INTRODUCTION TO ORACLE SQL*PLUS

by Junping Sun

Graduate School of Computer and Information Sciences
Nova Southeastern University
3301 College Avenue
Fort Lauderdale, Florida 33314-7796
E-mail: jps@nova.edu

What is ORACLE?

There are several major components as referred to the ORACLE DBMS software package as follows:

- ORACLE DBMS Server
  The major DBMS engine.

- ORACLE SQL*PLUS
  The user-friendly interactive SQL (structural query language) interface to access databases.

- ORACLE PL/SQL
  Combination of procedural language (PL) and structural query language (SQL).

- ORACLE SQL*FORMS
  The front-end application development tool.

- ORACLE SQL*REPORTWRITER
  Report generator.

- ORACLE SQL*NET
  Oracle client/server database connect tool.

- ORACLE Open Client Adapter for ODBC (Open Database Connectivity)
  Heterogeneous client/server database connect tool.

In this document, we will concentrate on the ORACLE SQL*PLUS because it is not only the fundamental component in the ORACLE DBMS environment, but also a user-friendly interactive interface from which you can start to learn about ORACLE DBMS.

Starting and Stopping SQL*PLUS:

This document describes how to use SQL*PLUS from the beginning. You should be able to follow the examples in SQL*PLUS given here and to observe the results of executions.

At NSU, the computer on which the ORACLE DBMS resides is called oracle11g.

In order to use the ORACLE SQL*PLUS, you can login on `scis` (computer) first. After you login on `scis` successfully, you will see the following at your `scis` prompt:

```
usercode@scis>
```

where the `usercode` is your login `usercode` and `scis` is the name of the computer.

To access SQL*PLUS, you just type `oracle11g` followed by either the `return` or the `enter` key after the prompt `usercode@scis>` as the following:

```
usercode@scis> oracle11g
```

The ORACLE DBMS will display the messages for you to input both your ORACLE user-name and password as follows:

```
+-------------------------------------------------------------+
|           Welcome to Nova Southeastern University           |
|          Office of Information and Technology (OIT)         |
|                   Oracle Database Server                    |
|     Running Oracle Enterprise Edition 11.1.0.6.0 64bit      |
|       Please login with syntax : username@oracle11g         |
|  Please contact oracle@nsu.nova.edu for account problems.   |
+-------------------------------------------------------------+

Executing SQL*PLUS...


SQL*Plus: Release 11.1.0.6.0 - Production on Mon Dec 29 12:07:34 2008

Copyright (c) 1982, 2007, Oracle.  All rights reserved.


Enter user-name:
```

At the prompt of Enter user-name from ORACLE SQL*PLUS, please enter both your user-name and the password **in one line**.

For example:

```
Enter user-name:  username/password@oracle11g
```

Please note that the username is the same as your email `usercode` that you use to login on `scis` unix; the password is followed by `@oracle11g.` The symbol '/' is necessary to separate the username and the password in the connecting string.

If you enter both the user-name and the password correctly, then the system will display the followings:

```
Connected to:
Oracle Database 11g Enterprise Edition Release 11.1.0.6.0 - 64bit
Production
With the Partitioning, OLAP, Data Mining and Real Application
Testing options

SQL>
```

When it prompts `SQL>`, the system is ready for you to enter either a SQL statement or a SQL*PLUS command.

A SQL statement is the statement that complies with the syntax rule of SQL. A list of SQL statements such as `create`, `delete`, `insert`, `select`, and `update` will be frequently used.

The SQL*PLUS commands are the auxiliary set of commands that may be only recognized by ORACLE SQL*PLUS. Frequently used commands in ORACLE SQL*PLUS interface are `clear`, `describe`, `edit`, `get`, `help`, `list`, `save`, `spool`, and `run`.
To quit and end the SQL*PLUS session, you could use either the `exit` or the `quit` SQL*PLUS command after the prompt of `SQL>`.

In the next, we illustrate both how to use SQL statements to access a database and how to use the auxiliary set of interface SQL*PLUS commands.

To Retrieve Data from a Database Table:

To print out the data from the database table, the employee table owned by the user `jps`, simply type the following statement at the prompt `SQL>`.

```
select     *     from       jps.employee;
```

After you type the statement followed by either the return or the enter key, the system will automatically execute the SQL statement and display the result from the execution on your screen.

In the above statement, there are two clauses as follows:

The select clause: `select *`

The from clause: `from jps.employee;`

The `select` clause will include all the columns from the `jps.employee` table in the result table, and the `from` clause indicates the `jps.employee` table where the data records or the tuples will be retrieved. Here, the symbol `'*'` in the select clause implies all the column names in the `jps.employee` table, i.e., the values of all the columns in the table will be retrieved and displayed as the result from the execution of the select SQL statement.

In order to improve the readability, you could enter the select statement in several lines such that each line contains one clause as follows:

```
SQL>  select     *
  2   from       jps.employee;
```

First, after the prompt `SQL>`, you can type the select clause, `select *`, with entering either the `return` or `enter` key at the end of the current line. The system will prompt the line number, `2`, with the cursor at the beginning of line `2`.

Second, you can enter the from clause, from `jps.employee`, in line `2`. Also please note that there is a semicolon at the end of the `from` clause.

- The semicolon at the end of the second clause means the end of the whole select statement.
- After a SQL statement is entered, the SQL statement will be kept in a specific place called SQL*PLUS buffer or memory buffer, but the SQL*PLUS command will not. There is also a corresponding file named `afiedt.buf` that is used to save the current content in the buffer.
- From the point view of SQL syntax rules, you need to put a semicolon at the end of each SQL statement to indicate the end of the statement, especially, there are multiple SQL statements for the execution in the buffer. In some situation, the semicolon of the last SQL statement can be omitted. This also applies to the situation where only one SQL statement is executed at one time.

If you key in the statement correctly, the system will display the data in the `employee` table owned by the user `jps`.

```
NAME                 SSN          BDATE    SEX   SALARY   SUPERSSN   DNO
------------------   ---------  ----------  ----  -------  ---------  ---
John B Smith         123456789  09-JAN-55   M     30000    333445555  5
Franklin T Wong      333445555  08-DEC-45   M     40000    888665555  5
Alicia J Zelaya      999887777  19-JUL-85   F     25000    987654321  4
Jennifer S Wallace   987654321  20-JUN-31   F     43000    888665555  4
Ramesh K Narayan     666884444  15-SEP-52   M     38000    333445555  5
Joyce A English      453453453  31-JUL-62   F     25000    333445555  5
Ahmad V Jabbar       987987987  29-MAR-59   M     25000    987654321  4
James E Borg         888665555  10-NOV-27   M     55000               1
```

8 rows selected.

What is displayed on your screen is the data in the `jps.employee` table owned by user `jps`.

If you do not succeed in the execution, then there might be some typos. At this time, it is better to use the command `clear buffer` to remove the old statement in the SQL*PLUS buffer before you key in a new statement. Later on, we will discuss how to use editor to correct the errors in the SQL*PLUS buffer.

To clear the SQL*PLUS buffer, type the command clear after the prompt `SQL>`.

```
SQL> clear buffer
```

It is also a good practice to clear the SQL*PLUS buffer before you start entering any new statement.

Check the Relations or Tables in a Database:

There are two types of relations or tables in a database. One is the set of tables created by yourself, and another is the set of tables created by some other users as we just see. In order to access the tables created by others, you need to be granted the access privilege by either the DBA (database administrator) or the original user who creates these tables. Next, we will learn how to find out not only a list of tables created by yourself, but also a list of tables that are created by other users and granted for you to access.

To display a list of tables created by yourself, type the following line after the prompt `SQL>`:

```
SQL> select      *
  2  from        tab
  3  where       tabtype = 'TABLE';

no rows selected
```

If you key in everything correctly, the system will display the message such that "`no rows selected`," i.e., there is nothing retrieved from the execution of the select statement. That is true since you have not created any table yet. Please note that the constant string 'TABLE' is different from the constant 'table' in the where clause because the string constant enclosed within both the left single quote and the right single quote are case-sensitive. Since the internal representations of upper case letters in computer systems are different from those of lower case letters.

Another way to view the list of tables created by yourself is to use the following command:

```
SQL> select *
  2  from   system.tab;
```

Although you have not created any table, you may have been granted the access to other tables created by some other users. To find it out if you are granted the access to the tables created by some user, for example, `jps`, type the select statement as follows:

```
SQL>  select     object_name, object_type
  2  from        all_objects
  3   where      owner = 'JPS' and object_type = 'TABLE';


OBJECT_NAME           OBJECT_TYPE
--------------------  --------------------
DEPARTMENT            TABLE
DEPENDENT             TABLE
DEPTLOCATION          TABLE
EMPLOYEE              TABLE
PROJECT               TABLE
R                     TABLE
S                     TABLE
WORKSON               TABLE


8 rows selected.
```

Please note that the constant string JPS is capitalized in the where clause,

```
where owner = 'JPS'
```

You need to type the following command after the prompt `SQL>` in order to find out the attribute types in the table `jps.employee` owned by `jps`,

```
SQL> describe jps.employee;

Name                  Null?                Type
--------------------  ---------------      --------------------
NAME                  NOT NULL             VARCHAR2(19)
SSN                   NOT NULL             CHAR(9)
BDATE                                      DATE
SEX                                        CHAR(3)
SALARY                                     NUMBER(8,2)
SUPERSSN                                   CHAR(9)
DNO                                        VARCHAR2(8)

6 rows selected.
```

The describe command displays not only the names of all the columns but also the corresponding data types of all the columns in a table. The `'NOT NULL'` in the second column from the left indicates that the `NULL` values are not allowed in both `NAME` and `SSN` column entries from the `jps.employee` table.

Please try the following commands one by one after the prompt `SQL>`,

```
describe  jps.department;

describe  jps.dependent;

describe  jps.deptlocation;
```

```
describe  jps.project;

describe  jps.workson;

describe  jps.r;

describe  jps.s;
```

After you finish the above commands, type in the following SQL statements one by one to print out the data in each table.

```
select     *
from       jps.employee;

select     *
from       jps.department;

select     *
from       jps.dependent;

select     *
from       jps.deptlocation;

select     *
from       jps.project;

select     *
from       jps.workson;

select     *
from       jps.r;

select     *
from       jps.s;
```

Up to now we have learned how to print out both the description of a table structure (relation schema) and the data in a table (relation). We will learn more about the SQL*PLUS commands before we learn more about the SQL (Structural Query Language).

The SQL*PLUS Commands:

In general, the set of SQL*PLUS commands is independent of the ANSI-SQL standard, i.e., it is the set of proprietary commands in ORACLE SQL*PLUS interface. There is no guarantee that the set of proprietary commands in ORACLE SQL*PLUS will be supported by other relational DBMS.

All the SQL*PLUS commands **must** be issued after the `SQL>` prompt instead of the line number prompt. Some of the SQL*PLUS commands require the argument(s) and some of them do not.

If the current prompt is the line number instead of the prompt `SQL>`, just hit either the return or the enter key to switch from the line number prompt to the `SQL>` prompt.

The following is the list of frequently used SQL*PLUS commands:

`clear`, `describe`, `edit`, `get`, `help`, `list`, `run`, `save`, `spool`, and `start`.

`clear buffer`

The **`clear`** command removes all the contents in the SQL*PLUS buffer and reset the buffer.

`describe tablename`

The **`describe`** command displays all the attribute names and the attribute types in the table **`tablename`**.

Example:

```
describe   jps.project;
```

It will display all the attribute names and attribute types in the table `jps.project`.

`edit`

The **`edit`** command will invoke the default system editor. If you find the errors in the SQL statement, you could use the edit command to correct these errors in the SQL statement. Before exiting the edit session, you need to save the changes that you make during the editing session.

The current default editor in our system is `PICO`. You could redefine the default editor by using the **`define_editor`** command.

To redefine the default system editor as the `vi` editor, just type the followings after the prompt `SQL>`,

```
define_editor = "vi"
```

`get filename`

The **`get`** command will load the file **`filename`** under your current working directory to the SQL*PLUS buffer. After the file is loaded successfully by `get` command, the content in the file will be listed and displayed on the screen.

`help`

The **`help`** command will display the on-line help menu in the SQL*PLUS. The on-line help menu is self-contained and very easy to read.

`list`

The **`list`** command will display on the screen the current contents in the SQL*PLUS buffer.

`run`

The **`run`** command will `list` the current contents in the SQL*PLUS buffer and execute it.

There are three ways to execute the SQL statement in the SQL*PLUS buffer:

1. Put the semicolon at the end of the SQL statement.

The current SQL statement in the SQL*PLUS buffer will be automatically executed after you put the semicolon at the end of the SQL statement with following either `return` or `enter` key. This will apply to the situation when you just type in a new SQL statement and want to execute it immediately.

Example:

```
SQL> select   *
  2  from     jps.employee;
```

2. If you did not put the semicolon at the end of the SQL statement and the current cursor is at the beginning of the next line below the SQL statement, put the forward slash `/`.
   This will apply to the situation where there is no semicolon at the end of the SQL statement and the cursor has been moved to the next line below the SQL statement.

   Example:

```
SQL> select   *
  2  from     jps.employee
  3  /
```

3. You can use either run or forward slash `/` command to execute the SQL statement in the SQL*PLUS buffer after you finish editing the SQL statement with the default system editor.

   - The `run` command must be issued after the prompt `SQL>`, i.e., you must switch from the line number prompt to the `SQL>` prompt by hitting either the `return` or `enter` key before issuing the `run` command to execute SQL statement in the buffer. Please see the following example:

     ```
     SQL> select      *
       2  from        jps.employee
       3  <enter key> or <return key>
     SQL> run
     ```
   - The forward slash `/` command is different from the run command in that the forward slash `/` will not list the current contents in the SQL*PLUS buffer before the execution.

   - The forward slash `/` command can be used after either the line number prompt or the `SQL>` prompt.

`save filename`

The **`save`** command will save the current contents of the SQL*PLUS buffer into the file **`filename`** in your current working directory on scis unix.

`save filename  replace`

The **`save`** command with the keyword **`replace`** at the end of the command line will overwrite the current content in the file **`filename`** with the current content in the buffer.

spool

The **spool** command can be used to save a SQL*PLUS session into a file.

To start the spool session, simple type **spool filename.lst** after the prompt SQL>.

To end the spool session, simple type **spool off** after the prompt SQL>.

Any session information such as both SQL statement entered from the keyboard and its execution result between the command line **spool filename.lst** and the command line **spool off** will be saved in the file **filename.lst**. If the file extension .lst is omitted, the system will automatically append it for you.

Example:

```
SQL>  spool log.lst

SQL>  select     *
SQL>  from       jps.employee;

SQL>  spool off
```

In this example, the SQL statement and its execution result will be spooled or saved into the file with the name log.lst.

Please note that the **spool off** command will terminate the spooling session, and each **spool off** command matches the *latest* **spool filename.lst** command (beginning of the spooling session).

If you start a new spooling session with the existing file name, the new spooling contents will overwrite the contents in the existing file.

If you use the command spool with the option append, then the new spooling session contents will be appended to the end of the existing file.

Example:

```
SQL>  spool   log.lst  append

SQL>  select     *
SQL>  from       jps.employee;

SQL>  spool off
```

The new contents with the select statement and its execution result will be appended to the end of the file log.lst.

start filename

The **start** command can be used to load one or more SQL statements stored in a file under your current working directory and to execute the SQL statement(s) immediately after loading. You should put a semicolon at the end of each SQL statement except the last SQL statement in the sequence if there are several SQL statements in the loaded file.

Operations in the Relational Database Model:

In the traditional relational database model, there are eight operations such as select, project, join (theta-join), union, intersect, minus, product, and division. There is no commercial relational DBMS product that directly supports the division operation. The indirect implementation method in the SQL can support the division operation defined in the relational database model. The ORACLE DBMS supports outer join, and transitive closure operations besides select, project, join (theta-join), union, intersect, minus, and product operations. Both outer join and transitive closure operations have found many applications in data processing. Oracle10g SQL*PLUS supports the syntax of outer joins, cross join, natural join, inner join, linear transitive closure operation, etc.

SELECT Operation:

The select operation in terms of the relational database model is a unary operation to retrieve records or tuples in a table based on some select condition. The select operation is used to retrieve one or more records (tuples) from a database table and can be specified in the where clause of a SQL select statement.

The following SQL statement is to **retrieve the employees who work for department number 5**. After the prompt SQL>, type in the statement as follows:

```
SQL> select      *
     from        jps.employee
     where       dno = 5;

      NAME            SSN        BDATE     SEX   SALARY   SUPERSSN    DNO
----------------  ----------  ----------  ---  -------  ----------   ---
John B Smith       123456789   09-JAN-55   M     30000   333445555    5
Franklin T Wong    333445555   08-DEC-45   M     40000   888665555    5
Ramesh K Narayan   666884444   15-SEP-52   M     38000   333445555    5
Joyce A English    453453453   31-JUL-62   F     25000   333445555    5

4 rows selected
```

The result table from querying employee table with the select condition `dno = 5` lists all the employees' information from department number 5. The symbol '=' is the comparison operator, and the comparison expression `dno = 5` specifies the select condition for the query.

Comparison Operators:

The following list contains the comparison operators available in ORACLE SQL*PLUS:

```
=          (equal)
!=         (not equal)
<>         (not equal)
>          (greater than)
>=         (greater than or equal)
```

```
<           (less than)
<=          (less than or equal)
```

The above list of comparison operators can be used for any comparison expression in a where clause. The comparison expression can be used to specify either the select condition for the select operation or the join condition for the join operation in terms of relational database operations. If the comparison expression contains one item that is a column name in a table and one constant of the same type as the column name, then the comparison expression specifies a select condition for a query. If the comparison expression contains the items such as column names on the both sides of the comparison operator, then the comparison expression defines a join condition. The join operation will be discussed later.

To retrieve the employees whose salary is greater than $30,000, we could use the following SQL statement:

```
SQL> select *
  2  from   jps.employee
  3  where  salary > 30000;


       NAME           SSN         BDATE    SEX   SALARY   SUPERSSN  DNO
------------------  ---------  ----------  ----  -------  ---------  ---
Franklin T Wong     333445555  08-DEC-45    M     40000   888665555   5
Jennifer S Wallace  987654321  20-JUN-31    F     43000   888665555   4
Ramesh K Narayan    666884444  15-SEP-52    M     38000   333445555   5
James E Borg        888665555  10-NOV-27    M     55000               1

4 rows selected.
```

In the where clause, the comparison expression, `salary > 30000`, contains a column name `salary` and a constant `30000`.

Logical Operators:

There are three logical operators that can be used to connect multiple comparison expressions in a `where` clause.

```
and
or
not
```

To retrieve the female employees whose salary is more than $30000, the following select statement can be used:

```
SQL> select *
  2  from   jps.employee
  3  where  sex = 'F' and salary > 30000;


       NAME           SSN         BDATE    SEX   SALARY   SUPERSSN  DNO
------------------  ---------  ----------  ----  -------  ---------  ---
Jennifer S Wallace  987654321  20-JUN-31    F     43000   888665555   4

1 rows selected.
```

in the `where` clause, the comparison expressions such as `sex = 'F'` and `salary > 30000` are connected by the logical operator `and`.

To retrieve the employees who work on at least one of projects with project number 1, 2, or 3.

The query statement can be formulated with comparison expressions `pno = 1`, `pno = 2`, and `pno = 3` connected by operators `or` in the `where` clause.

```
SQL> select *
  2  from   jps.workson w
  3  where  w.pno = 1 or w.pno = 2 or w.pno = 3;


ESSN              PNO         HOURS
---------         -----       ------
123456789         1           32.5
123456789         2           7.5
666884444         3           40
453453453         1           20
453453453         2           20
333445555         2           10
333445555         3           10


7 rows selected.
```

- In the above select statement, the table `jps.workson` is aliased as `w` in the `from` clause.
- In the `where` clause, the column name (attribute name) `pno` is prefixed with the aliased table name `w`. The purpose of doing that is to avoid ambiguity when the same column name appears in more than one table in a database. The prefix is used to indicate from which table the attribute name is referred.

`IN` Operator:

The select condition in the above query statement can be reformulated by using operator IN.

```
SQL> select *
  2  from   jps.workson w
  3  where  w.pno in (1,2,3);
```

The select condition `pno in (1, 2, 3)` is equivalent to the following:

```
w.pno = 1 or w.pno = 2 or w.pno = 3
```

Try the above select statement to see whether the execution result is the same or not.

By using the IN operator, the join operation can be eliminated in certain situations.

For example, to retrieve the names and social security numbers of employees who works on at least one of projects with project number `1`, `2`, or `3`. There are several ways to formulate the query by using select statement.
The first select statement used in the following includes a join operation, the result from the execution of the query is duplicated. The query statement with the join operation will **retrieve the employee's name and social security number for the employee who works on project 1, 2, or 3**.

```
SQL> select e.name, e.ssn
  2  from   jps.employee e, jps.workson w
  3  where  w.pno in (1,2,3) and e.ssn = w.essn;


NAME                  SSN
-------------------  ---------
John B Smith         123456789
John B Smith         123456789
Ramesh K Narayan     666884444
Joyce A English      453453453
Joyce A English      453453453
Franklin T Wong      333445555
Franklin T Wong      333445555

7 rows selected.
```

- the expression `e.ssn = w.essn` specifies the join condition.
- the expression `w.pno in (1,2,3)` specifies the select condition.

The second select statement as the following uses the operator IN to answer the query without any duplicates.

```
SQL> select e.name, e.ssn
  2  from   jps.employee e
  3  where  e.ssn  in (select    w.essn
  4                             from   jps.workson w
  5                             where  w.pno in (1,2,3));

NAME                 SSN
-------------------  ---------
John B Smith         123456789
Franklin T Wong      333445555
Joyce A English      453453453
Ramesh K Narayan     666884444

4 rows selected.
```

In the above query statement, there are two subqueries, one is called the inner subquery enclosed within the left and right parentheses, another one is called the outer subquery.

The inner subquery:

```
select w.essn
from   jps.workson w
where  w.pno in (1,2,3)
```

The outer subquery:

```
select e.name, e.ssn
from   jps.employee e
where  e.ssn in ...   (result from the inner subquery)
```

The execution result from the inner subquery will be used as the part of select condition in the where clause of the outer query.

One of the advantages of this select statement with subqueries is to eliminate the join operation and to reduce the cost of execution. This select statement requires only two unary operations such as the two select operations instead of join operation and generates small intermediate and final result table without any duplicates.

`NOT IN` as the Negation of `IN`:

The select condition `pno in (1, 2, 3)` is equivalent to the following:

```
pno = 1 or pno = 2 or pno =3
```

whereas the select condition `pno not in (1, 2, 3)` is equivalent to the following:

```
pno <> 1 and pno <> 2 and pno <> 3
```

To retrieve both the social security numbers and names of employees who work on these projects other than the projects with project number `1`, `2`, or `3`, you could use the expression `not in (1,2,3)` in the `where` clause.

```
SQL> select e.name, e.ssn
  2  from   jps.employee e
  3  where  e.ssn  in (select w.essn
  4                    from   jps.workson w
  5                    where  w.pno not in (1,2,3));

NAME                 SSN
-------------------  ---------
Franklin T Wong      333445555
James E Borg         888665555
Jennifer S Wallace   987654321
Ahmad V Jabbar       987987987
Alicia J Zelaya      999887777

5 rows selected.
```

To retrieve both the social security numbers and names of employees who do not work on any one of the projects with project number `1`, `2`, and `3`. The list of employees who work on one of the project `1`, `2`, or `3` should be excluded from the whole list of employees.

```
SQL> select e.name, e.ssn
  2  from   jps.employee e
  3  where  e.ssn  not in  (select w.essn
```

```
  4                          from   jps.workson w
  5                           where  w.pno  in (1,2,3));


NAME                 SSN
-------------------  ---------
Alicia J Zelaya      999887777
Jennifer S Wallace   987654321
Ahmad V Jabbar       987987987
James E Borg         888665555

4 rows selected.
```

The above query only lists those who are not in the list of employees who work on at least one of projects with project number 1, 2, or 3.

EXISTS v.s. NOT EXISTS:

These two above queries can be implemented with the functions such as EXISTS and NOT EXISTS.

To retrieve both the names and the social security numbers of employees who work on at least one of projects with the project number 1, 2, or 3.

```
SQL> select e.name, e.ssn
  2  from   jps.employee e
  3  where  exists (select *
  4                 from   jps.workson w
  5                 where  w.pno  in (1,2,3)
  6                  and   e.ssn = w.essn);


NAME                 SSN
-------------------  ---------
John B Smith         123456789
Franklin T Wong      333445555
Ramesh K Narayan     666884444
Joyce A English      453453453

4 rows selected.
```

To retrieve both the names and social security numbers of employees who do not work on any one of projects with the project number 1, 2, and 3.

```
SQL> select e.name, e.ssn
  2  from   jps.employee e
  3  where  not exists     (select    *
  4                         from      jps.workson w
  5                         where     w.pno  in (1,2,3)
  6                         and       e.ssn = w.essn);
```

```
NAME                 SSN
-------------------  ---------
Alicia J Zelaya      999887777
Jennifer S Wallace   987654321
Ahmad V Jabbar       987987987
James E Borg         888665555

4 rows selected.
```

These two query statements also involve the join operation, but they do perform the select operation first before the join operation. That is the reason why the result table does not contain the duplicates.

The expression `w.pno in (1,2,3)` in the where clause of the inner query is the `select` condition, and the expression `e.ssn = w.essn` is the join condition that correlates records from both the `workson` table in the inner subquery and the employee table in the outer subquery by the join condition.

DISTINCT to Remove the Duplicates:

In some situation, the duplicates are inevitable and it is necessary to remove these duplicates. In order to remove duplicates in a query, the keyword `DISTINCT` can be used in the front of the column name in the `select` clause as the final filtering operation before the final result is displayed.
For example, there is a query **to list the social security numbers of supervisors in the employee table**. It is unnecessary to list the duplicates of the social security numbers in the result table.

```
SQL> select distinct (e.superssn)
  2  from   jps.employee e;


SUPERSSN
---------
333445555
888665555
987654321

3 rows selected.
```

- If the keyword distinct is not used, then the result table will contain the duplicates. Removing duplicates in a relation table will incur extra cost.

`LIKE` Operator:

The `LIKE` operator can be used for pattern-matching in the select condition of a select statement.

For example, **to retrieve the employees whose last name is Wong**, the following select condition can be used to do pattern-matching in the query.

```
SQL> select      *
  2   from       jps.employee e
  3  where       e.name like '%Wong';
```

```
NAME                SSN          BDATE    SEX   SALARY   SUPERSSN   DNO
------------------  ---------  ----------  ----  -------  ---------  ---
Franklin T Wong     333445555  08-DEC-45   M     40000    888665555  5

1 rows selected.
```

The character "%" is called a wild card, which matches any string of zero or more characters.

Similarly, you could use the wild card character "%" to retrieve the employees whose names contain the capital letter "W" as the following:

```
SQL> select      *
  2   from       jps.employee e
  3  where       e.name like '%W%';

NAME                SSN          BDATE    SEX   SALARY   SUPERSSN   DNO
------------------  ---------  ----------  ----  -------  ---------  ---
Franklin T Wong     333445555  08-DEC-45   M     40000    888665555  5
Jennifer S Wallace  987654321  20-JUN-31   F     43000    888665555  4

2 rows selected.
```

The first wild card character '%' matches in the `where` clause any string of zero or more characters before the capital letter 'W', and the second wild card character '%' matches any string of zero or more characters after the capital letter 'W'.

PROJECT Operation:

The project operation in the relational database model is to retrieve some of column data in a table but not necessary all of the columns.

To retrieve the name, social security number, and salary of employees from department number 5. After the prompt `SQL>`, type in the statement as follows:

```
SQL> select      name, ssn, salary
     from        jps.employee
     where       dno = 5;

NAME              SSN         SALARY
----------------  ----------  -------
John B Smith      123456789   30000
Franklin T Wong   333445555   40000
Ramesh K Narayan  666884444   38000
Joyce A English   453453453   25000

4 rows selected.
```

The result table from the query contains the data from the columns such as NAME, SSN, and SALARY for those employees in department number 5.

JOIN Operation:

The join operation in the relational database model is used to combine the related data from two or more relational tables. For example, in order to retrieve the name of the manager for each department, the query requires combining data from both `employee` and `department` tables. The `employee` table contains all the employees information such as `NAME`, `SSN` (social security number), `SEX`, `SALARY`, `DNO` (the department for which the employee works). The `department` table contains the information for each department such as `DNUMBER` (the department number), `DNAME` (the department name), `MGRSSN` (the social security number of the department's manager), and so on. In this database model, the manager's name and his/her other information are kept in the `employee` table since the manager of a department is also treated as an employee.

Before answering the query, it is a good idea to look at data in both `jps.employee` and `jps.department` tables.

Type in the following statement after the prompt `SQL>` to list the data in the `jps.department` table.

```
SQL>  select      *
  2   from        jps.department;
```

DNAME	DNUMBER	MGRSSN	MGRSTARDA
Research	5	333445555	22-MAY-78
Administration	4	987654321	01-JAN-85
Headquarters	1	888665555	19-JUN-71

```
3 rows selected.
```

Type in the following statement after the prompt `SQL>` to list the data in the `jps.employee` table.

```
SQL>  select      *
  2   from        jps.employee;

NAME                SSN          BDATE    SEX   SALARY   SUPERSSN   DNO
------------------  ---------  ----------  ----  -------  ---------  ---
John B Smith        123456789  09-JAN-55   M     30000    333445555  5
Franklin T Wong     333445555  08-DEC-45   M     40000    888665555  5
Alicia J Zelaya     999887777  19-JUL-85   F     25000    987654321  4
Jennifer S Wallace  987654321  20-JUN-31   F     43000    888665555  4
Ramesh K Narayan    666884444  15-SEP-52   M     38000    333445555  5
Joyce A English     453453453  31-JUL-62   F     25000    333445555  5
Ahmad V Jabbar      987987987  29-MAR-59   M     25000    987654321  4
James E Borg        888665555  10-NOV-27   M     55000               1
```

```
8 rows selected.
```

To retrieve the department number, the department name, and both the name and the social security number of the department's manager, the following select statement can be used.

```
SQL>  select     dnumber, dname, mgrssn, name
  2   from       jps.department, jps.employee
  3   where      mgrssn = ssn;

DNUMBER     DNAME              MGRSSN            NAME
----------  ---------------    ----------------  --------------------
5           Research           333445555         Franklin T Wong
4           Administration     987654321         Jennifer S Wallace
1           Headquarters       888665555         James E Borg

3 rows selected.
```

In the `select` statement for this query, the expression `mgrssn = ssn` specifies the join condition (equi-join condition) between the `jps.department` and `jps.employee` tables.

Please note that the data items such as `DNUBMER`, `DNAME`, and `MGRSSN` are from the `jps.department` table, and the data item `NAME` is from the `jps.employee` table. In order to denote the column names explicitly, the aliasing mechanism can be used as in the following select statement:

```
SQL>  select     d.dnumber, d.dname, d.mgrssn, e.name
  2   from       jps.department d, jps.employee e
  3   where      d.mgrssn = e.ssn;

 DNUMBER         DNAME           MGRSSN            NAME
---------------  ---------------------------  ----------------  ---------------------------------
5                Research                     333445555         Franklin T Wong
4                Administration               987654321         Jennifer S Wallace
1                Headquarters                 888665555         James E Borg

3 rows selected.
```

In the above select statement, `d` is the alias of `jps.department`, and `e` is the alias of `jps.employee`.

By using alias, you could join the table itself for certain applications. For example, the application query requires retrieving all the employees and their corresponding supervisors (here it is assumed that each employee has at most one supervisor, but a supervisor may supervises several supervisees)

Before writing the SQL query statement, it is better to print out the `jps.employee` table again to determine the join columns (sometime, it is called a join path).

```
SQL> select * from jps.employee;
```

EMPLOYEE

```
NAME                 SSN          BDATE    SEX   SALARY   SUPERSSN   DNO
------------------   ---------   ---------- ----  -------  ---------  ---
John B Smith         123456789   09-JAN-55  M     30000    333445555  5
Franklin T Wong      333445555   08-DEC-45  M     40000    888665555  5
Alicia J Zelaya      999887777   19-JUL-85  F     25000    987654321  4
Jennifer S Wallace   987654321   20-JUN-31  F     43000    888665555  4
Ramesh K Narayan     666884444   15-SEP-52  M     38000    333445555  5
Joyce A English      453453453   31-JUL-62  F     25000    333445555  5
Ahmad V Jabbar       987987987   29-MAR-59  M     25000    987654321  4
James E Borg         888665555   10-NOV-27  M     55000               1

8 rows selected.
```

In the `jps.employee` table, the column `SSN` and the column `SUPERSSN` have the same domain, and these two columns can be used as join columns (sometimes called the join path). The set of values in the column `SUPERSSN` is the subset of the values in the column SSN since not every employee is a supervisor.

To retrieve the pairs of `NAME`, `SSN` for both employees and their supervisors, the following query can be used:

```
SQL>  select     e.ssn, e.name, s.ssn, s.name
  2   from       jps.employee e, jps.employee s
  3   where      e.superssn = s.ssn;


     SSN              NAME              SSN              NAME
------------   -------------------  ------------   -------------------
123456789      John B Smith         333445555      Franklin T Wong
333445555      Franklin T Wong      888665555      James E Borg
999887777      Alicia J Zelaya      987654321      Jennifer S Wallace
987654321      Jennifer S Wallace   888665555      James E Borg
666884444      Ramesh K Narayan     333445555      Franklin T Wong
453453453      Joyce A English      333445555      Franklin T Wong
987987987      Ahmad V Jabbar       987654321      Jennifer S Wallace

7 rows selected
```

By using alias, it is possible to imagine there are two copies of the `jps.employee` table, one is the copy as the `employee`, and another is the copy as the `supervisor`.
In the from clause: `from jps.employee e, jps.employee s`

The `jps.employee` table is aliased as an employee copy (via the expression `jps.employee e`) and a supervisor copy (via the expression `jps.employee s`).

By specifying the join condition in the where clause where `e.superssn = s.ssn`, the value of `e.superssn` in the employee copy of the `jps.employee` table is used to match the value of `s.ssn` in the supervisor copy of the `jps.employee` table, which is the social security number of the corresponding supervisor.

OUTER-JOIN Operation:

There is an inherited problem from the join (inner join) operation. If you look at the result table from the execution of the query statement. In the original `EMPLOYEE` table, there are 8 employee records. As a correct query, the system should also display 8 records of employees. The reason why the system does not display the employee record, `James E Borg`, is that the person whose name is `James E Borg` has no supervisor. There is no value in the entry `superssn` for the record of `James E Borg.` The regular join (sometimes called inner join) operation will not match any `NULL` values in the join columns.

In order to display both the social security number and name of both the employee and his/her supervisor, even though the employee has no supervisor, we could use another type of join called outer-join to solve this type of problems.

The query can be stated as the following:

To retrieve the social security numbers and the names of each employee and his/her corresponding supervisor. The corresponding SQL statement for the query is written as follows:

```
SQL>  select e.ssn, e.name, s.ssn, s.name
  2  from   jps.employee e, jps.employee s
  3   where  e.superssn = s.ssn (+);


    SSN             NAME             SSN             NAME
----------  -------------------  ----------  -------------------
123456789   John B Smith         333445555   Franklin T Wong
333445555   Franklin T Wong      888665555   James E Borg
999887777   Alicia J Zelaya      987654321   Jennifer S Wallace
987654321   Jennifer S Wallace   888665555   James E Borg
666884444   Ramesh K Narayan     333445555   Franklin T Wong
453453453   Joyce A English      333445555   Franklin T Wong
987987987   Ahmad V Jabbar       987654321   Jennifer S Wallace
888665555   James E Borg

8 rows selected.
```

In the where clause of the above query statement as the following:

where `e.superssn = s.ssn (+);`

it specifies the left-outer-join operation, i.e., the left-outer-join operation will keep all the records in the table on the left side of the join expression. The left-outer join expression is divided into left-hand side and right side by the symbol '='. Here, the `jps.employee` table as the role of employee is on the left side of the left-outer-join expression, and the `jps.employee` table as the role of supervisor is on the right side of the left-outer-join expression. The symbol `(+)` indicates that all the records from the employee table specified on the left-hand side will be kept and the `NULL` value will be padded if there is no matching record from the table as the role of supervisor on the right-hand side. In this case, the query will display the record for `James E Borg`, and the `NULL` values will be padded for the corresponding column entries of James E Borg's supervisor.

Please note that the left-outer-join is symmetric to right-outer-join. For example, the result of operation that the `employee` table as the role of employee left-outer-joins the employee table as the role of supervisor is equivalent to the result of operation that the employee table as the role of supervisor right-outer-joins the employee table as the role of the employee.

All the following expressions are equivalent:

```
where  e.superssn = s.ssn (+);
```

The where clause states that the employee table as the role of employee left-outer-joins the employee table as the role of supervisor.

```
where  s.ssn (+) = e.superssn;
```

The where clause states that the employee table as the role of supervisor right-outer-joins the employee table as the role of employee.

You may want to try the following statement to see whether these two query statements give the equivalent result.

```
SQL>  select e.ssn, e.name, s.ssn, s.name
  2  from   jps.employee e, jps.employee s
  3   where  s.ssn (+) = e.superssn;
```

Look at the following query statement, it states that the employee table as the role of supervisor left-outer-joins the employee table as the role of the employee.

The semantic meaning of the statement is to find the supervisees for each employee, i.e., to retrieve the social security number and the name of each employee, and the corresponding social security number and the name of his/her supervisees if any. (Otherwise, just display the employee's social security number and name by himself/herself.)

```
SQL>  select     s.ssn      as supervisorssn,
  2              s.name     as supervisorname,
  3              e.ssn      as employeessn,
  4              e.name     as employeename
  5   from       jps.employee e, jps.employee s
  6   where      s.ssn = e.superssn (+);
```

SUPERVISORSSN	SUPERVISORNAME	EMPLOYEESSN	EMPLOYEENAME
123456789	John B Smith		
333445555	Franklin T Wong	123456789	John B Smith
333445555	Franklin T Wong	666884444	Ramesh K Narayan
333445555	Franklin T Wong	453453453	Joyce A English
453453453	Joyce A English		
666884444	Ramesh K Narayan		
888665555	James E Borg	333445555	Franklin T Wong
888665555	James E Borg	987654321	Jennifer S Wallace
987654321	Jennifer S Wallace	999887777	Alicia J Zelaya
987654321	Jennifer S Wallace	987987987	Ahmad V Jabbar
987987987	Ahmad V Jabbar		

```
999887777      Alicia J Zelaya

12 rows selected.
```

1. In the select clause of the above query statement such as:

```
select   s.ssn      as supervisorssn,
         s.name     as supervisorname,
         e.ssn      as employeessn,
         e.name     as employeename
```

   the column name `s.ssn` will be displayed as `SUPERVISORSSN` by using the keyword as, the column name `s.name` will be displayed as `SUPERVISORNAME`, the column name `e.ssn` will be displayed as `EMPLOYEESSN`, and the column name `e.name` will be displayed as `EMPLOYEENAME`.

2. In the result table from the execution of the query statement, the first two columns in the table means the social security number and the name of the employee as the role of a supervisor, the third and the fourth columns in the table means the social security number and the name of the employee's corresponding supervisees if there is one, otherwise `NULL` values are padded.

3. In the result table from the execution of this query statement, only `James E Borg`, `Franklin T Wong`, and `Jennifer S Wallace` have a list of supervisees. The other employees from the original employee table have no supervisees, whose corresponding supervisee columns are padded with `NULL` values.

4. `James E Borg` has both `Jennifer S Wallace` and `Franklin T Wong` as supervisees;

   `Jennifer S Wallace` has both `Ahmad V Jabbar` and `Alicia J Zelaya` as supervisees;

   `Franklin T Wong` has three supervisees such as `John B Smith`, `Ramesh K Narayan`, and `Joyce A English`.

   Please keep in mind that there are the supervising relationships both from `James E Borg` to `Jennifer S Wallace` and from `Jennifer S Wallace` to `Ahmad V Jabbar` and to `Alicia J Zelaya`, which implies the transitive supervising relationship from `James E Borg` to `Ahmad V Jabbar` and to `Alicia J Zelaya`.

   There also exist the supervising relationships both from `James E Borg` to `Franklin T Wong` and from `Franklin T Wong` to `John B Smith`, to `Ramesh K Narayan`, and to `Joyce A English`, which also implies the transitive supervising relationships from `James E Borg` to `John B Smith`, to `Ramesh K Narayan`, and to `Joyce A English`. The transitive supervising relationships will be encountered again in the later part.

Consider the query as following:

To retrieve the social security numbers of all direct supervisees of `James E. Borg`.

By giving the following SQL select statement:

```
SQL> select e.ssn
  2  from   jps.employee e
  3  where  e.superssn in
  4          (select  s.ssn
  5           from    jps.employee s
  6           where   s.name = 'James E Borg');
```

The result from the query statement:

```
     SSN
  ---------
  333445555
  987654321

2 rows selected.
```

James E. Borg has two direct supervisees whose social security numbers are 333445555 (Franklin T Wong) and 987654321 (Jennifer S Wallace). In the above select query statement, the alias e means the employee role and the alias s means supervisor role.

In the above SQL select query statement, if the name 'James E Borg' is replaced by 'Franklin T Wong' in the where clause of the select statement, you will be able to retrieve the supervisees of Franklin T Wong.

```
SQL>  select e.ssn
  2  from   jps.employee e
  3  where  e.superssn in
  4          (select  ssn
  5           from    jps.employee
  6           where   name = 'Franklin T Wong');


     SSN
  ---------
  123456789
  666884444
  453453453

3 rows selected.
```

Franklin T Wong has three supervisees whose social security numbers are 123456789 (John B Smith), 666884444 (Ramesh K Narayan), and 453453453 (Joyce A English).

As the same, if you replace the name 'Franklin T Wong' with 'Jennifer S Wallace' in the where clause, you could retrieve the direct supervisees of Jennifer S Wallace.

```
SQL> select      e.ssn
  2  from        jps.employee e
  3              where      superssn in
  4              (select    ssn
  5               from      jps.employee s
  6               where     s.name = 'Jennifer S Wallace');
```

```
SSN
---------
999887777
987987987

2 rows selected.
```

`Jennifer S Wallace` has two direct supervisees whose social security numbers are `999887777` (`Alicia J Zelaya`) and `987987987` (`Ahmad V Jabbar`).

From all of three above queries, you will be able to find out

- James E. Borg has two direct supervisees whose social security numbers are `333445555` (`Franklin T Wong`) and `987654321` (`Jennifer S Wallace`).

- James E. Borg has five indirect supervisees,
  three of them are the direct supervisees of Franklin T Wong and two of them are the direct supervisees of Jennifer S Wallace.

- In the employee table, there are only three levels in the supervision hierarchy.

  The first level of supervision is the supervisor, James E Borg who has no supervisor.

  The second level includes the supervisors such as Franklin T Wong and Jennifer S Wallace, who are the direct supervisees of James E Borg.

  The third level includes both the direct supervisees of Franklin T Won, such as `123456789` (`John B Smith`), `666884444` (`Ramesh K Narayan`), and `453453453` (`Joyce A English`) and the direct supervisees of `Jennifer S Wallace` such as `999887777` (`Alicia J Zelaya`) and `987987987` (`Ahmad V Jabbar`).

  All of supervisees from the third level have no supervisees.

UNION Operation:

In order to retrieve all of direct and indirect supervisees of James E Borg in the employee table, the `UNION` operator can be used for this purpose as the followings:

```
SQL> select e1.ssn
  2  from   jps.employee e1
  3  where  e1.superssn in
  4          (select  s1.ssn
  5           from    jps.employee s1
  6           where   s1.name = 'James E Borg')
```

```
 7  union
 8  select e2.ssn
 9  from   jps.employee e2
10  where  e2.superssn in
11         (select   s2.ssn
12          from     jps.employee s2
13          where   s2.name = 'Franklin T Wong')
14  union
15  select e3.ssn
16  from   jps.employee e3
17  where  e3.superssn in
18          (select  s3.ssn
19           from    jps.employee s3
20           where   s3.name = 'Jennifer S Wallace');
```

```
    SSN
---------
123456789
333445555
453453453
666884444
987654321
987987987
999887777

7 rows selected.
```

There are three subqueries connected by two `UNION` operators in the above select statement:

- The first subquery from line 1 to 6 retrieves all the direct supervisees of James E Borg.

  The second subquery from line 8 to 13 retrieves the direct supervisees of Franklin T Wong who is the direct supervisee of James E Borg.

  The third retrieves the direct supervisees of Jennifer S Wallace who is the direct supervisees of James E Borg.

- The unions of three subqueries give all of direct and indirect supervisees of James E Borg.

  From the first level of supervision relationship to the second level of the supervision relationship, there are exactly two branches in this supervision hierarchy of the employee data:
  One is from James E Borg to Franklin T Wong, and another is from James E Borg to Jennifer S Wallace.

  Two union operators have been used to retrieve all of supervisees (both direct and indirect) of James Borg. N union operators will be required if there are N branches in the supervision hierarchy.

In order to solve this type of problems, the following select query statement can be used to retrieve the supervisees of James E Borg at both the second level and the third level of the supervision relationships no matter how many branches exist.

```
SQL> select e.ssn
  2  from   jps.employee e
  3  where  e.superssn in
  4          (select  s.ssn
  5           from    jps.employee s
  6           where   s.name = 'James E Borg')
  7  union
  8  select e1.ssn
  9  from   jps.employee e1
 10  where  e1.superssn in
 11          (select s1.ssn
 12           from   jps.employee s1
 13                  where  s1.superssn in
 14                         (select s2.ssn
 15                          from   jps.employee s2
 16                          where  s2.name = 'James E Borg'));


     SSN
  ---------
  123456789
  333445555
  453453453
  666884444
  987654321
  987987987
  999887777

7 rows selected.
```

The subquery from line 1 to line 6 retrieves all direct supervisees of James E Borg at the second level of the supervision relationship.

```
SQL> select e.ssn
  2  from   jps.employee e
  3  where  e.superssn in
  4          (select  s.ssn
  5           from    jps.employee s
  6           where   s.name = 'James E Borg');
```

The subquery gives the result as the following table:

```
     SSN
  ---------
  333445555
  987654321

2 rows selected.
```

The subquery from line 8 to 16 retrieves all the indirect supervisees of James E. Borg at the third level of the supervision relationship.

```
  8  select e1.ssn
```

```
  9  from   jps.employee e1
 10  where  e1.superssn in
 11          (select s1.ssn
 12           from   jps.employee s1
 13                  where  s1.superssn in
 14                         (select s2.ssn
 15                          from   jps.employee s2
 16                          where  s2.name = 'James E Borg'));
```

The result table from this subquery includes the supervisees of both `Franklin T Wong` and `Jennifer S Wallace`.

```
     SSN
---------
123456789
453453453
666884444
987987987
999887777

5 rows selected.
```

Retrieving all the indirect supervisees of James E Borg at the third level of the suerpvision relationship can be done in a nested subquery which includes three subqueries. This is corresponding to the number of levels in the supervision hierarchy.

By using the above subquery, it is possible to reduce the number of `UNION` operations with respect to the number of branches at each level of the supervision relationship. But there are still some problems if the number of levels in the supervision relationship is unknown. In the employee table of the above example, there are only two levels in the supervision hierarchy. The only one `UNION` operator is used to combine the supervisees at both the second level and the third level of the supervision relationship. In general, N-1 `UNION` operators will be required if there are N levels in the supervision hierarchy.

It is not computationally feasible for the following reasons.

- The first, it is assumed that the number of levels in a supervision hierarchy is unknown in most dynamic database.
- The second, it is hard to write the SQL select query statement in advance if the number of levels is unknown, i.e., the number of `UNION` operators can not be determined in advance.

- The third, the number of levels in the largest nested subquery can not be determined in advance.

In order to solve this type of problems, SQL*PLUS of ORACLE 7 or later provide the `start with ... connect` clause to resolve this type of problems in the need of applications. The application of retrieving both direct and indirect supervisees of a supervisor at all levels of the supervision relationship is typically called transitive closure.

START WITH ... CONNECT BY Operator

To retrieve all the supervisees of James E Borg at all the levels of the supervision relationship, the following select statement with the `start with ... connect` by clause can be used.

```
select  lpad(' ', (level - 1) * 2) || ssn as padded_ssn
from    jps.employee
connect by prior ssn = superssn
start   with ssn in (select ssn
                     from   jps.employee
                     where  name = 'James E Borg');

SSN
---------
888665555
  333445555
    123456789
    666884444
    453453453
  987654321
    999887777
    987987987

8 rows selected.
```

Please note that the result table also includes the social security number of James E Borg itself. The advantages of the select statement with the `start with ... connect` by clause are not only that the queries of transitive closure can be answered without predicting the number of levels in the hierarchy in advance, but also that the query of transitive closure can be answered at any starting level.

For example, to retrieve all the supervisees of Franklin T Wong by using the following select query statement:

```
SQL>  select  e.ssn
   2  from    jps.employee e
   3  start   with e.name = 'Franklin T Wong'
   4  connect by e.superssn = prior e.ssn;




     SSN
  ---------
  333445555
  123456789
  666884444
  453453453

4 rows selected.
```

The result table includes also the social security numbers of both Franklin T Wong's supervisees and himself.
To apply the `start with ... connect` by clause for Jennifer S Wallace with the following the select statement:

```
SQL> select  e.ssn
  2  from    jps.employee e
  3  start   with e.name = 'Jennifer S Wallace'
  4  connect by e.superssn = prior e.ssn;


     SSN
  ---------
  987654321
  999887777
  987987987

3 rows selected.
```

The following query statement retrieves not only the supervisor's social security numbers appeared in the leftmost column of result table, but also the supervisees' social security numbers at different level the hierarchy.

```
SQL> select  lpad(' ', (level - 1) * 2) || ssn as padded_ssn
2          from    jps.employee e
3          connect by prior ssn = superssn
4          start   with ssn in (select ssn
5                    from   jps.employee
6                    where  name = 'James E Borg')


SSN
---------
888665555
  333445555
    123456789
    666884444
    453453453
  987654321
    999887777
    987987987

8 rows selected.
```

- At the first level, the person with the social security number `888665555` (`James E Borg`) has no supervisor but two supervisees whose social security number are `33344555` (`Franklin T Wong`) and `987654321` (`Jennifer S Wallace`) at the second level.

- The person with the social security number `333445555` (`Franklin T Wong`) at the second level has three supervisees whose social security numbers are `123456789` (`John B Smith`), `666884444` (`Ramesh K Narayan`), and `453453453` (`Joyce A English`) from the third level, and the person with the social security number `987654321` (`Jennifer S Wallace`) at the second level has two supervisees whose social security numbers are `999887777` (`Alicia J Zelaya`) and `987987987` (`Ahmad V Jabbar`) from the third level.

- For the detailed information about the expression, `LPAD(' ', 9*(Level - 1)) || ssn ssn`, please make reference to ORACLE: The Complete Reference, 3rd Edition, by George Koch and Kevin Loney.

INTERSECT

The INTERSECT operation retrieves some records with certain common properties. For example, **to retrieve the social security numbers of employees who have dependents**. The query can be answered by the following query statement with the INTERSECT operator.

```
SQL> select e.ssn
  2  from   jps.employee e
  3  intersect
  4  select d.essn
  5  from   jps.dependent d;


     SSN
  ---------
  123456789
  333445555
  987654321

3 rows selected.
```

MINUS

To find the social security numbers of employees who have no dependents. You could use the set operator MINUS to exclude those employees who have dependents from the employee table. The result table from the following query statement will only include the social security numbers of these employees who are in the employee table, but not in the dependent table.

```
SQL> select      e.ssn
  2  from        jps.employee e
  3  minus
  4  select      d.essn
  5  from        jps.dependent d;


     SSN
  ---------
  453453453
  666884444
  888665555
  987987987
  999887777


5 rows selected.
```

DIVISION

The division operation is commonly used to check the set containment type query such as to find the employee who works on all the projects controlled by department #5. An employee could work a set of projects, and the department #5 could control or manage a list of projects. The division operation will be used to test whether the set of projects that the employee works on includes all projects in the set of projects controlled by department #5. Here, the word '**all**' implies every project, but not some

of the projects controlled by department #5. The set of projects worked by the employee can be a superset of the set of projects controlled by department #5. In general, the query can be stated as "**retrieve the employee** (social security number and name) **who works on all the project controlled by the department 5**."

```
select      ssn,name
from        jps.employee e
where       not exists ((select         pnumber
                         from           jps.project
                         where          dnum = 5)
                        minus
                        (select         pno
                         from           jps.workson w
                         where          e.ssn = w.essn));

no row selected
```

Department 5 controls projects with the project number 1, 2, and 3. The set of projects controlled by department 5 is returned by the following query:

```
select      pnumber
from        jps.project
where       dnum = 5;

PNUMBER
--------
1
2
3
```

The `workson` table shows that there is no employee who works on all the projects controlled by department. With respect to the data sets in the project table, `workson` table, and employee table, the result from the division query is empty. Actually, the SQL statement can be phrased as "**there does not exist a project controlled by department 5 that the employee does not work on**." The SQL statement actually uses the double negation to implement the query: **retrieve the employee who works on all the projects controlled by Department #5**.
The syntax structure of the SQL statement shows the query logic as follows:

The first inner subquery

```
select      pnumber
from        jps.project
where       dnum = 5;
```

returns the set of projects controlled by Department 5 as follows:

```
PNUMBER
--------
1
2
3
```

The second inner (correlated) subquery

```
select     pno
from       jps.workson w
where      e.ssn = w.essn)
```

retrieves a set of projects each employee works on at a time. The join condition `e.ssn = w.essn` correlates an employee tuple from table `jps.employee` in the outer subquery one at a time for the testing purpose.
If the employee tuple under current evaluation works on all the projects controlled by Department 5 or the set of projects on which the employee works includes all the projects controlled by Department 5. The result from the first inner subquery MINUS the result from the second inner subquery will be empty based on the definition of set-oriented difference operation, then the function NOT EXISTS will be evaluated to be true, the employee under current evaluation will be displayed as the part of query result. If neither the employee work on all the projects controlled by Department 5, nor the set of projects on which the employee works includes the set of projects controlled by Department, then the function NOT EXISTS will be evaluated to be false because the result of MINUS operation between the first inner subquery and the second inner subquery will not be empty. So the employee tuple under current evaluation will fail the test, and the employee will not be displayed as part of the query result.

To retrieve the employee who works on all the projects that John Smith works on, the query can be formulated as follows:

```
select     ssn,name
from       jps.employee  e
where      not exists ((select   pno
                        from     jps.workson w1, jps.employee  e1
                        where    e1.name = 'John B Smith'
                        and      e1.ssn = w1.essn)
                       minus
                       (select   pno
                        from     jps.workson w
                        where    e.ssn = w.essn));
```

```
SSN       NAME
--------- -------------------
123456789 John B Smith
453453453 Joyce A English
```

Many practical applications require the division operation written in SQL, for example, to find the students who have completed all the prerequisites; list the students who passed all the exams required by their respective study plan; etc.

For the following query statement, can you describe what the SQL statement will retrieve and stand for?

```
select     ssn,name
from       jps.employee e
```

```
where      not exists ((select        pno
                        from           jps.workson w
                        where          e.ssn = w.essn)
                       minus
                       (select         pnumber
                        from           jps.project
                        where          dnum = 5))
```

```
SSN       NAME
--------- -------------------
123456789 John B Smith
666884444 Ramesh K Narayan
453453453 Joyce A English
```

New ANSI Syntax for Joins and Outer-Joins Supported by Oracle 9i and Later Version:

The new ANSI Standard syntax allows specifying join operations such as: CROSS JOIN (actually it is the CARTESIAN PRODUCT operation), JOIN (THETA JOIN), NATURAL JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN, FULL OUTER JOIN, etc. in the `from` clause of a `select` statement.

Assuming two following given tables R and S:

```
SQL> select *
  2  from jps.R;

A  B  C
-- -- --
a1 b1 c1
a1 b2 c2

SQL> select *
```

```
  2  from jps.S;

B  C  D
-- -- --
b1 c1 d1
b1 c3 d2
```

We will illustrate these SQL statements with the new syntax supported by Oracle10g in the following:

CROSS JOIN: (CARTESIAN PRODUCT)

```
SQL> select *
  2  from   jps.R  cross join jps.S;

A  B  C  B  C  D
-- -- -- -- -- --
a1 b1 c1 b1 c1 d1
a1 b2 c2 b1 c1 d1
a1 b1 c1 b1 c3 d2
a1 b2 c2 b1 c3 d2
```

JOIN: (THETA JOIN on R.B = S.B)

```
SQL> select *
  2  from   jps.R join jps.S on jps.R.B = jps.S.B;

A  B  C  B  C  D
-- -- -- -- -- --
a1 b1 c1 b1 c1 d1
a1 b1 c1 b1 c3 d2
```

JOIN: (THETA JOIN on R.C = S.C)

```
SQL> select *
  2  from  jps.R join jps.S on jps.R.C = jps.S.C;

A  B  C  B  C  D
-- -- -- -- -- --
a1 b1 c1 b1 c1 d1
```

INNER JOIN: (Same as THETA JOIN)

```
SQL> select *
  2  from   jps.R inner join jps.S on jps.R.C = jps.S.C;

A  B  C  B  C  D
-- -- -- -- -- --
a1 b1 c1 b1 c1 d1
```

NATURAL JOIN:

```
SQL> select *
  2  from  jps.R natural join jps.S;


B  C  A  D
-- -- -- --
b1 c1 a1 d1
```

or use the following select statement to explicitly order the attribute list in display:

```
SQL> select A, B, C, D
  2  from   jps.R natural join jps.S;


A  B  C  D
-- -- -- --
a1 b1 c1 d1
```

LEFT OUTER JOIN on R.B = S.B: (Based on THETA JOIN definition)

```
SQL> select *
  2  from   jps.R left outer join jps.S on jps.R.B = jps.S.B;


A  B  C  B  C  D
-- -- -- -- -- --
a1 b1 c1 b1 c1 d1
a1 b1 c1 b1 c3 d2
a1 b2 c2
```

LEFT OUTER JOIN on R.C = S.C: (Based on THETA JOIN definition)

```
SQL> select *
  2  from   jps.R left outer join jps.S on jps.R.C = jps.S.C;


A  B  C  B  C  D
-- -- -- -- -- --
a1 b1 c1 b1 c1 d1
a1 b2 c2
```

RIGTHT OUTER JOIN on R.B = S.B: (Based on THETA JOIN definition)

```
SQL> select *
  2  from  jps.R right outer join jps.S on jps.R.B = jps.S.B;


A  B  C  B  C  D
-- -- -- -- -- --
```

```
a1 b1 c1 b1 c3 d2
a1 b1 c1 b1 c1 d1
```

RIGTHT OUTER JOIN on R.C = S.C: (Based on THETA JOIN definition)

```
SQL> select *
  2  from  jps.R right outer join jps.S on jps.R.C = jps.S.C;

A  B  C  B  C  D
-- -- -- -- -- --
a1 b1 c1 b1 c1 d1
         b1 c3 d2
```

FULL OUTER JOIN ON R.B = S.B: (Based on THETA JOIN definition)

```
SQL> select *
  2  from   jps.R full outer join jps.S on jps.R.B = jps.S.B;

A  B  C  B  C  D
-- -- -- -- -- --
a1 b1 c1 b1 c1 d1
a1 b1 c1 b1 c3 d2
a1 b2 c2
```

FULL OUTER JOIN ON R.C = S.C: (Based on THETA JOIN definition)

```
SQL> select *
  2  from  jps.R full outer join jps.S on jps.R.C = jps.S.C;


A  B  C  B  C  D
-- -- -- -- -- --
a1 b1 c1 b1 c1 d1
a1 b2 c2
         b1 c3 d2
```

FULL OUTER JOIN ON R.B = S.B and R.C = S.C:

```
SQL> select *
  2  from  jps.R full outer join jps.S
  3  on    jps.R.B = jps.S.B and jps.R.C = jps.S.C;

A  B  C  B  C  D
-- -- -- -- -- --
a1 b1 c1 b1 c1 d1
a1 b2 c2
         b1 c3 d2
```

Aggregate Functions:

AVG, COUNT, MAX, MIN, and SUM

The above list of aggregate functions such as AVG, COUNT, MAX, MIN, and SUM provide group values for various purposes.

1. The functions such as AVG, MAX, MIN, and SUM calculate the average, maximum, minimum, and summation of value(s) in a column respectively. The function COUNT can be used to count the number of entries in a column.

2. The functions such as AVG and SUM can be only applied to the columns of numerical data type, whereas the functions such as MAX, MIN, and COUNT can be applied to the column of either numerical data type or character data type.

3. The functions such as AVG and SUM will discard any NULL values in a column before the calculation, whereas the functions such as MAX, MIN, and COUNT do take the consideration of the NULL values in a column.

To calculate the average, maximum, minimum, and total salary from the EMPLOYEE table, the following select statement can be used:

```
SQL>  select avg(salary),min(salary),max(salary),sum(salary)
 2    from   jps.employee;

   AVG(SALARY)        MIN(SALARY)        MAX(SALARY)        SUM(SALARY)
   -----------        -----------        -----------        -----------
      35125              25000              55000              281000


1 rows selected.
```

To calculate the total number of employees, the average salary, and the total salary of the department 5 in the EMPLOYEE table, the corresponding select query statement is as the following:

```
SQL> select count(ssn), avg(salary), sum(salary)
  2  from   jps.employee
  3  where  dno = 5;

     COUNT(SSN)           AVG(SALARY)           SUM(SALARY)
     ----------           -----------           -----------
         4                   33250                133000

1 rows selected.
```

Group By:

In order to retrieve the total number of employees, the average salary, and the total salary for each individual department, the group by clause can be used to serve this purpose.

```
SQL> select dno, count(ssn), avg(salary), sum(salary)
  2  from   jps.employee
  3  group  by dno;

DNO          COUNT(SSN)         AVG(SALARY)        SUM(SALARY)
------       ----------         -----------        -----------
1            1                  55000              55000
```

```
4           3                   31000             93000
5           4                   33250             133000
```

Having:

The select condition or constraint in the where clause is applied to each individual record or tuple, whereas the constraint specified in the having clause is applied to the each individual group obtained from group by clause.

For example, to retrieve the number of employees, the average salary, and the total salary of each department, only these departments have more than one employee. The following is the select query statement with the specification of the constraints applied to each individual group:

```
SQL>  select dno, count(ssn), avg(salary), sum(salary)
  2   from   jps.employee
  3   group  by dno
  4   having count(ssn) > 1;


DNO                 COUNT(SSN)          AVG(SALARY)         SUM(SALARY)
--------            ----------          -----------         -----------
4                   3                   31000               93000
5                   4                   33250               133000
```

Table Creation:

Up to now, you have experienced how to use the select statement to retrieve the data from the database. In the next, we present how to create a table and load data into the table.

To create the employee table, you could use the following create table statement:

```
create  table  EMPLOYEE
( name           varchar2(19)  not null,
  ssn            char (9),
  bdate          date,
  sex            char(3),
  salary         number(8,2),
  superssn       char(9),
  dno            varchar(8),
  constraint     empPK  primary key    (ssn));
```

There are several ways to load data into an existing table:

1. Use the insert statement to insert a single record at a time.
2. Use the insert statement to insert a list of records from either an existing table or a query.
3. Use the SQL*LOADER to load a list of records in a data file under your current directory into a database table.
4. Use a program with the embedded SQL statement to read a list of records in a data file into a database table.

In this tutorial, we will show how to use the insert statement to load the data into a database table.

To insert a single data record into a database table, the following insert statement can be used:

```
insert into EMPLOYEE values
('John B Smith','123456789','09-JAN 55','M',30000,'333445555','5');
```

- In the above insert statement, `EMPLOYEE` is the name of the table into which the record will be inserted.

- All the data items in the record are separated by a comma. The data item of character type should be enclosed with single quotes at both the beginning and the end of the character string. The list of items that comprise the data record is enclosed within the left and right parentheses.

To insert the rest of records in the `EMPLOYEE` table, use the following insert statements.

```
insert into EMPLOYEE values
('Franklin T Wong','333445555','08-DEC-45','M',40000,'888665555','5');

insert into EMPLOYEE values
('Alicia J Zelaya','999887777','19-JUL-85','F',25000,'987654321','4');

insert into EMPLOYEE values
('Jennifer S Wallace','987654321','20-JUN-31','F',43000,'888665555','4');

insert into EMPLOYEE values
('Ramesh K Narayan','666884444','15-SEP-52','M',38000,'333445555','5');

insert into EMPLOYEE values
('Joyce A English','453453453','31-JUL-62','F',25000,'333445555','5');

insert into EMPLOYEE values
('Ahmad V Jabbar','987987987','29-MAR-59','M',25000,'987654321','4');

insert into EMPLOYEE values
('James E Borg','888665555','10-NOV-27','M',55000,' ','1');
```

To insert a list of records from an existing table, for example, `jps.employee`, you could use the following insert statement:

```
insert into employee
select * from jps.employee;
```

Before you insert the list of records from the table `jps.employee` into your table EMPLOYEE, make your table EMPLOYEE empty by using the delete statement as the following since you just inserted the list of records by using the insert statement.

```
delete from employee;
```

To insert the data records from the table, `employee`, just use the following statement:

```
SQL>  insert into employee
  2   select * from jps.employee;
```

Now, you could use the select statement to view the data in the EMPLOYEE table created by yourself.

```
SQL>  select  *  from employee;
```

The Delete Statement:

To delete an employee whose name is Franklin T Wong, the delete statement is as the following:

```
delete from employee
where  name = 'Franklin T Wong';
```

To delete all the records in a table, you could use the statement in the following format:

```
delete  from    tablename
```

The delete statement deletes all the records in the table and makes the table empty.

- To look up constraints in a specific table created by yourself, you could use the following statement:

```
select owner, constraint_name
from   user_constraints
where  owner = 'JPS' and table_name = 'EMPLOYEE';
```

```
 OWNER       CONSTRAINT_NAME
---------    ---------------
   JPS          SYS_C001252
   JPS             EMPPK
   JPS           EMPDNOFRK
```

- The constraint with the name EMPPK is the primary key constraint in the employee table.

- The constraint with the name EMPDNOFRK is the foreign key constraint that references the primary key, dnumber, in the department table.

- The constraint with the system assigned name, SYS_C001252, is the NOT NULL constraint. If the user who creates the constraint without the constraint name, then the system will automatically assign a constraint name to it.

- To look up constraints in a specific table granted access to you, you might want to use the following SQL statement to view the constraints created in the table.

```
select owner, constraint_name
from   all_constraints
where  owner = 'JPS' and table_name = 'EMPLOYEE';
```

The Drop Table Statement:

In order to remove or destroy a database table, you could use the drop table statement in the following format:

```
drop table tablename
```

To remove the `EMPLOYEE` table, just simply type

```
drop table employee;
```

If there are any foreign keys in other tables referring to the primary keys of the table being dropped, this simple form of the drop table statement will cause an error, and the table will not be dropped by ORACLE DBMS server.

If the option `cascade constraints` is defined, then all the referential integrity constraints referring to keys in the dropped table are dropped.

To force a table dropped in the cases where other foreign keys refer to its keys, the following version of the `drop table` statement can be used:

```
drop table  employee  cascade constraints;
```

Please note that recent Oracle10g allows the recovery of mistakenly dropped tables. So the dropped tables will be fictitiously put into the so called recycle bin with a new name for your recovery.

If you would like to recover the mistakenly dropped table `employee`, you could use the following command:

```
flashback   table   employee  to before drop;
```

If you would like to clean the recycle bin, you could use the following command:

```
purge  recyclebin;
```

If you would like to drop table `employee` permanently, you could use the following command:

```
drop table  employee purge;
```

Any comments and suggestions will be appreciated.

Please send your comments to the following address:

Dr. Junping Sun
Graduate School of Computer and Information Sciences
Nova Southeastern University
3301 College Avenue
Fort Lauderdale, FL 33314-7796

Phone: (954) 262-2082
Fax: (954) 262-3915
Internet: jps@nsu.nova.edu

Acknowledgments:

I would like to thank many inputs, remarks, suggestions, etc. from my students and colleagues to improve this documents.

Appendix:

The appendix includes all the DDL (data definition language) statements for creating tables such as EMPLOYEE, DEPARTMENT, DEPT_LOCATION, PROJECT, WORKSON, and DEPENDENT, and the DML (data manipulation language) statements to insert each set of data records into each corresponding table.

DDL Statements to Create Tables:

To create EMPLOYEE table:

```
create  table   EMPLOYEE
(
```

```
  name           varchar2(19)  not null,
  ssn            char (9),
  bdate          date,
  sex            char(3),
  salary         number(8,2),
  superssn       char(9),
  dno            varchar2(8),
  constraint     empPK
  primary key    (ssn),
  constraint     empsuperFRK
  foreign key    (superssn)
  references     employee(ssn) disable
);
```

1. When you first create the `EMPLOYEE` table, the `DEPARTMENT` may not exist. The constraint `empdnoFRK` that references the primary key in the `DEPARTMENT` table can not be enabled. That is the reason why there is a keyword `disable`, which disables the activation of the constraint, at the end of the constraint clause.

2. After the loading data into the table, you can enable the constraint `empsuperFRK` by using the alter table statement with enable clause as the following:

   ```
   alter table EMPLOYEE enable constraint empsuperFRK;
   ```

3. By using alter table add constraint statement, you could add the referential constraint that make reference to the department table after the Department is created.

   ```
   alter table employee add
   (constraint empdnoFRK
    foreign key (dno) references department(dnumber));
   ```

   This also can be done after the data being loaded into both the employee and department table.

To create DEPARTMENT table:

```
create table DEPARTMENT
(
  dname          varchar2(15) not null,
  dnumber        varchar2(8),
  mgrssn         char(9),
  mgrstardate    date,
  constraint     departPK
  primary key   (dnumber),
  constraint     departUK
  unique         (dname),
  constraint     departFRK
  foreign key    (mgrssn)
```

```
  references    employee (ssn) on delete cascade disable
);
```

To create DEPTLOCATION table:

```
create table DEPTLOCATION
(
  dnumber       varchar2(8),
  dlocation     varchar2(15),
  constraint    dlocPK
  primary key   (dnumber, dlocation),
  constraint    dlocnoFRK
  foreign key   (dnumber)
  references    department (dnumber) on delete cascade disable
);
```

To create PROJECT table:

```
create table project
(
  pname        varchar2(15) not null,
  pnumber       varchar2(8),
  plocation     varchar2(15),
  dnum          varchar(8),
  constraint    projUK
  unique        (pname),
  constraint    projPK
  primary key   (pnumber),
  constraint    projFRK
  foreign key   (dnum)
  references    DEPARTMENT(dnumber)
 );
```

To create WORKSON table:

```
create table WORKSON
(
 essn           char(9),
 pno            varchar2(8),
 hours          number(5,1),
 constraint     workPK
 primary  key   (essn, pno),
 constraint     workssnFRK
 foreign  key   (essn)
 references     EMPLOYEE(ssn) on delete cascade disable,
 constraint     workpnoFRK
 foreign  key   (pno)
```

```
 references      PROJECT(pnumber) on delete cascade disable
);
```

To create DEPENDENT table:

```
create table DEPENDENT
(
  essn                  char(9),
  dependentname         varchar2(15),
  sex                   char(3),
  bdate                 date,
  relationship          varchar2(12),
  constraint            depenPK
  primary key           (essn, dependentname),
  constraint            depenFRK
  foreign key           (essn)
  references            EMPLOYEE (ssn) on delete cascade disable
 );
```

It is a good idea to disable the constraint at the time when you create the table and to enable the constraint after loading the data into each of the table.

DML Statements to Insert Data into Tables:

Insert the Data into the DEPARTMENT Table:

```
insert into DEPARTMENT values
('Research','5','333445555','22-MAY-78');

insert into DEPARTMENT values
('Administration','4','987654321','01-JAN-85');

insert into DEPARTMENT values
('Headquarters','1','888665555','19-JUN-71');
```

Insert the Data into the EMPLOYEE Table:

```
insert into EMPLOYEE values
('John B Smith','123456789','09-JAN-55','M',30000,'333445555','5');

insert into EMPLOYEE values
('Franklin T Wong','333445555','08-DEC-45','M',40000,'888665555','5');

insert into EMPLOYEE values
('Alicia J Zelaya','999887777','19-JUL-85','F',25000,'987654321','4');

insert into EMPLOYEE values
('Jennifer S Wallace','987654321','20-JUN-31','F',43000,'888665555','4');

insert into EMPLOYEE values
('Ramesh K Narayan','666884444','15-SEP-52','M',38000,'333445555','5');

insert into EMPLOYEE values
('Joyce A English','453453453','31-JUL-62','F',25000,'333445555','5');

insert into EMPLOYEE values
('Ahmad V Jabbar','987987987','29-MAR-59','M',25000,'987654321','4');

insert into EMPLOYEE values
('James E Borg','888665555','10-NOV-27','M',55000,' ','1');
```

Insert the Data into the DEPTLOACITON Table:

```
insert into deptlocation values ('1','Houston');

insert into deptlocation values ('4','Stafford');

insert into deptlocation values ('5','Bellaire');

insert into deptlocation values ('5','Sugarland');

insert into deptlocation values ('5','Houston');
```

Insert the Data into the PROJECT Table:

```
insert into project values ('ProductX','1','Bellaire','5');

insert into project values ('ProductY','2','Sugarland','5');

insert into project values ('ProductZ','3','Houston','5');

insert into project values ('Computerization','10','Stafford','4');

insert into project values ('Reorganization','20','Houston','1');

insert into project values ('Newbenefits','30','Stafford','4');
```

Insert the Data into the DEPENDENT Table:

```
insert into dependent values
('333445555','Alice','F','05-APR-76','Daughter');

insert into dependent values
('333445555','Theodore','M','25-OCT-73','Son');

insert into dependent values
('333445555','Joy','F','03-MAY-48','Spouse');

insert into dependent values
('987654321','Abner','M','29-FEB-32','Spouse');

insert into dependent values
('123456789','Michael','M','01-JAN-78','Son');

insert into dependent values
('123456789','Alice','F','31-DEC-78','Daughter');

insert into dependent values
('123456789','Elizabeth','F','05-MAY-57','Spouse');
```

Insert the Data into the WORKSON Table:

```
insert into workson values ('123456789','1',32.5);

insert into workson values ('123456789','2',7.5);

insert into workson values ('666884444','3',40.0);

insert into workson values ('453453453','1',20.0);

insert into workson values ('453453453','2',20.0);

insert into workson values ('333445555','2',10.0);

insert into workson values ('333445555','3',10.0);

insert into workson values ('333445555','10',10.0);

insert into workson values ('333445555','20',10.0);

insert into workson values ('999887777','30',30.0);

insert into workson values ('999887777','10',10.0);

insert into workson values ('987987987','10',35.0);

insert into workson values ('987987987','30',5.0);

insert into workson values ('987654321','30',20.0);

insert into workson values ('987654321','20',15.0);

insert into workson values ('888665555','20',NULL);
```