data which still complies with the check (data migration is prevented)

EXERCISES 7 Data Manipulation

1 Create a new table called loans with columns named LNO NUMERIC (3), EMPNO NUMERIC (4), TYPE CHAR(1), AMNT NUMERIC (8,2)

2 Insert the following data

LNO	EMPNO	TYPE	AMNT
23	7499	M	20000.00
42	7499	C	2000.00
65	7844	M	3564.00

3 Check that you have created 3 new records in Loans

4 The Loans table must be altered to include another column OUTST NUMERIC(8,2)

5 Add 10% interest to all M type loans

6 Remove all loans less than £3000.00

7 Change the name of loans table to accounts

8 Change the name of column LNO to LOANNO

9 Create a view for use by personnel in department 30 showing employee name, number, job and hiredate

10 Use the view to show employees in department 30 having jobs which are not salesman

11 Create a view which shows summary information for each department.

SET OPERATORS

UNION returns all distinct rows returned by *either* of the queries it applies to

INTERSECT returns all rows returned by *both* of the queries it applies to

MINUS returns all rows returned by the *preceding* query, but not by the following query

other operators are **JOIN**

THE UNION OPERATOR

this is a more generalised form of **'OR'** it allows the result from two (or more) queries to be returned as a single set.

e.g. Finding details of people who earn the same salary as Scott or Ward can be achieved using an OR construct as follows :-

```
SELECT ename, job, sal FROM    emp
    WHERE sal IN
        (SELECT sal FROM emp
            WHERE ename = 'SCOTT'
                OR ename = 'WARD');
```

BUT if Scott and Ward are in different tables a union construct is necessary

```
SELECT ename, job, sal FROM emp
    WHERE sal IN
        (SELECT sal FROM emp
            WHERE ename ='SCOTT'
        UNION
        SELECT sal FROM emp2
            WHERE ename ='WARD');
```

USE OF INTERSECT

This is a more general form of **AND**

```
SELECT job FROM emp
    WHERE sal >2000
INTERSECT
SELECT job FROM shopfloordetails;
```

USE OF MINUS

this is sometimes called DIFFERENCE

```
SELECT deptno, dname, loc    FROM    dept
    WHERE deptno    IN
        (SELECT deptno FROM    dept        [first]
        MINUS
        SELECT deptno FROM emp);    [second]
```

MINUS returns the rows from the first query which are not also returned by the second query.

THE ANY OPERATOR

Find all employees who earn more than **any** employee in department 30

```
SELECT sal, job, ename, deptno FROM    emp
    WHERE sal > ANY (SELECT sal FROM emp
                WHERE deptno = 30);
```

Can be rewritten using the **'MIN'** aggregate function

```
SELECT sal, job, ename, deptno FROM    emp
    WHERE    sal> (SELECT MIN(sal) FROM emp
            WHERE deptno = 30);
```

The ANY construct is almost entirely redundant

(=ANY performs the same function as IN)

THE ALL OPERATOR

To display information about employees who earn more than all employees in department 30

```
SELECT sal, job, ename, deptno FROM    emp
    WHERE sal > ALL (SELECT sal FROM emp
                WHERE deptno = 30);
```

Can be rewritten using the '**MAX'** aggregate function

```
SELECT sal, job, ename, deptno FROM emp
    WHERE sal > (SELECT MAX(sal) FROM    emp
            WHERE deptno = 30);
```

LOGICAL OPERATORS

= equal to

!= not equal to

<> not equal to

> greater than

>= greater than or equal to

< less than

<= less than or equal to

ALL modifies the action of a condition to apply to all members of a list of values

ANY modifies the action of a condition to apply to any member of a list of values

[NOT] BETWEEN ..... AND..... greater than or equal to one value and less than or equal to another

EXISTS Evaluates true if the following subquery returns at least one row

[NOT] IN equal to any member of the set of values

IS [NOT] NULL tests for null