Oracle Database/Print version

Contents

Introduction

Oracle RDBMS is one of the most used relational database system[1]. Its request language is called the PL/SQL.

Installing Oracle

As with most software products, it must be installed; Windows, Linux and Unix versions are available for use, and there are four editions available.

- **Express Edition** (XE)download (http://www.oracle.com/technetwork/database/database-technologies/express-edition/downloads/index.html) is free but uses a slightly older version of the Oracle database engine, and has upper RAM and storage limits of 4 GB in mono-processor. It is not available for Unix and needs to register online.
- **Standard Edition One** removes the basic limitations for storage, and will support multi-cpu systems.
- **Standard Edition** (SE) provides additional features pertaining to cluster management (*Oracle Real Application Clusters*, alias *Oracle RAC*), and may be run on systems containing additional CPUs.
- **Enterprise Edition** has no limitations, and may also include optional features that are suitable to large corporations.

Once downloaded, the .zip file(s) must be extracted, if they are two their folders must be merged (they are named "database"), and launch to install:

- In Linux, *runInstaller.sh*.

- In Windows, *setup.exe*.

Starting script

Oracle Database launches automatically at each boot, which slows the system significantly. To avoid this:

- In Linux : check /etc/init.d.
- In Windows : execute *services.msc*, and toggle the services *OracleServiceXE* and *OracleXETNSListener* in manual start. When you need to use Oracle, launch as an administrator, the following script *Oracle.cmd*:
  - For the XE version:

```
net start OracleServiceXE
net start OracleXETNSListener
pause
net stop OracleXETNSListener
net stop OracleServiceXE
```

  - For the SE version:

```
net start OracleServiceORCL
net start OracleDB12Home1TNSListener
pause
net stop OracleDB12Home1TNSListener
net stop OracleServiceORCL
```

If the message "Access denied" occurs, relaunch the script with a right click, as an administrator.

Identifying system requirements

The database server needs at least[2]:

1. 1 GB free space on a hard drive for XE, 3.5 for SE.
2. 1 GB RAM.
3. Windows, Linux, Oracle Solaris, or IBM AIX.

Since the version 12c, a 64 bits processor is mandatory.

The environment variables settings are automatic.

The system objects naming follows the Optimal Flexible Architecture (OFA). [3]

References

1. http://db-engines.com/en/ranking
2. https://docs.oracle.com/html/B13614_01/preinsta.htm#i1067829
3. https://docs.oracle.com/cd/E11882_01/install.112/e47689/appendix_ofa.htm#LADBI1381

Database Interfaces

SQL*Plus

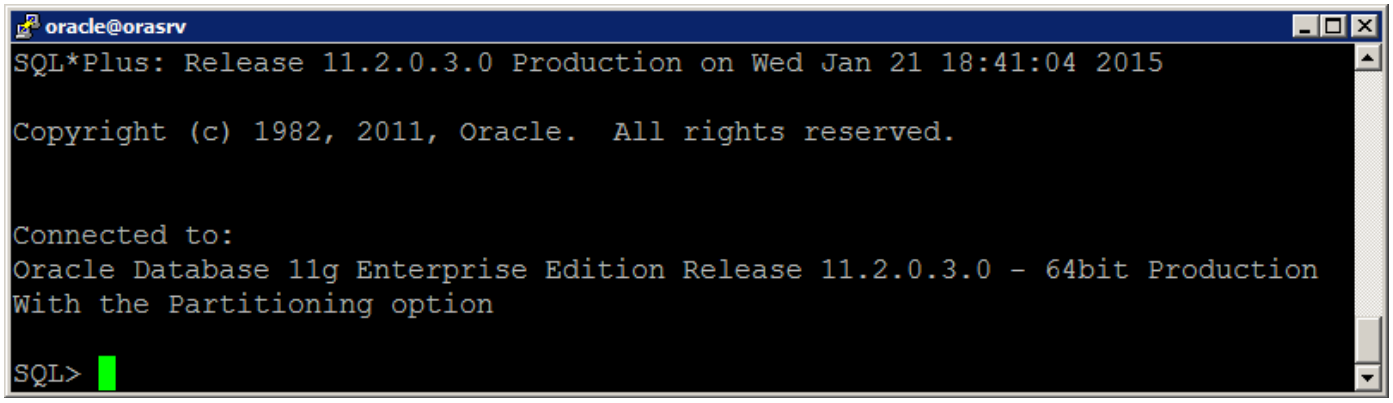
SQL*Plus is a command interface provided with the DBMS. On Windows it can be launched either from:

- The start menu, Oracle folder, *Run SQL Command Line* shortcut.
- The path C:\oraclexe\app\oracle\product\11.2.0\server\bin\sqlplus.exe.
- But the best is to use the environment variable and to connect to the DBMS at the same time, with a shell console. By default it can be done with:

```
sqlplus / as sysdba
```

Otherwise if you already have an account, the syntax is:

```
sqlplus MyAccount/MyPassword@localhost
```



The first step is to create a user (eg: root):

```
CREATE USER root IDENTIFIED BY MyPassword;
```

The second to confer him the necessary privileges:

```
GRANT sysdba to root;
```

Web interface

A second interface is provided with the DBMS: the web interface. It can be accessed from:

- For SE 12c :
  - https://localhost:5500/em/shell
- For XE 11g :
  - The start menu, Oracle folder, *Get Started* shortcut.
  - The URL http://localhost:8080/apex/f?p=4950.
- For XE 10g :
  - The start menu, Oracle folder, *Database control*.
  - http://localhost:1158/em/console/logon/logon.

Then we must connect to the DBMS:

- User name: sys (sometimes sysman)
- Password: *the one chosen during the installation*.
- Connect as: SYSDBA.

The console appears after, allowing to modify the database configuration (service restart, architecture, performances, backups...).

By clicking on *Application Express*, we can optionally create a new connection user account, which will be used to connect to Oracle. Once logged, it can access to some the database manipulation tools, for example *SQL Workshop\SQL Commands* to enter some SQL code.

Attention: if you create a first account with the GUI, they will need the default SQL*Plus account for `GRANT`.

Oracle SQL Developer

Oracle SQL Developer is an IDE developed in Java. It's provided with SE, but for XE the rich client must be downloaded on http://www.oracle.com/technetwork/developer-tools/sql-developer/overview/index.html.

Once installed and launched, we have to set a connection to let it be authenticated on the Oracle databases. Just fill the account created on the web interface, to access to the data manipulation options.

For example by clicking on the top left on *New* (the *plus* icon or CTRL + N), we can open an SQL file to begin to execute some code.

DBCA

Database Configuration Assistant (DBCA) is a graphical interface[1] available on Windows or the *nixes[2].

Hello world

Once one SQL console of those seen previously launched, it becomes possible to execute some PL/SQL (Procedural Language/Structured Query Language): the procedural language created by Oracle, and specific to its relational database.

```
BEGIN
  DBMS_OUTPUT.put_line ('Hello World!');
END;
```

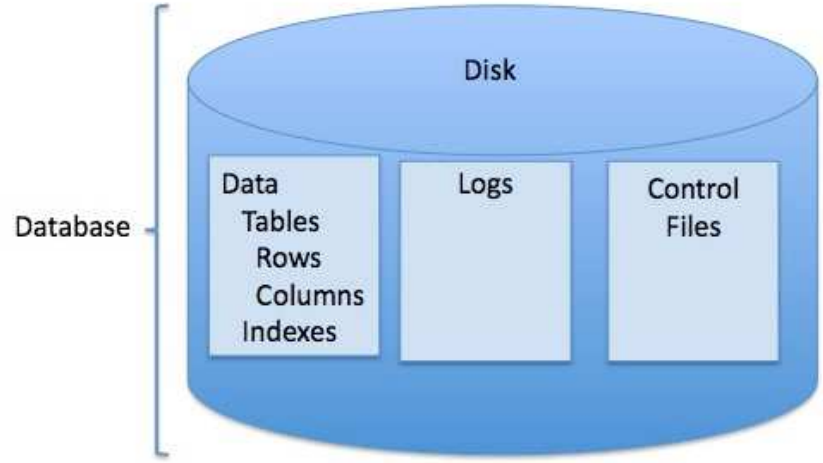
References

1. http://www.snapdba.com/2013/05/creating-an-oracle-11g-database-using-dbca-non-asm/
2. http://docs.oracle.com/cd/B16351_01/doc/server.102/b14196/install003.htm

Tables

Architecture

The Oracle architecture considers one database per server, in which we can find several tablespaces, equivalent to the MySQL and MS-SQL databases objects, containing tables and stored procedures.



In the Windows Express version, these data are stored into C:\oraclexe\app\oracle\oradata\XE.

These variables and keywords are not sensible to casing.

Create tablespaces

Once connected, it's possible to begin to create some tables directly, in the default tablespace. However before, we can add some tablespaces in some defined files:

```
   CREATE TABLESPACE Wikibooks
   DATAFILE 'C:\oraclexe\app\oracle\oradata\XE\Wikibooks.dbf' size 10M reuse
   DEFAULT STORAGE (INITIAL 10K NEXT 50K MINEXTENTS 1 MAXEXTENTS 999)
   ONLINE;
```

Create schemas

A schema is a permission accorded to a set of elements[1], like tables and stored procedures. The keyword `AUTHORIZATION` specifies the user name:

```
CREATE SCHEMA AUTHORIZATION root
  CREATE TABLE table1...
  CREATE TABLE table2...
;
```

List tables

```
SELECT owner, table_name FROM all_tables;
```

Create tables

Example:

```
CREATE TABLE client1 (last VARCHAR(10), first VARCHAR(10), address VARCHAR(20) );
```

```
Table created.
```

In SQL Developer, with a right click on the tables, *New table...*, we can generate and execute this creation in an array, which is translated into PL/SQL in the DDL tab:

```
CREATE TABLE client1
( id INT NOT NULL
, last VARCHAR2(50)
, first VARCHAR2(50)
```

```
, address VARCHAR2(255)
, CONSTRAINT client1_PK PRIMARY KEY (ID) ENABLE
) TABLESPACE Wikibooks;
```

We can also set the table tablespace, by selecting it into the GUI, or with the keyword `TABLESPACE` in the creation clause.

Available data types

The possible column types are:[2]

1. Characters:
    1. `CHAR`: 2 kB.
    2. `VARCHAR`: 4 kB.
    3. `VARCHAR2`: 4 kB, synonymous of `VARCHAR`.
    4. `NCHAR`: 2 kB.
    5. `NVARCHAR2`: 4 kB.
2. Numeric:
    1. `NUMBER`.
    2. `BINARY_INTEGER`.
    3. `BINARY_FLOAT`.
    4. `BINARY_DOUBLE`.
3. Date:
    1. `DATE`.
    2. `TIMESTAMP`.
4. `RAW`.
5. `LONG RAW`.
6. `BLOB`.
7. `CLOB`.
8. `NCLOB`.
9. `ROWID`.
10. `UROWID`.
11. `BFILE`.
12. `XMLType`.
13. `UriType`.

Reviewing the table structure

The table `ALL_TAB_COLUMNS` can provide any table column type:

```
SELECT * FROM ALL_TAB_COLUMNS WHERE TABLE_NAME = "client1"
```

Modify tables structure

Example of renaming:

```
ALTER TABLE client1 RENAME to client2
```

First field values constraint addition:

```
ALTER TABLE client1 CHECK id > 1;
```

Primary key addition:

```
ALTER TABLE client1 ADD CONSTRAINT client1_pk PRIMARY KEY (id);
```

Primary key removal:

```
ALTER TABLE client1 ADD PRIMARY KEY (id) DISABLE;
```

Foreign key addition:

```
ALTER TABLE client1
ADD CONSTRAINT fk_client2
  FOREIGN KEY (client2_id)
  REFERENCES client2(id);
```

Drop tables

```
DROP TABLE client1;
```

Insert rows

```
INSERT INTO client1 (last, first, address) VALUES ('Doe', 'Jane', 'UK'), ('Doe', 'Jane', 'UK');
```

```
1 line created.
```

Read a table

To get its structure:

```
desc client1;
```

```
Name    NULL     Type
------- -------- -------------
ID      NOT NULL NUMBER(38)
LAST             VARCHAR2(10)
FIRST            VARCHAR2(10)
ADDRESS          VARCHAR2(20)
```

If the table doesn't exist, the error which occurs is: `ORA-00923: FROM keyword not found where expected`.

To get its content:

```
SELECT * from client1;
```

```
LAST        FIRST     ADDRESS
---------- ---------- --------------------
Doe         Jane      UK
```

The number of dashes represents the field size.

Update rows

```
UPDATE client1 SET address = 'US' WHERE id = 1;
```

Delete rows

```
DELETE client1 WHERE ID = 2;
```

Partitioning

The Oracle partitioning is a process to split a huge table into several smaller ones in order to increase its performance.

Range

Example:

```
CREATE TABLE t_range
( t1      VARCHAR2(10) NOT NULL,
  t2      NUMBER       NOT NULL,
  t3      NUMBER
)
PARTITION BY RANGE (t2)
( PARTITION part1 VALUES LESS THAN (1),
  PARTITION part2 VALUES LESS THAN (11),
  PARTITION part3 VALUES LESS THAN (MAXVALUE)
);
```

Hash

Example:

```
CREATE TABLE t_hash
( t1      VARCHAR2(10) NOT NULL,
  t2      NUMBER       NOT NULL,
  t3      NUMBER
PARTITION BY HASH (t2)
PARTITIONS 4
;
```

List

Example:

```
CREATE TABLE t_list
( ort     VARCHAR2(30) NOT NULL,
  t2      NUMBER,
  t3      NUMBER
)
PARTITION BY LIST(ort)
( PARTITION part_nord VALUES IN ('Hamburg','Berlin'),
  PARTITION part_sued VALUES IN ('Muenchen', 'Nuernberg'),
  PARTITION part_west VALUES IN ('Koeln','Duesseldorf'),
  PARTITION part_ost VALUES IN ('Halle'),
  PARTITION part_def VALUES (DEFAULT)
);
```

Interval

Example:

```
CREATE TABLE t_interval
( buchungs_datum  DATE NOT NULL,
  buchungs_text   VARCHAR2(100),
  betrag          NUMBER(10,2)
)
PARTITION BY RANGE (buchungs_datum)
INTERVAL(NUMTOYMINTERVAL(1, 'MONTH'))
( PARTITION p_historie VALUES LESS THAN (TO_DATE('2014.01.01', 'YYYY.MM.DD')),
  PARTITION p_2014_01  VALUES LESS THAN (TO_DATE('2014.02.01', 'YYYY.MM.DD')),
  PARTITION p_2014_02  VALUES LESS THAN (TO_DATE('2014.03.01', 'YYYY.MM.DD'))
);
```

Explaining how constraints are created at the time of table creation

Concept In essence, constraints safeguard and validate the data.

Primary Key and Unique constraints both ensure the data is not duplicated. Primary Key also ensure the data is not null. Oracle will automatically generate index for Primary Key and Unique constraints. A table can only have one Primary Key, but

it can have multiple unique constraints.

Foreign Key ensure the data exists in the column of the parent table it refer to. Each parent record can have multiple child records, but each child can relate to ONLY one parent record. A column with Foreign Key may not necessary to have an index.

Foreign Key can only refer to column with Primary Key or Unique Constraint

```
create table tblA (colX number, colY char);
create table tblB (colX number);
```

```
alter table tblB add (constraint colX_FK foreign key (colX) references tblA(colX));
-- ORA-02270: no matching unique or primary key for this column-list
```

```
alter table tblA add (constraint colX_PK primary key (colX));
alter table tblB add (constraint colX_FK foreign key (colX) references tblA(colX));
-- alter table success.
```

A table can ONLY have one Primary Key, but it can have multiple UNIQUE key. if the child table(s) require to referencing column other than primary key, the column on the parent table must have UNIQUE constraint.

```
alter table tblA add (constraint colY_PK primary key (colY));
-- ORA-02260: table can have only one primary key
```

Cannot create PK or Unique on a column contains duplicate data

```
insert into tblA values(1,'A');
insert into tblA values(2,'A');
alter table tblA add (constraint colY_UK unique (colY));
-- ORA-02299: cannot validate (HR.COLY_UK) - duplicate keys found
```

```
delete from tblA where colx = 2;
alter table tblA add (constraint colY_UK unique (colY));
-- alter table success.
```

```
create table tblC (colY char);
alter table tblC add (constraint colY_FK foreign key (colY) references tblA(colY));
-- alter table success.
```

Insert data into a column with FK, the value must already exist in the column that the FK reference to.

```
insert into tblC values ('B');
-- ORA-02291: integrity constraint (HR.COLY_FK) violated - parent key not found
```

```
insert into tblC values ('A');
-- 1 rows inserted
```

As long as a foreign key exist, the parent table can truncate/delete the data or disable the PK or Unique constraint

```
truncate table tblA;
-- ORA-02266: unique/primary keys in table referenced by enabled foreign keys
```

Find out the constraint information in Oracle

```
desc all_constraints;
```

```
select
  a.owner, a.table_name, a.constraint_name,
  a.constraint_type, a.status, a.r_owner, a.r_constraint_name,
  b.table_name as r_table_name, b.status as r_status
from all_constraints a
  left join all_constraints b on a.owner = b.owner and a.r_constraint_name = b.constraint_name
where a.table_name like 'TBL%';
```

```
select *
from all_cons_columns
where table_name like 'TBL%';
```

Disable constraint that have foreign key refer to is not allowed, in order to do this, you have to disable the foreign key first.

```
alter table tblA disable constraint colX_PK;
-- ORA-02297: cannot disable constraint (HR.COLX_PK) - dependencies exist
alter table tblA disable constraint colY_UK;
-- ORA-02297: cannot disable constraint (HR.COLY_UK) - dependencies exist
```

```
alter table tblC disable constraint colY_FK;
alter table tblB disable constraint colX_FK;
```

```
alter table tblA disable constraint colX_PK;
alter table tblA disable constraint colY_UK;
truncate table tblA;
```

If the data in parent table is deleted, re-enable the foreign key that contain data reference to the missing data is not allowed.

```
select * from tblC;
alter table tblA enable constraint colY_UK;
alter table tblC enable constraint colY_FK;
-- ORA-02298: cannot validate (HR.COLY_FK) - parent keys not found
```

Generate a SQL statements to disable all the Foreign Key on a specified table

```
select
  'alter table '||a.owner||'.'||a.table_name||
  ' disable constraint '||a.constraint_name||';' as STMT
from all_constraints a, all_constraints b
where a.constraint_type = 'R'
  and a.r_constraint_name = b.constraint_name
  and a.r_owner = b.owner
  and b.table_name = 'TBLA';
```

References

1. https://docs.oracle.com/cd/B12037_01/server.101/b10759/statements_6013.htm
2. https://docs.oracle.com/cd/B28359_01/server.111/b28318/datatype.htm

SELECT Statement

Listing the capabilities of SQL SELECT statements

A SELECT statement retrieves data from database. With a SELECT statement, you can use the following capabilities:

- Projection: Select the columns in a table that are returned by a query.
- Selection: Select the rows in a table that are returned by a query using certain criteria to restrict the result
- Joining: Bring together data that is stored in different tables by specifying the link between them

Executing a basic SELECT statement

```
SELECT *|{[DISTINCT] column|expression [[AS] alias],...}
FROM    table;
```

1. SQL statements are not case-sensitive.
2. SQL statements can be entered on one or more lines.
3. Keywords like SELECT, FROM cannot be abbreviated or split across lines.
4. In SQL Developer, SQL statements can optionally be terminated by a semicolon (;). Semicolons are required when you execute multiple SQL statements.
5. In SQL*Plus, you are required to end each SQL statement with a semicolon (;).

- Select All Columns

```
SELECT *
FROM   hr.employees;
```

- Select Specific Columns

```
SELECT employee_id, last_name, hire_date
FROM   hr.employees;
```

- Exclude duplicate rows

```
SELECT DISTINCT last_name
FROM   hr.employees;
```

- Use Arithmetic Operators
  - The operator precedence is the same as normal mathematics, (ie. / * + -)
  - Arithmetic expressions containing a null value evaluate to null

```
SELECT last_name, salary, (salary+100-20)*105/100
FROM   hr.employees;
```

- Use Column Heading Defaults
  - SQL Developer:
    - Default heading display: Uppercase
    - Default heading alignment: Left-aligned
  - SQL*Plus:
    - Default heading display: Uppercase
    - Character and Date column headings: Left-aligned
    - Number column headings: Right-aligned

- Use Column Alias
  - Renames a column heading
  - AS keyword between the column name and alias is optional
  - Requires double quotation marks if it contains spaces, special characters, or case-sensitive

```
SELECT last_name AS name, commission_pct comm, salary*12 "Annual Salary"
FROM   hr.employees;
```

- Literal Character Strings
  - Date and character literal values must be enclosed within single quotation marks
  - Each character string is output once for each row returned

```
SELECT last_name || ' annually earns ' || salary*12
FROM   hr.employees;
```

- Escape the single quote character use two single quotes

```
SELECT last_name || '''s employee no is ' || employee_id
FROM   hr.employees;
```

- Escape the single quote character use alternative quote (q) operator

```
SELECT last_name || q'<'s employee no is >' || employee_id
FROM   hr.employees;
```

- Learn the DESCRIBE command to display the table structure

```
DESC[RIBE] table
```

Describing various types of conversion functions that are available in SQL

Implicit data type conversion

Implicit conversion occurs when Oracle attempts to convert the values, that do not match the defined parameters of functions, into the required data types.

Explicit data type conversion Explicit conversion occurs when a function like TO_CHAR is invoked to change the data type of a value.

Using the TO_CHAR, TO_NUMBER, and TO_DATE conversion functions

- Nest multiple functions
- Apply the NVL, NULLIF, and COALESCE functions to data

Applying conditional expressions in a SELECT statement

- Use conditional IF THEN ELSE logic in a SELECT statement

Describing various types of functions available in SQL

- Describe the differences between single row and multiple row functions

Single row functions return one result per row.

Single row functions:

```
Manipulate data items
Accept arguments and return one value
Act on each row that is returned
Return one result per row
May modify the data type
Can be nested
Accept arguments that can be a column or an expression
```

Character functions

```
Case manipulation functions
  LOWER
  UPPER
  INITCAP
```

Using character, number, and date functions in SELECT statements

- Manipulate strings with character function in the SELECT and WHERE clauses
- Manipulate numbers with the ROUND, TRUNC and MOD functions

- Perform arithmetic with date data
- Manipulate dates with the date functions

Identifying the available group functions

Describing the use of group functions

Grouping data by using the GROUP BY clause

Including or excluding grouped rows by using the HAVING clause

Writing SELECT statements to access data from more than one table using equijoins and nonequijoins

Joining a table to itself by using a self-join

Viewing data that generally does not meet a join condition by using outer joins

Generating a Cartesian product of all rows from two or more tables

Restricting and Sorting Data

```
SELECT *|{[DISTINCT] column|expr [[AS] alias],...}
  FROM table
[WHERE condition(s)]
[ORDER BY {column, alias, expr, numeric_position} [ASC|DESC] [NULLS FIRST|NULLS LAST] ];
```

Limiting the rows that are retrieved by a query

- Write queries that contain a WHERE clause to limit the output retrieved
  - Character strings and date values are enclosed with single quote
  - Character values are case-sensitive and date values are format-sensitive
  - The default date display format is DD-MON-YY
  - An alias cannot be used in the WHERE clause

```
SELECT last_name, department_id, hire_date
FROM   hr.employees
WHERE department_id = 90;

SELECT last_name, department_id, hire_date
FROM   hr.employees
WHERE last_name = 'King';

SELECT last_name, department_id, hire_date
FROM   hr.employees
WHERE hire_date = '30-JAN-96';
```

- List the comparison operators and logical operators that are used in a WHERE clause

Operator	Meaning
=	Equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
<>	Not equal to (can also use != or ^=)
BETWEEN ... AND ...	Between two values (inclusive)
IN (set)	Match any value in a list
LIKE	Match a character pattern '%' - zero or many; '_' - one character
IS NULL	is a null value
AND	returns TRUE if both conditions are true
OR	returns TRUE if either condition is true
NOT	returns TRUE if the condition is false

```
-- must specify the lower limit first
SELECT last_name, salary
FROM   hr.employees
WHERE salary BETWEEN 4000 AND 5000;

-- can also use on character value
SELECT last_name, salary
FROM   hr.employees
WHERE last_name BETWEEN 'Abel' AND 'Bull'
ORDER BY last_name;

SELECT last_name, salary
FROM   hr.employees
WHERE salary in (4000,6000,8000);

-- last name start with 'A' and 2 characters at least
SELECT last_name, salary
FROM   hr.employees
WHERE last_name like 'A_%';

-- hire date at year 1999
SELECT last_name, salary, hire_date
FROM   hr.employees
WHERE hire_date like '%99';

-- employee doesn't report to any manager
SELECT last_name, salary
FROM   hr.employees
WHERE manager_id is null;

-- use AND, OR, NOT operators
SELECT last_name, job_id, salary
FROM   hr.employees
WHERE (job_id like 'AD%' OR job_id like 'IT%')
AND salary > 5000
AND NOT last_name = 'King';

-- use ESCAPE identifier
SELECT last_name, job_id
FROM   hr.employees
WHERE job_id like 'A_\_P%' ESCAPE '\';
```

- Describe the rules of precedence for comparison and logical operators

Precedence	Operators	Description
1	parenthesis	Expression within parenthesis always evaluate first
2	/, *	Division and Multiplication
3	+, -	Addition and subtraction
4	||	Concatenation
5	=, <, >, <=, >=	Equality and inequality comparison
6	[NOT] LIKE, IS [NOT] NULL, [NOT] IN	Pattern, null, and set comparison
7	[NOT] BETWEEN	Range comparison
8	<>, !=, ^=	Not equal to
9	NOT	NOT logical condition
10	AND	AND logical condition
11	OR	OR logical condition

Sorting the rows that are retrieved by a query

- Write queries that contain an ORDER BY clause sort the output of a SELECT statement

```
* The default sort order is ascending
* Null values are displayed last for ascending sequences and first for descending sequence
* You can also sort by a column that is not in the SELECT list
```

```
   SELECT employee_id, last_name, salary*12 annsal
   FROM   hr.employees
   ORDER BY annsal DESC ;
```

- Sort output in descending and ascending order

```
   SELECT   last_name, job_id, salary, commission_pct, salary*commission_pct "Comm"
   FROM     hr.employees
   ORDER BY commission_pct NULLS FIRST, 2 DESC, salary, "Comm";
```

Using ampersand substitution to restrict and sort output at runtime

Use substitution variables to:

- Temporarily store values with single-ampersand (&) and double-ampersand (&&) substitution

Use substitution variables to supplement the following:

- WHERE conditions
- ORDER BY clauses
- Column expressions
- Table names
- Entire SELECT statements

```
   --any &column_name after the &&column_name will not prompt for value again
   SELECT   employee_id, last_name, job_id, &&column_name
   FROM     hr.employees
   ORDER BY &column_name ;
```

Controlling

Start and stop iSQL*Plus

Start and stop Enterprise Manager (EM) Database Control

Start and stop the Oracle Listener

Start up and shut down Oracle Database 10g

Startup and shutdown options for Oracle Database

Handling parameter files

Locating and viewing the Database alert log

Storage Structures

The purpose of tablespaces and datafiles

Creating tablespaces

Managing tablespaces (alter, drop, generate DDL, take offline, put on line, add data files, make read-only/read-write)

Obtaining tablespace information from EM and the data dictionary views

Dropping tablespaces

The default tablespaces

Administering Users

Within Oracle, users may be managed through the webpage under the administration section, and within the Database Users subsection.

Creating and managing database user accounts

Adding users

New users can be created by an administrator or a user who has a "CREATE USER" privilege using Oracle Enterprise Manager GUI by clicking on the "Create" link in the "Users" section . Enter the username and password for the new user. You can also set the password to expire (where the user must change the password on the next login), and whether an account is locked (to prevent a user from connecting to the system).

In most cases, users should belong to the Connect role. If the user needs to create tables or have more advanced capability, the user should be placed within the Resource role. Database admins should appear in the DBA role. If desired, users may be given individual permissions within Directly Grant Privileges section.

Alternatively, users can also be created using the "CREATE USER" SQL command.

Eg: CREATE USER test IDENTIFIED BY 'naveen123' DEFAULT TABLESPACE USER_DATA PASSWORD EXPIRE

QUOTA UNLIMITED ON USER_DATA;

After creating the user CONNECT role needs to be granted for the user to connect to the database.

Eg: GRANT CONNECT TO test;

Removing users

When a user no longer needs to be present within the database, you click the drop button within the web interface.

Creating and managing roles

Granting and revoking privileges

Controlling resource usage by users

Managing Schema Objects

Creating and modifying tables

Defining constraints

Viewing the attributes of a table

Viewing the contents of a table

Creating indexes and views

Naming database objects

Selecting appropriate datatypes

Creating and using sequences

Adding constraints

Creating indexes

Creating indexes using the CREATE TABLE statement

Creating function-based indexes

Dropping columns and set column UNUSED

Performing FLASHBACK operations

Creating and using external tables

Managing Data

Manipulating data through SQL using INSERT, UPDATE, and DELETE

Using Data Pump to export data

Using Data Pump to import data

SQL Dump should either Exports in CSV or Plain SQL Format.

Loading data with SQL*Loader

Creating directory objects

Views

Creating simple and complex views

Retrieving data from views

Creating, maintaining, and using sequences

Creating and maintaining indexes

Creating private and public synonyms

Dictionary views

Explaining the data dictionary

Finding table information

Reporting on column information

Viewing constraint information

Finding view information

Verifying sequence information

Understanding synonyms

Adding comments

SQL

Retrieving Data Using the SQL SELECT Statement

List the capabilities of SQL SELECT statements

Selection, projection, join

Execute a basic SELECT statement

- Select All Columns:

```
Select * from table_name;
```

- Select Specific Columns:

```
Select column1, column2 from tables_name;
```

- Use Column Heading Defaults
- Use Arithmetic Operators:

```
Select 12 salary+100 from emp --sell value is 2.
Result: 12 * cell's value + 100   --i.e. 12 * 2 + 100= 124
```

- Understand Operator Precedence
- Learn the DESCRIBE command to display the table structure

```
Type- DESCRIBE table_name;
*NOTE: Your Oracle user and/or schema must have permissions/privaliages or be within the schema to describe the table.
 You can use the data_dictionary views to get the table info.
```

Restricting and Sorting Data

Limit the rows that are retrieved by a query

1. Write queries that contain a WHERE clause to limit the output retrieved
2. List the comparison operators and logical operators that are used in a WHERE clause
3. Describe the rules of precedence for comparison and logical operators
4. Use character string literals in the WHERE clause

Sort the rows that are retrieved by a query

1. Write queries that contain an ORDER BY clause sort the output of a SELECT statement
2. Sort output in descending and ascending order

Use ampersand substitution to restrict and sort output at runtime

the ampersand operator is used to take the input at runtime(ex:-&employeename) and if ampersand is used twice i.e && then it will take the input of single ampersand operator and is used to provide data to the query at runtime.

Using Single-Row Functions to Customize Output

Describe various types of functions available in SQL

```
  * Describe the differences between single row and multiple row functions
```

Use character, number, and date functions in SELECT statements

```
* Manipulate strings with character function in the SELECT and WHERE clauses
* Manipulate numbers with the ROUND, TRUNC and MOD functions
* Perform arithmetic with date data
* Manipulate dates with the date functions
```

Using Conversion Functions and Conditional Expressions

Describe various types of conversion functions that are available in SQL

Implicit data type conversion

Implicit conversion occurs when Oracle attempts to convert the values, that do not match the defined parameters of functions, into the required data types.

Explicit data type conversion Explicit conversion occurs when a function like TO_CHAR is invoked to change the data type of a value.

Use the TO_CHAR, TO_NUMBER, and TO_DATE conversion functions

```
* Nest multiple functions
* Apply the NVL, NULLIF, and COALESCE functions to data
```

Apply conditional expressions in a SELECT statement

```
* Use conditional IF THEN ELSE logic in a SELECT statement
```

Reporting Aggregated Data Using the Group Functions

Identify the available Group Functions

Describe the use of group functions

Group data by using the GROUP BY clause

Include or exclude grouped rows by using the HAVING clause

Displaying Data from Multiple Tables

Write SELECT statements to access data from more than one table using equijoins and nonequijoins

Join a table to itself by using a self-join

View data that generally does not meet a join condition by using outer joins

1. Join a table by using a self join

Generate a Cartesian product of all rows from two or more tables

Using Subqueries to Solve Queries

Define subqueries

Describe the types of problems that the subqueries can solve

List the types of subqueries

Write single-row and multiple-row subqueries

Using the Set Operators

Describe set operators

Use a set operator to combine multiple queries into a single query

Control the order of rows returned

Manipulating Data

Describe each data manipulation language (DML) statement

Insert rows into a table

Inserting data in database is done through "insert" command in oracle.

Syntax:

```
INSERT INTO [table name][column1,column2,.....] values(value1,value2,....);
```

Example:

```
insert into employee values(1,'Rahul','Manager');
```

By the above query the employee table gets populated by empid:-1 , empname:-'Rahul' and empdesignation:-'Manager'.

Delete rows from a table

```
DELETE client1 WHERE ID = 2;
```

Update rows in a table

To update rows in a table, write:

```
update [table name] set [column name] = [your value];
```

It will update all the rows present in the table by the given value in the selected field.

We can also add queries to this command to make a real use for example,

```
update [table name] set [column name] = [value] where [column name]>=[value];
```

You can add your query after the where clause according to your need.

Example:

```
UPDATE client1 SET address = 'the middle of nowhere' WHERE id = 1;
```

Using a set operator to combine multiple queries into a single query

Controlling the order of rows returned

Defining subqueries

Describing the types of problems that the subqueries can solve

Listing the types of subqueries

Writing single-row and multiple-row subqueries

Controlling transactions

1. Save and discard changes with the COMMIT and ROLLBACK statements
2. Explain read consistency

Using DDL Statements to Create and Manage Tables

Categorize the main database objects

Review the table structure

List the data types that are available for columns

Create a simple table

"Create table" comand is used to create table in database.

Syntax:- create table employee(empid number,empname varchar2(20),empdesignation(varchar2(20)));

The above Query will create a table named employee with which contain columns empid , empname , empdesignation followed by their datatypes/

Explain how constraints are created at the time of table creation

Concept In essence, constraints safeguard and validate the data.

Primary Key and Unique constraints both ensure the data is not duplicated. Primary Key also ensure the data is not null. Oracle will automatically generate index for Primary Key and Unique constraints. A table can only have one Primary Key, but it can have multiple unique constraints.

Foreign Key ensure the data exists in the column of the parent table it refer to. Each parent record can have multiple child records, but each child can relate to ONLY one parent record. A column with Foreign Key may not necessary to have an index.

Foreign Key can only refer to column with Primary Key or Unique Constraint

```
create table tblA (colX number, colY char);
create table tblB (colX number);
```

```
alter table tblB add (constraint colX_FK foreign key (colX) references tblA(colX));
-- ORA-02270: no matching unique or primary key for this column-list
```

```
alter table tblA add (constraint colX_PK primary key (colX));
alter table tblB add (constraint colX_FK foreign key (colX) references tblA(colX));
-- alter table success.
```

A table can ONLY have one Primary Key, but it can have multiple UNIQUE key. if the child table(s) require to referencing column other than primary key, the column on the parent table must have UNIQUE constraint.

```
alter table tblA add (constraint colY_PK primary key (colY));
-- ORA-02260: table can have only one primary key
```

Cannot create PK or Unique on a column contains duplicate data

```
insert into tblA values(1,'A');
insert into tblA values(2,'A');
alter table tblA add (constraint colY_UK unique (colY));
-- ORA-02299: cannot validate (HR.COLY_UK) - duplicate keys found
```

```
delete from tblA where colx = 2;
alter table tblA add (constraint colY_UK unique (colY));
-- alter table success.
```

```
create table tblC (colY char);
alter table tblC add (constraint colY_FK foreign key (colY) references tblA(colY));
-- alter table success.
```

Insert data into a column with FK, the value must already exist in the column that the FK reference to.

```
insert into tblC values ('B');
-- ORA-02291: integrity constraint (HR.COLY_FK) violated - parent key not found
```

```
insert into tblC values ('A');
-- 1 rows inserted
```

As long as a foreign key exist, the parent table can truncate/delete the data or disable the PK or Unique constraint

```
truncate table tblA;
-- ORA-02266: unique/primary keys in table referenced by enabled foreign keys
```

Find out the constraint information in Oracle

```
desc all_constraints;
```

```
select
  a.owner, a.table_name, a.constraint_name,
  a.constraint_type, a.status, a.r_owner, a.r_constraint_name,
  b.table_name as r_table_name, b.status as r_status
from all_constraints a
  left join all_constraints b on a.owner = b.owner and a.r_constraint_name = b.constraint_name
where a.table_name like 'TBL%';
```

```
select *
from all_cons_columns
where table_name like 'TBL%';
```

Disable constraint that have foreign key refer to is not allowed, in order to do this, you have to disable the foreign key first.

```
alter table tblA disable constraint colX_PK;
-- ORA-02297: cannot disable constraint (HR.COLX_PK) - dependencies exist
alter table tblA disable constraint colY_UK;
-- ORA-02297: cannot disable constraint (HR.COLY_UK) - dependencies exist
```

```
alter table tblC disable constraint colY_FK;
alter table tblB disable constraint colX_FK;
```

```
alter table tblA disable constraint colX_PK;
```

```
alter table tblA disable constraint colY_UK;
truncate table tblA;
```

If the data in parent table is deleted, re-enable the foreign key that contain data reference to the missing data is not allowed.

```
select * from tblC;
alter table tblA enable constraint colY_UK;
alter table tblC enable constraint colY_FK;
-- ORA-02298: cannot validate (HR.COLY_FK) - parent keys not found
```

Generate a sql statements to disable all the Foreign Key on a specified table

```
select
  'alter table '||a.owner||'.'||a.table_name||
  ' disable constraint '||a.constraint_name||';' as STMT
from all_constraints a, all_constraints b
where a.constraint_type = 'R'
  and a.r_constraint_name = b.constraint_name
  and a.r_owner = b.owner
  and b.table_name = 'TBLA';
```

Describe how schema objects work

Creating Other Schema Objects

Create simple and complex views

Retrieve data from views

Create, maintain, and use sequences

Create and maintain indexes

Create private and public synonyms

Controlling User Access

Differentiate system privileges from object privileges

Grant privileges on tables

View privileges in the data dictionary

Grant roles

Distinguish between privileges and roles

Managing Schema Objects

Add constraints

Create indexes

Create indexes using the CREATE TABLE statement

Create function-based indexes

Drop columns and set column UNUSED

Perform FLASHBACK operations

Create and use external tables

Managing Objects with Data Dictionary Views

Explain the data dictionary

Find table information

Report on column information

View constraint information

Find view information

Verify sequence information

Understand synonyms

Add comments

Manipulating Large Data Sets

Manipulate data using sub-queries

Describe the features of multi-table inserts

Use the different types of multi-table inserts

Merge rows in a table

Track the changes to data over a period of time

Managing Data in Different Time Zones

Use data types similar to DATE that store fractional seconds and track time zones

Use data types that store the difference between two date-time values

Practice using the multiple data-time functions for globalize applications

Retrieving Data Using Sub-queries

Write a multiple-column sub-query

Use scalar sub-queries in SQL

SELECT * FROM TAB

Solve problems with correlated sub-queries

Update and delete rows using correlated sub-queries

Use the EXISTS and NOT EXISTS operators

Use the WITH clause

Write a multiple-column sub-query

Use scalar sub-queries in SQL

Solve problems with correlated sub-queries

Update and delete rows using correlated sub-queries

Use the EXISTS and NOT EXISTS operators

Use the WITH clause

Hierarchical Query

Hierarchical Query allows you the transverse through a self-reference table and display the Hierarchical structure. eg. the employee table contain the manager id the employee.

list out the whole hierarchical structure of the employees

```
SELECT LPAD(' ', 4*(level-1))||last_name "Last Name", salary, department_id
FROM hr.employees
CONNECT BY PRIOR employee_id = manager_id
      START WITH manager_id is null
ORDER SIBLINGS BY last_name;
```

list out all the employees under manager 'Kochhar'

```
SELECT LPAD(' ', 4*(level-1))||last_name "Last Name",
       salary,
       department_id,
       CONNECT_BY_ISLEAF
FROM hr.employees
  CONNECT BY PRIOR employee_id = manager_id
        START WITH last_name = 'Kochhar'
 ORDER SIBLINGS BY last_name;
```

list out all the manager that 'Lorentz' report to

```
SELECT LPAD(' ', 4*(level-1))||last_name "Last Name", salary, department_id,
       SYS_CONNECT_BY_PATH(last_name, '/') "Path", CONNECT_BY_ISLEAF
FROM hr.employees
        CONNECT BY employee_id = PRIOR manager_id
        START WITH last_name = 'Lorentz'
 ORDER SIBLINGS BY last_name;
```

- pseudocolumn LEVEL -> root = 1, next level=2,3,4,5...etc
- SYS_CONNECT_BY_PATH(col, '/') shows the full path, 2nd parameter is seperator (9i)
- CONNECT_BY_ROOT(col) return the value of the root node in the current hierarchy (10g)
- pseudocolumn CONNECT_BY_ISLEAF return 1 if the return value is at the last node on the Hierarchy (ie. leaf) (10g)
- order SIBLINGS by re-order the sequence of the output and preserve the hierarchical relationship (10g)

- connect by NOCYCLE prior child = parent
  - NOCYCLE means stop tranverse the hierarchy at the level when the child reference back to the root. (10g)
  - pseudocolumn CONNECT_BY_ISCYCLE evaluate to "1" if the current row references a parent. (10g)

Regular Expression Support

List the benefits of using regular expressions

Use regular expressions to search for, match, and replace strings

Regular Expression

Class	Expression	Description
Anchoring Character	^	Start of a line
	-$	End of a line
Quantifier Character	*	Match 0 or more times
	+	Match 1 or more times
	?	Match 0 or 1 time
	{m}	Match exactly m times
	{m,}	Match at least m times
	{m, n}	Match at least m times but no more than n times
	\n	Cause the previous expression to be repeated n times
Alternative and Grouping		Separates alternates, often used with grouping operator ()
	( )	Groups subexpression into a unit for alternations, for quantifiers, or for backreferencing (see "Backreferences" section)
	[char]	Indicates a character list; most metacharacters inside a character list are understood as literals, with the exception of character classes, and the ^ and - metacharacters
Posix Character	[:alnum:]	Alphanumeric characters
	[:alpha:]	Alphabetic characters
	[:blank:]	Blank Space Characters
	[:cntrl:]	Control characters (nonprinting)
	[:digit:]	Numeric digits
	[:graph:]	Any [:punct:], [:upper:], [:lower:], and [:digit:] chars
	[:lower:]	Lowercase alphabetic characters
	[:print:]	Printable characters
	[:punct:]	Punctuation characters
	[:space:]	Space characters (nonprinting), such as carriage return, newline, vertical tab, and form feed
	[:upper:]	Uppercase alphabetic characters
	[:xdigit:]	Hexidecimal characters
Equivalence class	= =	An equivalence classes embedded in brackets that matches a base letter and all of its accented versions. eg, equivalence class '[=a=]' matches ä and â.
Match Option	c	Case sensitive matching
	i	Case insensitive matching
	m	Treat source string as multi-line activating Anchor chars
	n	Allow the period (.) to match any newline character

PL/SQL

PL/SQL

Introduction

PL/SQL stands for Procedural Language extension of SQL. It is a combination of SQL along with the procedural features of programming languages and it enhances the capabilities of SQL by injecting the procedural functionality, like conditional or looping statements, into the set-oriented SQL structure.

Advantages of PL/SQL

- Procedural Language Capability: PL/SQL consists of procedural language constructs such as conditional statements (if else statements) and loops like (FOR loops).
- Block Structures: PL/SQL consists of blocks of code, which can be nested within each other. Each block forms a unit of a task or a logical module. PL/SQL Blocks can be stored in the database and reused.
- Better Performance: PL/SQL engine processes multiple SQL statements simultaneously as a single block, thereby reducing network traffic.
- Exception Handling: PL/SQL handles exceptions (or errors) effectively during the execution of a PL/SQL program. Once an exception is caught, specific actions can be taken depending upon the type of the exception or it can be displayed to the user with a message.

Limitation

PL/SQL can only use SELECT, DML(INSERT, UPDATE, DELETE) and TC(COMMIT, ROLLBACK, SAVEPOINT) statements, DDL (CREATE, ALTER, DROP) and DCL(GRANT, REVOKE) cannot be used directly. Any DDL/DCL however, can be executed from PL/SQL when embedded in an EXECUTE IMMEDIATE statement.

Basic Structure

Each PL/SQL program consists of SQL and PL/SQL statements which form a PL/SQL block. A PL/SQL Block consists of three sections:

Declaration Section: This section is optional and it starts with the reserved keyword DECLARE. This section is used to declare any placeholders like variables, constants, records and cursors, which are used to manipulate data in the execution section.

Execution Section: This section is mandatory and it starts with the reserved keyword BEGIN and ends with END. This section is where the program logic is written to perform any task. The programmatic constructs like loops, conditional statement and SQL statements form the part of execution section.

Exception Section: The section is optional and it starts with the reserved keyword EXCEPTION. Any exception in the program can be handled in this section, so that the PL/SQL Blocks terminates gracefully. If the PL/SQL Block contains exceptions that cannot be handled, the block terminates abruptly with errors.

Every statement in the above three sections must end with a ; (semicolon). PL/SQL blocks can be nested within other PL/SQL blocks. Comments can be used to document code.

This is how PL/SQL looks.

```
/* multi-lines comments */
-- single line comments
DECLARE
    Variable declaration
BEGIN
    Program Execution
EXCEPTION
    Exception handling
END;
```

PL/SQL Placeholders

Placeholders are temporary storage area. Placeholders can be any of Variables, Constants and Records. Oracle defines placeholders to store data temporarily, which are used to manipulate data during the execution of a PL SQL block.

Depending on the kind of data you want to store, you can define placeholders with a name and a datatype. Few of the datatypes used to define placeholders are as given below.

```
Number(n,m), Char(n), Varchar2(n), Date, Long, Long Raw, Raw, Blob, Clob, Nclob, Bfile
```

The placeholders, that store the values, can change through the PL/SQL Block.

PL/SQL Variables

The General Syntax to declare a variable is:

```
variable_name datatype [NOT NULL := value ];
```

- variable_name is the name of the variable.
- datatype is a valid PL/SQL datatype.
- NOT NULL is an optional specification on the variable. If NOT NULL is specified, you must provide the initial value.
- value or DEFAULT value is also an optional specification, where you can initialize a variable.
- Each variable declaration is a separate statement and must be terminated by a semicolon.

The below example declares two variables, one of which is a not null.

```
DECLARE
  emp_id varchar2(10);
  salary number(9,2) NOT NULL := 1000.00;
```

The value of a variable can change in the execution or exception section of the PL/SQL Block. We can assign values to variables in the two ways given below.

1) Directly assign value to variable.

```
variable_name:=  value;
```

2) Assign values to variables directly from the database columns.

```
SELECT column_name
INTO variable_name
FROM table_name
[WHERE condition];
```

The example below will get the salary of an employee with id '12345' and display it on the screen.

```
DECLARE
  var_emp_id varchar2(10) = 'A12345';
  var_salary number(9,2);
BEGIN
  SELECT salary
  INTO var_salary
  FROM employee
  WHERE emp_id = var_emp_id;
  dbms_output.put_line(var_salary);
  dbms_output.put_line('Employee ' || var_emp_id || ' earns salary ' || var_salary);
END;
/
```

NOTE: The slash '/' indicates to execute the above PL/SQL Block.

PL/SQL Records

Records are composite datatypes, which contains a combination of different scalar datatypes like char, varchar, number etc. Each scalar data types in the record holds a value. A record can store values of a row in a table.

```
TYPE record_name IS RECORD
  (col_name_1 datatype,
   col_name_2 table_name.column_name%type);
```

A datatype can be declared in the same way as you create a table, like col_name_1. If a field is based on a column from database table, you can define the datatype as col_name_2. You can also use %type method to declare datatype of variable and constant. Similar to %type, if all the fields of a record are based on the columns of a table, it can be declared by using %rowtype method.

```
record_name table_name%ROWTYPE;
```

For Example:

```
DECLARE
  TYPE rec_employee IS RECORD
   (emp_id          varchar2(10),
    emp_last_name   employees.last_name%type,
    emp_dept        employees.dept%type,
    salary          number(9,2)
   );
```

```
DECLARE
  rec_employee employees%ROWTYPE;
```

Declaring the record as a ROWTYPE Advantages: 1) Do not need to explicitly declare variables for all the columns in a table. 2) If the column specification in the database table is altered, the code does not need to update.

Disadvantage: 1) When a record is created as a ROWTYPE, fields will be created for all the columns in the table and memory will be used to create the datatype for all the fields.

Assign values to record Similar to variable, you can assign value to record either by direct assign or through the SELECT statements

```
record_name.col_name := value;
```

```
SELECT col_1, col_2
INTO record_name.col_name_1, record_name.col_name_2
FROM table_name
[WHERE condition];
```

If the records is declared as ROWTYPE, SELECT * can be used to assign values.

```
SELECT * INTO record_name
FROM table_name
[WHERE condition];
```

The column value of the record can be retrieved as below syntax

```
var_name := record_name.col_name;
```

Scope of Variables and Records

PL/SQL allows the nesting of Blocks within Blocks i.e, the Execution section of an outer block can contain inner blocks. Variables which are accessible to an outer Block are also accessible to all nested inner blocks; however, the variables declared in the inner blocks are not accessible to the outer blocks.

Based on their declaration we can classify variables into two types.

- Local variables - These are declared in a inner block and cannot be referenced by outer blocks.
- Global variables - These are declared in an outer block and can be referenced by its itself and by its inner blocks.

In the below example, two variables are created in the outer block and assigning their product to the third variable created in the inner block. The variables 'var_num1' and 'var_num2' can be accessed anywhere in the block; however, the variable 'var_result' is declared in the inner block, so it cannot be accessed in the outer block.

```
DECLARE
  var_num1 number;
  var_num2 number;
BEGIN
  var_num1 := 100;
  var_num2 := 200;
  DECLARE
    var_result number;
```

```
  BEGIN
    var_result := var_num1 * var_num2;
  END;
  /* var_result is not accessible to here */
END;
/
```

PL/SQL Constants

As the name implies a constant is a value used in a PL/SQL Block that remains unchanged throughout the program. A constant is a user-defined literal value. You can declare a constant and use it instead of actual value.

```
constant_name CONSTANT datatype := VALUE;
```

For example:

```
DECLARE
  comm_pct CONSTANT number(3) := 10;
```

You must assign a value to the constant while declaring it. If you assign a value to the constant later, Oracle will prompt exception.

PL/SQL Conditional Statements

PL/SQL supports programming language features like conditional statements, iterative statements.

The syntax of conditional statements:

```
IF condition_1 THEN
 statement_1;
 statement_2;
[ELSIF condtion_2 THEN
 statement_3;]
[ELSE
 statement_4;]
END IF;
```

Note: be aware of the keyword ELSIF, there is no 'E' before 'IF'.

PL/SQL Iterative Statements

An iterative statements are used when you want to repeat the execution of one or more statements for specified number of times. There are three types of loops in PL/SQL:

1. Simple Loop A Simple Loop is used when a set of statements is to be executed at least once before the loop terminates. An EXIT condition must be specified in the loop, otherwise the loop will get into an infinite number of iterations. When the EXIT condition is satisfied the process exits from the loop.

```
LOOP
  statements;
  EXIT;
  {or EXIT WHEN condition;}
END LOOP;
```

Note: a) Initialize a variable before the loop body. b) Increment the variable in the loop. c) Use a EXIT WHEN statement to exit from the Loop. If you use a EXIT statement without WHEN condition, the statements in the loop is executed only once.

2. While Loop A WHILE LOOP is used when a set of statements has to be executed as long as a condition is true. The condition is evaluated at the beginning of each iteration. The iteration continues until the condition becomes false.

```
WHILE <condition>
 LOOP statements;
END LOOP;
```

Note: a) Initialize a variable before the loop body. b) Increment the variable in the loop. c) EXIT WHEN statement and EXIT

statements can be used in while loops it is seldom used.

3. FOR Loop A FOR LOOP is used to execute a set of statements for a pre-determined number of times. Iteration occurs between the start and end integer values given. The counter is always incremented by 1. The loop exits when the counter reaches the value of the end integer.

```
FOR counter IN start_val..end_val
  LOOP statements;
END LOOP;
```

Note: a) The counter variable is implicitly declared in the declaration section, so it's not necessary to declare it explicitly. b) The counter variable is incremented by 1 and does not need to be incremented explicitly. c) EXIT WHEN statement and EXIT statements can be used in FOR loops but it is seldom used.

PL/SQL Cursors

A cursor contains information on a select statement and the rows of data accessed by it. This temporary work area is used to store the data retrieved from the database, and manipulate this data. A cursor can hold more than one row, but can process only one row at a time. The set of rows the cursor holds is called the active set.

There are two types of cursors in PL/SQL:

Implicit cursor:

When you execute DML statements like DELETE, INSERT, UPDATE and SELECT..INTO statements, implicit statements are created to process these statements.

Oracle provides few attributes called as implicit cursor attributes to check the status of DML operations. The cursor attributes available are %FOUND, %NOTFOUND, %ROWCOUNT, and %ISOPEN.

For example, When you execute INSERT, UPDATE, or DELETE statements the cursor attributes tell us whether any rows are affected and how many have been affected. When a SELECT... INTO statement is executed in a PL/SQL Block, implicit cursor attributes can be used to find out whether any row has been returned by the SELECT statement. PL/SQL returns an error when no data is selected.

The status of the cursor for each of these attributes are defined in the below table.

Attribute	Return Value
SQL%FOUND	The return value is TRUE, if the DML statements like INSERT, DELETE and UPDATE affect at least one row and if SELECT ….INTO statement return at least one row. The return value is FALSE, if DML statements like INSERT, DELETE and UPDATE do not affect row and if SELECT….INTO statement do not return a row.
SQL%NOTFOUND	The return value is FALSE, if DML statements like INSERT, DELETE and UPDATE at least one row and if SELECT ….INTO statement return at least one row. The return value is TRUE, if a DML statement like INSERT, DELETE and UPDATE do not affect even one row and if SELECT ….INTO statement does not return a row.
SQL%ROWCOUNT	Return the number of rows affected by the DML operations INSERT, DELETE, UPDATE, SELECT
SQL%ISOPEN	Always return FALSE

uses implicit cursor attributes:

```
DECLARE  var_rows number(5);
BEGIN
  UPDATE employee
  SET salary = salary + 2000;
  IF SQL%NOTFOUND THEN
    dbms_output.put_line('None of the salaries where updated');
  ELSIF SQL%FOUND THEN
    var_rows := SQL%ROWCOUNT;
```

```
    dbms_output.put_line('Salaries for ' || var_rows || 'employees are updated');
  END IF;
END;
```

In the above PL/SQL Block, the salaries of all the employees in the 'employee' table are updated. If none of the employee's salary are updated we get a message 'None of the salaries where updated'. Else we get a message like for example, 'Salaries for 100 employees are updated' if there are 100 rows in 'employee' table.

Explicit cursor:

An explicit cursor is defined in the declaration section of the PL/SQL Block. It must be created when you are executing a SELECT statement that returns more than one row. Even though the cursor stores multiple records, only one record can be processed at a time, which is called as current row. When you fetch a row the current row position moves to next row.

There are four steps in using an Explicit Cursor.

- **DECLARE** the cursor in the declaration section.
- **OPEN** the cursor in the Execution Section.
- **FETCH** the data from cursor into PL/SQL variables or records in the Execution Section.
- **CLOSE** the cursor in the Execution Section before you end the PL/SQL Block.

Declaration

```
CURSOR cursor_name IS select_statement;
```

For example:

```
DECLARE
   CURSOR cur_emp IS
   SELECT *
   FROM employees
   WHERE salary > 10000;
```

Using Cursor When a cursor is opened, the first row becomes the current row. When the data is fetched it is copied to the record or variables and the logical pointer moves to the next row and it becomes the current row. On every fetch statement, the pointer moves to the next row. If you want to fetch after the last row, the program will throw an error. When there is more than one row in a cursor we can use loops along with explicit cursor attributes to fetch all the records.

```
OPEN cursor_name;
FETCH cursor_name INTO record_name|variable_list;
CLOSE cursor_name;
```

Note:

- We can fetch the rows in a cursor to a PL/SQL Record or a list of variables created in the PL/SQL Block.
- If you are fetching a cursor to a PL/SQL Record, the record should have the same structure as the cursor.
- If you are fetching a cursor to a list of variables, the variables should be listed in the same order in the fetch statement as the columns are present in the cursor.
- When we try to open a cursor which is not closed in the previous operation, it throws exception.
- When we try to fetch a cursor after the last operation, it throws exception.

For Example:

```
DECLARE
  rec_emp employees%rowtype;
  CURSOR cur_emp IS
  SELECT *
  FROM employees
  WHERE salary > 10000;
BEGIN
  OPEN cur_emp;
  FETCH cur_emp INTO rec_emp;
    dbms_output.put_line (rec_emp.first_name || '  '
                        || rec_emp.last_name);
  CLOSE emp_cur;
END;
```

Oracle provides some attributes known as Explicit Cursor Attributes to control the data processing while using cursors. We use these attributes to avoid errors while accessing cursors through OPEN, FETCH and CLOSE Statements.

Attributes	Return Values
Cursor_name%FOUND	TRUE, if fetch statement returns at least one row. FALSE, if fetch statement doesn't return a row.
Cursor_name%NOTFOUND	TRUE, , if fetch statement doesn't return a row. FALSE, if fetch statement returns at least one row.
Cursor_name%ROWCOUNT	The number of rows fetched by the fetch statement. If no row is returned, the PL/SQL statement returns an error.
Cursor_name%ISOPEN	TRUE, if the cursor is already open in the program. FALSE, if the cursor is not opened in the program.

Cursor with a Simple Loop:

```
DECLARE
  CURSOR cur_emp IS
  SELECT first_name, last_name, salary FROM employees;
  rec_emp cur_emp%rowtype;
BEGIN
  IF NOT cur_emp%ISOPEN THEN
    OPEN cur_emp;
  END IF;
  LOOP
    FETCH cur_emp INTO rec_emp;
    EXIT WHEN cur_emp%NOTFOUND;
    dbms_output.put_line(cur_emp.first_name || ' ' ||cur_emp.last_name
      || ' ' ||cur_emp.salary);
  END LOOP;
END;
/
```

The cursor attribute %ISOPEN is used to check if the cursor is open, if the condition is true the program does not open the cursor again. The cursor attribute %NOTFOUND is used to check whether the fetch returned any row. If there is no row found, the program would exit. Typically, when the cursor reach the last row, no more row can be fetched.

Cursor with a While Loop:

```
DECLARE
  CURSOR cur_emp IS
  SELECT first_name, last_name, salary FROM employees;
  rec_emp cur_emp%rowtype;
BEGIN
  IF NOT cur_emp%ISOPEN THEN
    OPEN cur_emp;
  END IF;
  FETCH cur_emp INTO sales_rec;
 WHILE cur_emp%FOUND THEN
 LOOP
   dbms_output.put_line(cur_emp.first_name || ' ' ||cur_emp.last_name
   || ' ' ||cur_emp.salary);
   FETCH cur_emp INTO sales_rec;
 END LOOP;
END;
/
```

Using %FOUND to evaluate if the first fetch statement returned a row, if TRUE, the program moves into the while loop. Inside the loop, use fetch statement again to process the next row. If the fetch statement is not executed once before the while loop, the while condition will return false in the first instance and the while loop is skipped.

Cursor with a FOR Loop: When using FOR LOOP, you do not need to declare a record or variables to store the cursor values, do not need to open, fetch and close the cursor. These functions are accomplished by the FOR LOOP automatically.

```
DECLARE
 CURSOR cur_emp IS
   SELECT first_name, last_name, salary FROM employees;
 rec_emp cur_emp%rowtype;
BEGIN
  FOR rec_emp in cur_emp
  LOOP
    dbms_output.put_line(cur_emp.first_name || ' ' ||cur_emp.last_name
    || ' ' ||cur_emp.salary);
  END LOOP;
END;
/
```

When the FOR loop is processed a record 'rec_emp' of structure 'cur_emp' gets created, the cursor is opened, the rows are fetched to the record 'rec_emp' and the cursor is closed after the last row is processed. By using FOR Loop, you can reduce the number of lines in the program.

PL/SQL Exception Handling

PL/SQL Procedures

When calling store procedure from the PL/SQL Block, you simply use the store procedure name to call. If you prefix the 'EXECUTE' keyword in front of the store procedure name, you will receive an error.

```
my_sproc;
```

```
EXECUTE my_sproc;

PLS-00103: Encountered the symbol "my_sproc" when expecting one of the following:
:= . ( @ % ; immediate
The symbol ":=" was substituted for "my_sproc" to continue.
```

```
EXECUTE IMMEDIATE my_sproc;

PLS-00222: no function with name exists in this scope
```

PL/SQL Functions

Parameters-Procedure, Function

PL/SQL Triggers

Multimedia Databases

Description

Oracle Multimedia is a services suite provided with Oracle Database (excluding the Express version where it can't be added[1]) since the version 8 (in 1997), to manage the multimedia databases.

It's composed by the package ORDSYS ("ORD" for *object-relational data*) allowing the multimedia objects management into the database[2]. This package includes several classes[3]:

- ORDMultimedia: abstract superclass storage the common attributes and methods to the classes ORDAudio, ORDImage, and ORDVideo[4].
- ORDAudio: sound properties storage.
- ORDDoc: heterogeneous properties storage.
- ORDImage: images properties storage.
- ORDVideo: videos properties storage.
- ORDSource: multimedia BLOB or BFILE (accessible in HTTP) properties storage[5].
- DICOM (Digital Imaging and Communications in Medicine[6]).

Attributes

ORDAudio[7]	ORDDoc[8]	ORDImage[9]	ORDVideo[10]
description	source	source	description
source	format	height	source
format	mimeType	width	format
mimeType	contentLength	contentLength	mimeType
comments	comments	fileFormat	comments
encoding		contentFormat	width
numberOfChannels		compressionFormat	height
sampleSize		mimeType	frameResolution
compressionType			frameRate
audioDuration			videoDuration
			numberOfFrames
			compressionType
			numberOfColors
			bitRate

Utilization

```
CREATE TABLE MyImages (
    id  INTEGER PRIMARY KEY,
    image   ORDSYS.ORDImage
);
```

Oracle HTTP Server[Download (http://www.oracle.com/technetwork/middleware/webtier/downloads/index.html)] allows to execute PL/SQL requests from a navigator.

References

1. "Managing Oracle Multimedia Installations". http://www.comp.dit.ie/btierney/Oracle11gDoc/appdev.111/b28415/ap_instl_upgrd.htm.
2. "Gestion avancée d'image sous Oracle avec Java" (in French). http://fildz.developpez.com/tutoriel/oracle-java/ordimage/.
3. "Common Methods and Notes for Oracle Multimedia Object Types". https://docs.oracle.com/cd/E11882_01/appdev.112/e10776/ch_comref.htm#AIVUG3000.
4. "Common Methods and Notes for Oracle Multimedia Object Types". https://docs.oracle.com/html/A67296_01/im_mmref.htm#998184.
5. Lynne Dunckley, Larry Guros (8 avril 2011). Digital Press. ed. *Oracle 10g Developing Media Rich Applications.* https://books.google.fr/books?id=-dbeFbCswAYC&pg=PA60&dq=ORDSYS++ORDAudio+ORDDoc+ORDImage+ORDVideo+ORDSource&hl=fr&sa=X&ved=0ahUKEwj7z9jC5OPKAhVHMhoKHXsYCr8Q6AEIJjAB#v=onepage&q=ORDSYS%20%20ORDAudio%20ORDDoc%20ORDImage%20ORDVideo%20ORDSource&f=false.
6. "Medical Imaging and Communication". https://docs.oracle.com/database/121/IMDCM/ch_intro.htm#IMDCM1100.
7. "ORDAudio". https://docs.oracle.com/cd/B19306_01/appdev.102/b14297/ch_audref.htm.
8. "ORDDoc". https://docs.oracle.com/cd/B28359_01/appdev.111/b28414/ch_docref.htm.
9. "ORDImage". https://docs.oracle.com/cd/B28359_01/appdev.111/b28414/ch_imgref.htm.
10. "ORDVideo". https://docs.oracle.com/cd/B28359_01/appdev.111/b28414/ch_vidref.htm.

- "Multimedia User's Guide". http://docs.oracle.com/database/121/IMURG/toc.htm.

Spatiotemporal Databases

Spatial data

When typing the fields, some represent graphical objects, and so are considered as "Spatial" (cf. spatial database). Consequently, they are manipulated with different requests than for the text.

With Oracle, its implemented since the version 7, in an extension of the *Enterprise Edition*Download (http://www.oracle.com/technetwork/database/options/spatialandgraph/downloads/index-093371.html), provided objects with the prefix SDO for *Spatial Data Option*.

Objects

To store the spatial objects, we use the field type *SDO_GEOMETRY*, and the seven methods to manipulate it[1]:

1. Get_Dims
2. Get_GType
3. Get_LRS_Dim
4. Get_WKB
5. Get_WKT
6. ST_CoordDim
7. ST_IsValid

Then the request operators[2]:

1. SDO_FILTER: list the objects which interact with the target.
2. SDO_JOIN: spatial join.
3. SDO_NN: target nearest neighbor.
4. SDO_NN_DISTANCE: distance with the nearest neighbor.
5. SDO_RELATE: list the objects which interact in a certain manner.
6. SDO_WITHIN_DISTANCE: returns *true* if two objects are within a certain distance from one to another.

Spatiotemporal data

We use a predicate to foresee the stored objects movement[3]. However, the spatiotemporal databases need frequent updates.

Indexation

Les modes d'indexation choisis par Oracle pour les données spatiales sont l'arbre R[4], l'arbre Q, et le Z-order[5].

Link with the GIS

To represent the data on maps, we use a geographic information system (GIS). For example:

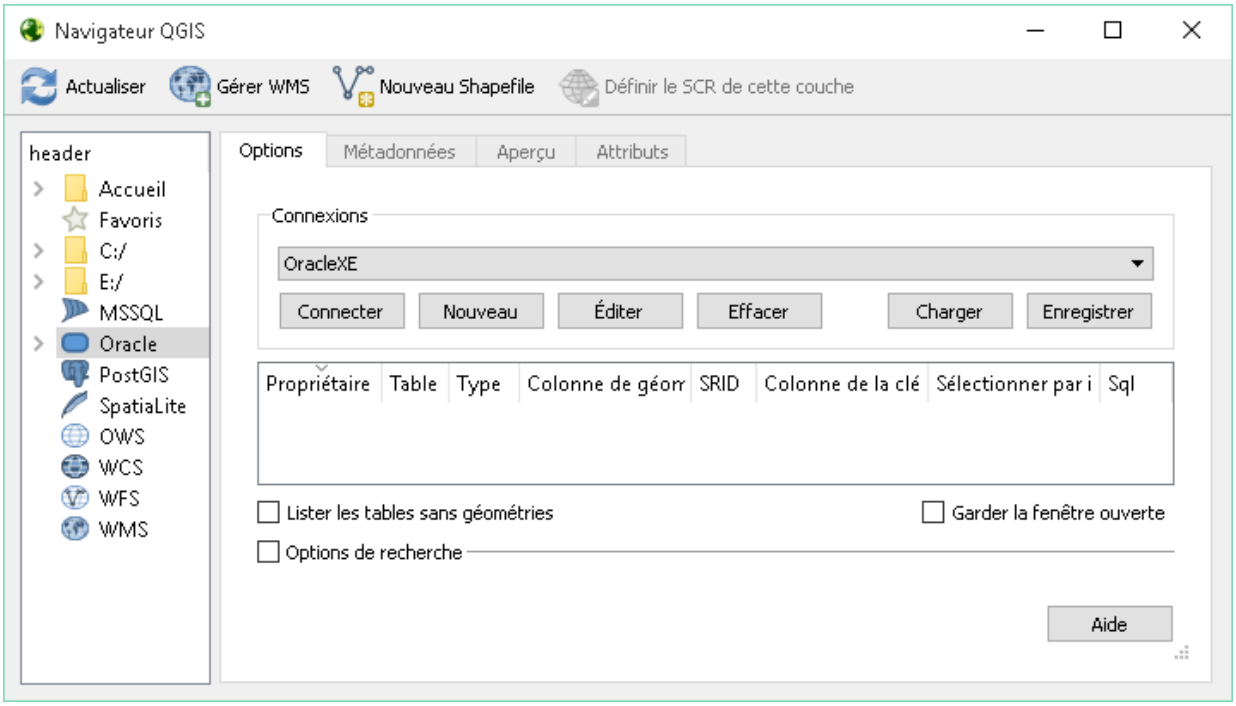
- GRASS GIS[6]
- QGIS

If the software need an ODBC data source to access to the Oracle databases:

1. Launch %windir%\system32\odbcad32.exe.
2. Add a system source. The Oracle driver can be chosen in the list if the DBMS is installed.
3. Fill the *TNS service name* with the name which can be found into C:\oraclexe\app\oracle\product\11.2.0\server\network\ADMIN\tnsnames.ora.
4. Then write the password of the connection created with SQL*Plus.

Examples

- http://download.oracle.com/otndocs/products/spatial/pdf/au_melbourne06_start.pdf



- http://download.oracle.com/otndocs/products/spatial/pdf/GeocodingInOracleUsing_HERE_MapContent.pdf

References

1. "SDO_GEOMETRY Object Type". https://docs.oracle.com/cd/B19306_01/appdev.102/b14255/sdo_objrelschema.htm#i1004087.
2. "Spatial Operators". https://docs.oracle.com/cd/B19306_01/appdev.102/b14255/sdo_operat.htm.
3. "Authorizing Access to Dynamic Spatial-Temporal Data". http://www.oracle.com/technetwork/articles/vanvelden-spatial-095837.html.
4. "Spatial Concepts". https://docs.oracle.com/html/A88805_01/sdo_intr.htm.
5. "ZOrder Method". http://docs.oracle.com/cd/E20213_01/doc/win.112/e17727/dcmethods006.htm.
6. "Geographic Resources Analysis Support System (GRASS): More Than a Mapping Tool". http://www.oracle.com/technetwork/articles/mitasova-grass-092663.html.

- "Spatial Developer's Guide". http://docs.oracle.com/cd/E11882_01/appdev.112/e11830/toc.htm.

10g Advanced SQL

This documentation details the usage of the latest query methodology on the Oracle 10g DBMS.

Joins

Join queries combine rows from two or more tables, views, or materialized views. If multiple tables are listed in the query's FROM clause the Oracle Database performs a join. Columns from any of the tables may be listed in the select list. Columns that exist in both tables, however, must be qualified, in order to avoid ambiguity.

The following query returns the mortgage information for all payments received from customers during the year 2007.

```
SELECT customer.account_no, mortgage.mortgage_id, payment.payment_id, payment.amount
FROM customer
    JOIN mortgage ON mortgage.customer_id = customer.customer_id
    JOIN payment ON payment.mortgage_id = mortgage.mortgage_id
WHERE payment.year = 2007;
```

The other way of writng the same query can be

```
SELECT customer.account_no, mortgage.mortgage_id, payment.payment_id, payment.amount
FROM customer,
     mortgage,
     payment
WHERE mortgage.customer_id = customer.customer_id
AND   payment.mortgage_id = mortgage.mortgage_id
AND   payment.year = 2007;
```

NATURAL JOIN

The NATURAL JOIN joins two tables which contain a column or multiple columns with the same name and data-type.

The following query joins the customer table to the invoice table with a natural join, the natural join utilizes the customer_id that is present on both the customer table and the invoice table. It returns the customer and invoice data for invoices that have not had any payments made on them.

```
SELECT customer_id, invoice_id, customer.first_name, customer.last_name
FROM CUSTOMER
NATURAL JOIN invoice
WHERE invoice.amount_paid = 0;
```

INNER JOIN

Most of the commonly used joins are actually INNER JOINs. The INNER JOIN joins two or more tables, returning only the rows that satisfy the JOIN condition. Here are some examples of INNER JOINs.

This joins the customer and order table, connecting the customers to their orders. The result contains a combined list of customers and their orders, if a customer does not have an order, they are omitted from the result.

```
SELECT customer_id, order_id
FROM customer c
INNER JOIN order o ON c.customer_id = o.customer_id;
```

The Other way of writing query is

```
SELECT c.customer_id, o.order_id
FROM   customer c, order o
WHERE  c.customer_id = o.customer_id;
```

OUTER JOIN

The OUTER JOIN joins two or more tables, returning all values whether or not the join condition is met. When a value exists in one table but not the other, nulls are used in the place of the columns that are joined to a record without a JOIN companion.

There are three specific types of outer joins: FULL OUTER JOIN, LEFT OUTER JOIN and RIGHT OUTER JOIN.

FULL OUTER JOIN

With the FULL OUTER JOIN the query will return rows from either of the tables joined, whether or not there is any matching data on the table joined. If no matching data exists, nulls are placed into the fields where data would have otherwise existed.

In the following example, the data in a table is synced with the data that is regularly imported into a data import table via SQL Loader. A stored procedure is then used to see if anything was added, updated or removed and the rows are merged accordingly.

```
SELECT p.name, p.status, p.description, p.qty, i.name, i.status, i.description, i.qty
FROM product p FULL OUTER JOIN import_product i
ON p.product_code = i.product_code;
```

LEFT OUTER JOIN

With the LEFT OUTER JOIN the query will return rows only if the row exists in the table specified on the left side of the join.

When no matching data is found from the table on the right side of the join, nulls are placed into the fields where the data would have otherwise existed.

The following example will return all of the customers and their associated cases if they have one. If the customer has no case then it will only return the data for the customer.

```
SELECT cust.customer_id, case.case_id, case.description
FROM customer cust LEFT OUTER JOIN casefile case
ON cust.case_id = case.case_id;
```

RIGHT OUTER JOIN

With the RIGHT OUTER JOIN the query will return rows only if the row exists in the table specified on the right side of the join. When no matching data is found from the table on the left side of the join, nulls are placed into the fields where the data would have otherwise existed.

The following example will return a list of trucks and their cargo. If a truck has no cargo then a null will be put in place of the field specifying the cargo's load_id.

```
SELECT truck.truck_id, cargo.load_id, cargo.description
FROM cargo RIGHT OUTER JOIN truck
ON truck.load_id = cargo.load_id;
```

Subqueries

Operators

UNION [ALL]

The UNION operator outputs the items that exist in both result sets. The UNION ALL operator outputs all of the items in the two sets, whether or not both sets contain the item.

The following query returns all customers from San Francisco whose balance is 100000 and 500000.

```
SELECT customer_id FROM customer WHERE city = 'SAN FRANCISCO'
UNION
SELECT customer_id FROM accounts WHERE balance BETWEEN 100000 AND 500000;
```

MINUS

The query after the MINUS operator is removed from the result set of the queries before the operator.

In the following example, the first part of the query gets all of the customers. In the second part of the inactive customers are taken out. Finally, in the third part of the query, customers with zip codes between 80000 and 90000 removed from the set.

```
SELECT customer_id FROM customer
MINUS
SELECT customer_id FROM customer WHERE status = 'I'
MINUS
SELECT customer_id FROM customer WHERE zip BETWEEN 80000 AND 99000;
```

INTERSECT

The INTERSECT operator only returns the results that are present in both of the queries.

The following example returns all of the customers who have a balance due in Los Angeles.

```
SELECT customer_id FROM customer WHERE city = 'LOS ANGELES'
INTERSECT
SELECT customer_id FROM orders WHERE balance_due > 0;
```

Case Statements

The following queries are equivalent, they return all of the customers from Switzerland. The CASE statement translates the single character status flags "A" and "I" to "ACTIVE" and "INACTIVE" If a value is NULL then it returns the string "NULL"

Basic Usage

The simplest form of a CASE statement specifies the variable and then the possible values to check for.

```
SELECT customer_id,
    CASE status
        WHEN 'A' THEN 'ACTIVE'
        WHEN 'I' THEN 'INACTIVE'
        ELSE 'NULL'
    END
FROM customer
WHERE country_name = 'SWITZERLAND';
```

Searched Case

The searched CASE expression is the more advanced form of case. Instead of specifying the value to be checked at the beginning, each WHEN statement has a comparison that is checked.

```
SELECT customer_id,
    CASE
        WHEN status = 'A' THEN 'ACTIVE'
        WHEN status = 'I' THEN 'INACTIVE'
        ELSE 'NULL'
    END
FROM customer
WHERE country_name = 'SWITZERLAND';
```

Regular Expression Support

List the benefits of using regular expressions

Use regular expressions to search for, match, and replace strings

Regular Expression

Class	Expression	Description
Anchoring Character	^	Start of a line
	$	End of a line
Quantifier Character	*	Match 0 or more times
	+	Match 1 or more times
	?	Match 0 or 1 time
	{m}	Match exactly m times
	{m,}	Match at least m times
	{m, n}	Match at least m times but no more than n times
	\n	Cause the previous expression to be repeated n times
Alternative and Grouping	|	Separates alternates, often used with grouping operator ()
	( )	Groups subexpression into a unit for alternations, for quantifiers, or for backreferencing (see "Backreferences" section)
	[char]	Indicates a character list; most metacharacters inside a character list are understood as literals, with the exception of character classes, and the ^ and - metacharacters
Posix Character	[:alnum:]	Alphanumeric characters
	[:alpha:]	Alphabetic characters
	[:blank:]	Blank Space Characters
	[:cntrl:]	Control characters (nonprinting)
	[:digit:]	Numeric digits
	[:graph:]	Any [:punct:], [:upper:], [:lower:], and [:digit:] chars
	[:lower:]	Lowercase alphabetic characters
	[:print:]	Printable characters
	[:punct:]	Punctuation characters
	[:space:]	Space characters (nonprinting), such as carriage return, newline, vertical tab, and form feed
	[:upper:]	Uppercase alphabetic characters
	[:xdigit:]	Hexidecimal characters
Equivalence class	= =	An equivalence classes embedded in brackets that matches a base letter and all of its accented versions. eg, equivalence class '[=a=]' matches ä and â.
Match Option	c	Case sensitive matching
	i	Case insensitive matching
	m	Treat source string as multi-line activating Anchor chars
	n	Allow the period (.) to match any newline character
x	ignore white space characters	

REGEXP_LIKE

REGEXP_LIKE performs complex regular expression pattern matching and supports much greater range of string patterns than LIKE. this function is introduced in 10g.

last name begin with T and the 2nd character is either 'o' or 'u'

```
SELECT last_name
FROM   hr.employees
WHERE  REGEXP_LIKE(last_name, '^T[ou]');
```

last name begin with 'T' and end with 'r'

```
select last_name
from   hr.employees
where  REGEXP_LIKE( last_name, '^T.*r$' );
```

first name is either 'Steven' or 'Stephen'

```
SELECT first_name
FROM hr.employees
WHERE REGEXP_LIKE (first_name, '^Ste(v|ph)en$');
```

last name contain double vowel characters (ie. 'aa', 'ee', 'ii', 'oo', 'uu') and the matching is non-case sensitive

```
SELECT last_name
FROM hr.employees
WHERE REGEXP_LIKE (last_name, '([aeiou])\1', 'i');
```

REGEXP_INSTR

REGEXP_INSTR performs complex regular expression pattern matching and supports much greater range of string patterns than INSTR. this function is introduced in 10g.

show the position of the 1st lowercase vowel characters

```
SELECT last_name, REGEXP_INSTR(last_name, '[aeiou]')
FROM   hr.employees;
```

REGEXP_SUBSTR

REGEXP_SUBSTR performs complex regular expression pattern matching and supports much greater range of string patterns than SUBSTR. this function is introduced in 10g.

extract the 1st character if the last name start with 'A' or 'C'

```
SELECT last_name, REGEXP_SUBSTR(last_name, '^[AC]')
FROM   hr.employees;
```

Start at 3rd position, extract 2 characters from the last name

```
SELECT last_name, REGEXP_SUBSTR(last_name, '..',3)
FROM   hr.employees;
```

REGEXP_COUNT

REGEXP_COUNT performs count against a value and it is different from the aggregate COUNT function. This function is introduced in 11g.

find the occurrences of the vowel pattern in the last name

```
SELECT last_name, REGEXP_COUNT( last_name, '[aeiou]' )
FROM   hr.employees;
```

Security

Applying the principle privilege

Managing accounts

Implementing standard password security features

Auditing database activity

Registering for security updates

Differentiating system privileges from object privileges

Granting privileges on tables

Viewing privileges in the data dictionary

Granting roles

syntax:GRANT role TO user **example**: GRANT dba TO scott

Distinguishing between privileges and roles

Net Services

Using Database Control to create additional listeners

Using Database Control to create Oracle Net service aliases

Using Database Control to configure connect time failover

Using Listener features

Using the Oracle Net Manager to configure client and middle-tier connections

Using TNSPING to test Oracle Net connectivity

Describing Oracle Net Services

Describing Oracle Net names resolution methods

Shared Servers

Identifying when to use Oracle Shared Servers

A shared server can treat several users processes. [1]

Configuring Oracle Shared Servers

Monitoring Shared Servers

Describing the Shared Server architecture

References

1. https://docs.oracle.com/cd/B28359_01/server.111/b28310/manproc001.htm

Performance Monitoring

On the oracle database Web administration tool, there is a usage monitor shown on the right-hand side of the screen. This displays both the Storage used by Oracle and the current Memory in use. It also lists the total sessions and users.

For a more detailed view of server settings, Enter the Administration section, and open the Monitor.

- **Sessions** displays a list of users currently connected to the database.
- **System Statistics** displays the current resources used by the database. It can also be set to display changes within the statistics from a given reference point.
- **Top SQL** displays a list of previous SQL statements that are the most resource intensive.
- **Long Operations** displays the status of operations that are taking more than 6 seconds.

Troubleshooting invalid and unusable objects

Gathering optimizer statistics

Viewing performance metrics

Reacting to performance issues

Proactive Maintenance

Setting warning and critical alert thresholds

Collecting and using baseline metrics

Using tuning and diagnostic advisors

Using the Automatic Database Diagnostic Monitor (ADDM)

Managing the Automatic Workload Repository

Describing server-generated alerts

Undo Management

Monitoring and administering undo

Configuring undo retention

Guaranteeing undo retention

Using the Undo Advisor

Describing the relationship between undo and transactions

Sizing the undo tablespace

Monitoring and Resolving Lock Conflicts

Detecting and resolving lock conflicts

Managing deadlocks

Describing the relationship between transactions and locks

Explaining lock modes

Backup and Recovery Concepts

Describing the basics of database backup, restore, and recovery

Describing the types of failure that can occur in an Oracle database

Describing ways to tune instance recovery

Identifying the importance of checkpoints, redo log files, and archived log files

Configuring ARCHIVELOG mode

Configuring a database for recoverability

Backups

Creating consistent database backups

Backing up your database without shutting it down

Creating incremental backups

Automating database backups

Monitoring the Flash Recovery area

Describing the difference between image copies and backup sets

Describing the different types of database backups

Backing up a control file to trace

Managing backups

Recovery

Recovering from loss of a control file

Using RMAN:

If flash recovery area is configured and control file auto backup is on then:

RMAN> connect target /

RMAN> startup nomount;

RMAN> restore controlfile from autobackup;

This will restore the control file to the location specified by the initialization parameter CONTROL_FILES mentioned in initialization parameter.

If flash recovery area is configured and control file auto backup is off then:

RMAN> connect target /

RMAN> startup nomount;

RMAN> restore controlfile from 'C:*FRA**DBNAME*\backupset*date_of_backup**backupset_name*';

This will restore the control file to the location specified by the initialization parameter CONTROL_FILES mentioned in initialization parameter.

If restoration is being done using recovery catalog then:

RMAN> connect target /

RMAN> connect catalog *catalog_database_user/password@recovery_catalog_service*;

RMAN> startup nomount;

RMAN> restore controlfile;

This will restore the control file to the location specified by the initialization parameter CONTROL_FILES mentioned in initialization parameter.

If no flash recovery area is configured, no recovery catalog is available and RMAN backup piece is available at default location then:

RMAN> connect target /

RMAN> startup nomount;

RMAN> set dbid 1234567890;

RMAN> restore controlfile from autobackup;

Recovering from loss of a redo log file

Recovering from loss of a system-critical datafile

Recovering from loss of a non–system-critical datafile

XML Cheatsheet

Oracle XML Reference

Oracle possesses a variety of powerful XML features. A tremendous amount of documentation exists regarding Oracle's XML features. This resource is intended to be a cheat sheet for those of us who don't have time to wade through the hundreds of pages of documentation, but instead wish to quickly understand how to create simple XML output and input XML into a database.

Other Oracle References

- Oracle PL/SQL Reference

DBMS_XMLGEN

Overview

See also:

- Generating XML from Oracle9i Database Using DBMS_XMLGEN (http://download-west.oracle.com/docs/cd/B10501_01/appdev.920/a96620/xdb12gen.htm#1025388): Official Oracle Documentation.

Functions

getXML()

Gets the XML document by fetching the maximum number of rows specified. It appends the XML document to the CLOB passed in. Use this version of GETXML Functions to avoid any extra CLOB copies and to reuse the same CLOB for subsequent calls. Because of the CLOB reuse, this GETXML Functions call is potentially more efficient.

Syntax:

```
DBMS_XMLGEN.GETXML (
  ctx          IN ctxHandle,
  tmpclob      IN OUT NCOPY CLOB,
  dtdOrSchema  IN number := NONE)
RETURN BOOLEAN;
```

Generates the XML document and returns it as a temporary CLOB. The temporary CLOB obtained from this function must be freed using the DBMS_LOB.FREETEMPORARY call:

```
DBMS_XMLGEN.GETXML (
```

```
  ctx          IN ctxHandle,
  dtdOrSchema  IN number := NONE)
RETURN CLOB;
```

Converts the results from the SQL query string to XML format, and returns the XML as a temporary CLOB, which must be subsequently freed using the DBMS_LOB.FREETEMPORARY call:

```
DBMS_XMLGEN.GETXML (
  sqlQuery     IN VARCHAR2,
  dtdOrSchema  IN number := NONE)
RETURN CLOB;
```

Example:

The following procedure parses the fields in the employee table into XML and saves the XML as CLOB rows in a table.

```
CREATE OR REPLACE procedure dump_pcd AS
    qryCtx DBMS_XMLGEN.ctxHandle;
    result CLOB;
BEGIN
    qryCtx :=  dbms_xmlgen.newContext ('SELECT * from employees;');
    DBMS_XMLGEN.setRowTag(qryCtx, 'EMPLOYEE'); DBMS_XMLGEN.setMaxRows(qryCtx, 5);
    LOOP

        -- save the XML into the CLOB result.
        result :=  DBMS_XMLGEN.getXML(qryCtx);
        EXIT WHEN DBMS_XMLGEN.getNumRowsProcessed((qryCtx)=0);

        -- store the data to a temporary table
        INSERT INTO temp_clob_tab VALUES(result);

    END LOOP;
END dump_pcd;
```

setRowSetTag()

Sets the name of the root element of the document. The default name is ROWSET. Setting the rowSetTag to NULL will stop this element from being output. An error is produced if both the row and the rowset are NULL and there is more than one column or row in the output. The error is produced because the generated XML would not have a top-level enclosing tag.

Syntax:

```
DBMS_XMLGEN.setRowSetTag (
  ctx        IN ctxHandle,
  rowSetTag  IN VARCHAR2);
```

Example:

```
DBMS_XMLGEN.setRowSetTag ( ctxHandle, 'ALL ROWS' );
```

Sample output:

This encloses the entire XML result set in the tag specified by the second parameter.

```
<ALL ROWS>
  <ROW>
     <NAME>John Doe</NAME>
  </ROW>
  <ROW>
     <NAME>Jane Doe</NAME>
  </ROW>
  ...
</ALL ROWS>
```

setRowTag()

This function sets the name of the element each row. The default name is ROW. Setting this to NULL suppresses the ROW element itself. This produces an error if both the row and the rowset are NULL and there is more than one column or row in

the output. The error is returned because the generated XML must have a top-level enclosing tag.

Syntax:

```
DBMS_XMLGEN.setRowTag (
  ctx        IN ctxHandle,
  rowTag     IN VARCHAR2);
```

Example:

This tells the XML generator to enclose the columns of each row in an AUTHOR tag.

```
DBMS_XMLGEN.setRowTag ( ctxHandle, 'AUTHOR' );
```

Sample output:

Every row output is now enclosed inside the AUTHOR tag.

```
<ROWSET>
  <AUTHOR>
     <NAME>John Doe</NAME>
  </AUTHOR>
  <AUTHOR>
     <NAME>Jane Doe</NAME>
  </AUTHOR>
  ...
</ROWSET>
```

Examples

Dumping a Query Result as XML

Sample procedure for dumping the results of an SQL query as XML.

```
CREATE OR REPLACE procedure dump_pcd AS
    qryCtx DBMS_XMLGEN.ctxHandle;
    result CLOB;
BEGIN
    qryCtx :=  dbms_xmlgen.newContext ('SELECT * from employees;');
    DBMS_XMLGEN.setRowTag(qryCtx, 'EMPLOYEE'); DBMS_XMLGEN.setMaxRows(qryCtx, 5);
    LOOP
        result :=  DBMS_XMLGEN.getXML(qryCtx);
        EXIT WHEN DBMS_XMLGEN.getNumRowsProcessed((qryCtx)=0);
        INSERT INTO temp_clob_tab VALUES(result);
    END LOOP;
END dump_pcd;
```

The returned XML results will look similar to the following:

```
<?xml version=''1.0''?>
 <ROWSET>
  <EMPLOYEE>
    <EMPLOYEE_ID>30</EMPLOYEE_ID>
    <LAST_NAME>SCOTT</LAST_NAME>
    <SALARY>20000<SALARY>
  </EMPLOYEE>
  <EMPLOYEE>
    <EMPLOYEE_ID>31</EMPLOYEE_ID>
    <LAST_NAME>MARY</LAST_NAME>
    <AGE>25</AGE>
  </EMPLOYEE>
</ROWSET>
```

SQL Cheatsheet

This "cheat sheet" covers most of the basic functionality that an Oracle DBA needs to run basic queries and perform basic

tasks. It also contains information that a PL/SQL programmer frequently uses to write stored procedures. The resource is useful as a primer for individuals who are new to Oracle, or as a reference for those who are experienced at using Oracle.

A great deal of information about Oracle exists throughout the net. We developed this resource to make it easier for programmers and DBAs to find most of the basics in one place. Topics beyond the scope of a "cheatsheet" generally provide a link to further research.

Other Oracle References

- Oracle XML Reference—the XML reference is still in its infancy, but is coming along nicely.

SELECT

The SELECT statement is used to retrieve rows selected from one or more tables, object tables, views, object views, or materialized views.

```
SELECT *
FROM beverages
WHERE field1 = 'Kona'
AND field2 = 'coffee'
AND field3 = 122;
```

SELECT INTO

Select into takes the values *name*, *address* and *phone number* out of the table *employee*, and places them into the variables *v_employee_name*, *v_employee_address*, and *v_employee_phone_number*.

This *only* works if the query matches a single item. If the query returns no rows it raises the NO_DATA_FOUND built-in exception. If your query returns more than one row, Oracle raises the exception TOO_MANY_ROWS.

```
SELECT name,address,phone_number
INTO v_employee_name,v_employee_address,v_employee_phone_number
FROM employee
WHERE employee_id = 6;
```

INSERT

The INSERT statement adds one or more new rows of data to a database table.

insert using the VALUES keyword

```
INSERT INTO table_name VALUES ('Value1', 'Value2', ... );
INSERT INTO table_name( Column1, Column2, ... ) VALUES ( 'Value1', 'Value2', ... );
```

insert using a SELECT statement

```
INSERT INTO table_name( SELECT Value1, Value2, ... from table_name );
INSERT INTO table_name( Column1, Column2, ... ) ( SELECT Value1, Value2, ... from table_name );
```

DELETE

The DELETE statement is used to delete rows in a table.

deletes rows that match the criteria

```
DELETE FROM table_name WHERE some_column=some_value
DELETE FROM customer WHERE sold = 0;
```

UPDATE

The UPDATE statement is used to udpate rows in a table.

updates the entire column of that table

```
UPDATE customer SET state='CA';
```

updates the specific record of the table eg:

```
UPDATE customer SET name='Joe' WHERE customer_id=10;
```

updates the column invoice as paid when paid column has more than zero.

```
UPDATE movies SET invoice='paid' WHERE paid > 0;
```

SEQUENCES

Sequences are database objects that multiple users can use to generate unique integers. The sequence generator generates sequential numbers, which can help automatically generate unique primary keys, and coordinate keys across multiple rows or tables.

CREATE SEQUENCE

The syntax for a sequence is:

```
CREATE SEQUENCE sequence_name
    MINVALUE value
    MAXVALUE value
    START WITH value
    INCREMENT BY value
    CACHE value;
```

For example:

```
CREATE SEQUENCE supplier_seq
    MINVALUE 1
    MAXVALUE 9999999999999999999999999999
    START WITH 1
    INCREMENT BY 1
    CACHE 20;
```

ALTER SEQUENCE

Increment a sequence by a certain amount:

```
ALTER SEQUENCE <sequence_name> INCREMENT BY <integer>;
ALTER SEQUENCE seq_inc_by_ten  INCREMENT BY 10;
```

Change the maximum value of a sequence:

```
ALTER SEQUENCE <sequence_name> MAXVALUE <integer>;
ALTER SEQUENCE seq_maxval  MAXVALUE  10;
```

Set the sequence to cycle or not cycle:

```
ALTER SEQUENCE <sequence_name> <CYCLE | NOCYCLE>;
ALTER SEQUENCE seq_cycle NOCYCLE;
```

Configure the sequence to cache a value:

```
ALTER SEQUENCE <sequence_name> CACHE <integer> | NOCACHE;
ALTER SEQUENCE seq_cache NOCACHE;
```

Set whether or not to return the values in order

```
ALTER SEQUENCE <sequence_name> <ORDER | NOORDER>;
ALTER SEQUENCE seq_order NOORDER;
```

LTER SEQUENCE seq_order

Generate query from a string

It is sometimes necessary to create a query from a string. That is, if the programmer wants to create a query at run time (generate an Oracle query on the fly), based on a particular set of circumstances, etc.

Care should be taken not to insert user-supplied data directly into a dynamic query string, without first vetting the data very strictly for SQL escape characters; otherwise you run a significant risk of enabling data-injection hacks on your code.

Here is a very simple example of how a dynamic query is done. There are, of course, many different ways to do this; this is just an example of the functionality.

```
PROCEDURE oracle_runtime_query_pcd IS
    TYPE ref_cursor IS REF CURSOR;
    l_cursor        ref_cursor;

    v_query         varchar2(5000);
    v_name          varchar2(64);
BEGIN
    v_query := 'SELECT name FROM employee WHERE employee_id=5';
    OPEN l_cursor FOR v_query;
    LOOP
       FETCH l_cursor INTO v_name;
       EXIT WHEN l_cursor%NOTFOUND;
    END LOOP;
    CLOSE l_cursor;
END;
```

String operations

Length

Length returns an integer representing the length of a given string. It can be referred to as: **lengthb**, **lengthc**, **length2**, and **length4**.

```
length( string1 );
```

```
SELECT length('hello world') FROM dual;
this returns 11, since the argument is made up of 11 characters including the space
```

```
SELECT lengthb('hello world') FROM dual;
SELECT lengthc('hello world') FROM dual;
SELECT length2('hello world') FROM dual;
SELECT length4('hello world') FROM dual;
these also return 11, since the functions called are equivalent
```

Instr

Instr returns an integer that specifies the location of a sub-string within a string. The programmer can specify which appearance of the string they want to detect, as well as a starting position. An unsuccessful search returns 0.

```
instr( string1, string2, [ start_position ], [ nth_appearance ] )

instr( 'oracle pl/sql cheatsheet', '/');
this returns 10, since the first occurrence of "/" is the tenth character

instr( 'oracle pl/sql cheatsheet', 'e', 1, 2);
this returns 17, since the second occurrence of "e" is the seventeenth character

instr( 'oracle pl/sql cheatsheet', '/', 12, 1);
this returns 0, since the first occurrence of "/" is before the starting point, which is the 12th character
```

Replace

Replace looks through a string, replacing one string with another. If no other string is specified, it removes the string specified in the replacement string parameter.

```
replace( string1, string_to_replace, [ replacement_string ] );
replace('i am here','am','am not');
this returns "i am not here"
```

Substr

Substr returns a portion of the given string. The "start_position" is 1-based, not 0-based. If "start_position" is negative, substr counts from the end of the string. If "length" is not given, substr defaults to the remaining length of the string.

substr(*string*, start_position [, length])

```
SELECT substr( 'oracle pl/sql cheatsheet', 8, 6) FROM dual;
```

returns "`pl/sql`" since the "p" in "pl/sql" is in the 8th position in the string (counting from 1 at the "o" in "oracle")

```
SELECT substr( 'oracle pl/sql cheatsheet', 15) FROM dual;
```

returns "`cheatsheet`" since "c" is in the 15th position in the string and "t" is the last character in the string.

```
SELECT substr('oracle pl/sql cheatsheet', -10, 5) FROM dual;
```

returns "`cheat`" since "c" is the 10th character in the string, counting from the *end* of the string with "t" as position 1.

Trim

These functions can be used to filter unwanted characters from strings. By default they remove spaces, but a character set can be specified for removal as well.

```
trim ( [ leading | trailing | both ] [ trim-char ] from string-to-be-trimmed );
trim ('   removing spaces at both sides     ');
this returns "removing spaces at both sides"

ltrim ( string-to-be-trimmed [, trimming-char-set ] );
ltrim ('   removing spaces at the left side     ');
this returns "removing spaces at the left side     "

rtrim ( string-to-be-trimmed [, trimming-char-set ] );
rtrim ('   removing spaces at the right side     ');
this returns "   removing spaces at the right side"
```

DDL SQL

Tables

Create table

The syntax to create a table is:

```
CREATE TABLE [table name]
      ( [column name] [datatype], ... );
```

For example:

```
CREATE TABLE employee
      (id int, name varchar(20));
```

Add column

The syntax to add a column is:

```
ALTER TABLE [table name]
      ADD ( [column name] [datatype], ... );
```

For example:

```
ALTER TABLE employee
      ADD (id int)
```

Modify column

The syntax to modify a column is:

```
ALTER TABLE [table name]
      MODIFY ( [column name] [new datatype] );
```

ALTER table syntax and examples:

For example:

```
ALTER TABLE employee
      MODIFY( sickHours s float );
```

Drop column

The syntax to drop a column is:

```
ALTER TABLE [table name]
      DROP COLUMN [column name];
```

For example:

```
ALTER TABLE employee
      DROP COLUMN vacationPay;
```

Constraints

Constraint types and codes

Type Code	Type Description	Acts On Level
C	Check on a table	Column
O	Read Only on a view	Object
P	Primary Key	Object
R	Referential AKA Foreign Key	Column
U	Unique Key	Column
V	Check Option on a view	Object

Displaying constraints

The following statement shows all constraints in the system:

```
SELECT
       table_name,
```

```
       constraint_name,
       constraint_type
FROM user_constraints;
```

Selecting referential constraints

The following statement shows all referential constraints (foreign keys) with both source and destination table/column couples:

```
SELECT
       c_list.CONSTRAINT_NAME as NAME,
       c_src.TABLE_NAME as SRC_TABLE,
       c_src.COLUMN_NAME as SRC_COLUMN,
       c_dest.TABLE_NAME as DEST_TABLE,
       c_dest.COLUMN_NAME as DEST_COLUMN
FROM ALL_CONSTRAINTS c_list,
     ALL_CONS_COLUMNS c_src,
     ALL_CONS_COLUMNS c_dest
WHERE c_list.CONSTRAINT_NAME = c_src.CONSTRAINT_NAME
  AND c_list.R_CONSTRAINT_NAME = c_dest.CONSTRAINT_NAME
  AND c_list.CONSTRAINT_TYPE = 'R'
GROUP BY c_list.CONSTRAINT_NAME,
          c_src.TABLE_NAME,
         c_src.COLUMN_NAME,
         c_dest.TABLE_NAME,
         c_dest.COLUMN_NAME;
```

Setting constraints on a table

The syntax for creating a check constraint using a CREATE TABLE statement is:

```
CREATE TABLE table_name
(
    column1 datatype null/not null,
    column2 datatype null/not null,
    ...
    CONSTRAINT constraint_name CHECK (column_name condition) [DISABLE]
);
```

For example:

```
CREATE TABLE suppliers
(
    supplier_id  numeric(4),
    supplier_name  varchar2(50),
    CONSTRAINT check_supplier_id
    CHECK (supplier_id BETWEEN 100 and 9999)
);
```

Unique Index on a table

The syntax for creating a unique constraint using a CREATE TABLE statement is:

```
CREATE TABLE table_name
(
    column1 datatype null/not null,
    column2 datatype null/not null,
    ...
    CONSTRAINT constraint_name UNIQUE (column1, column2, column_n)
);
```

For example:

```
CREATE TABLE customer
(
    id   integer not null,
    name varchar2(20),
    CONSTRAINT customer_id_constraint UNIQUE (id)
);
```

Adding unique constraints

The syntax for a unique constraint is:

```
ALTER TABLE [table name]
      ADD CONSTRAINT [constraint name] UNIQUE( [column name] ) USING INDEX [index name];
```

For example:

```
ALTER TABLE employee
      ADD CONSTRAINT uniqueEmployeeId UNIQUE(employeeId) USING INDEX ourcompanyIndx_tbs;
```

Deleting constraints

The syntax for dropping (removing) a constraint is:

```
ALTER TABLE [table name]
      DROP CONSTRAINT [constraint name];
```

For example:

```
ALTER TABLE employee
      DROP CONSTRAINT uniqueEmployeeId;
```

See also: Oracle Constraints (http://www.psoug.org/reference/constraints.html)

INDEXES

An index is a method that retrieves records with greater efficiency. An index creates an entry for each value that appears in the indexed columns. By default, Oracle creates B-tree indexes.

Create an index

The syntax for creating an index is:

```
CREATE [UNIQUE] INDEX index_name
    ON table_name (column1, column2, . column_n)
    [ COMPUTE STATISTICS ];
```

UNIQUE indicates that the combination of values in the indexed columns must be unique.

COMPUTE STATISTICS tells Oracle to collect statistics during the creation of the index. The statistics are then used by the optimizer to choose an optimal execution plan when the statements are executed.

For example:

```
CREATE INDEX customer_idx
    ON customer (customer_name);
```

In this example, an index has been created on the customer table called customer_idx. It consists of only of the customer_name field.

The following creates an index with more than one field:

```
CREATE INDEX customer_idx
    ON supplier (customer_name, country);
```

The following collects statistics upon creation of the index:

```
CREATE INDEX customer_idx
    ON supplier (customer_name, country)
    COMPUTE STATISTICS;
```

Create a function-based index

In Oracle, you are not restricted to creating indexes on only columns. You can create function-based indexes.

The syntax that creates a function-based index is:

```
CREATE [UNIQUE] INDEX index_name
    ON table_name (function1, function2, . function_n)
    [ COMPUTE STATISTICS ];
```

For example:

```
CREATE INDEX customer_idx
    ON customer (UPPER(customer_name));
```

An index, based on the uppercase evaluation of the customer_name field, has been created.

To assure that the Oracle optimizer uses this index when executing your SQL statements, be sure that UPPER(customer_name) does not evaluate to a NULL value. To ensure this, add UPPER(customer_name) IS NOT NULL to your **WHERE** clause as follows:

```
SELECT customer_id, customer_name, UPPER(customer_name)
FROM customer
WHERE UPPER(customer_name) IS NOT NULL
ORDER BY UPPER(customer_name);
```

Rename an Index

The syntax for renaming an index is:

```
ALTER INDEX index_name
    RENAME TO new_index_name;
```

For example:

```
ALTER INDEX customer_id
    RENAME TO new_customer_id;
```

In this example, **customer_id** is renamed to **new_customer_id**.

Collect statistics on an index

If you need to collect statistics on the index after it is first created or you want to update the statistics, you can always use the **ALTER INDEX** command to collect statistics. You collect statistics so that oracle can use the indexes in an effective manner. This recalcultes the table size, number of rows, blocks, segments and update the dictionary tables so that oracle can use the data effectively while choosing the execution plan.

The syntax for collecting statistics on an index is:

```
ALTER INDEX index_name
    REBUILD COMPUTE STATISTICS;
```

For example:

```
ALTER INDEX customer_idx
    REBUILD COMPUTE STATISTICS;
```

In this example, statistics are collected for the index called **customer_idx**.

Drop an index

The syntax for dropping an index is:

```
DROP INDEX index_name;
```

For example:

```
DROP INDEX customer_idx;
```

In this example, the **customer_idx** is dropped.

DBA Related

User Management

Creating a user

ggOracle Database/Print version **The syntax for creating a user is:**

```
CREATE USER username IDENTIFIED BY password;
```

For example:

```
CREATE USER brian IDENTIFIED BY brianpass;
```

Granting privileges

The syntax for granting privileges is:

```
GRANT privilege TO Gulfnet_krishna(user);
```

For example:

```
GRANT dba TO brian;
```

Change password

The syntax for changing user password is:

```
ALTER USER username IDENTIFIED BY password;
```

For example:

```
ALTER USER brian IDENTIFIED BY brianpassword;
```

Importing and exporting

There are two methods of backing up and restoring database tables and data. The 'exp' and 'imp' tools are simpler tools geared towards smaller databases. If database structures become more complex or are very large (> 50 GB for example) then using the RMAN tool is more appropriate.

Import a dump file using IMP

This command is used to import Oracle tables and table data from a *.dmp file created by the 'exp' tool. Remember that this a command that is executed from the command line through $ORACLE_HOME/bin and not within SQL*Plus.

The syntax for importing a dump file is:

```
imp KEYWORD=value
```

There are number of parameters you can use for keywords.

To view all the keywords:

```
imp HELP=yes
```

An example:

```
imp brian/brianpassword FILE=mydump.dmp FULL=yes
```

PL/SQL

Operators

(incomplete)

Arithmetic operators

- Addition: +
- Subtraction: -
- Multiplication: *
- Division: /
- Power (PL/SQL only): **

Examples

gives all employees from customer id 5 a 5% raise

```
UPDATE employee SET salary = salary * 1.05
                  WHERE customer_id = 5;
```

determines the after tax wage for all employees

```
SELECT wage - tax FROM employee;
```

Comparison operators

- Greater Than: >
- Greater Than or Equal To: >=
- Less Than: <
- Less Than or Equal to: <=
- Equivalence: =
- Inequality: != ^= <> ¬= (depends on platform)

Examples

```
SELECT name, salary, email FROM employees WHERE salary > 40000;

SELECT name FROM customers WHERE customer_id < 6;
```

String operators

- Concatenate: ||

create or replace procedure addtest(a in varchar2(100), b in varchar2(100), c out varchar2(200)) IS begin C:=concat(a,'-',b);

Date operators

- Addition: +
- Subtraction: -

Types

Basic PL/SQL Types

Scalar type (defined in package STANDARD): NUMBER, CHAR, VARCHAR2, BOOLEAN, BINARY_INTEGER, LONG\LONG RAW, DATE, TIMESTAMP(and its family including intervals)

Composite types (user-defined types): TABLE, RECORD, NESTED TABLE and VARRAY

LOB datatypes : used to store an unstructured large amount of data

%TYPE - anchored type variable declaration

The syntax for anchored type declarations is

```
<var_name> <obj>%type [not null][:= <init-val>];
```

For example

```
name Books.title%type;   /*  name is defined as the same type as column 'title' of table  Books */

commission number(5,2) := 12.5;

x commission%type;   /*  x is defined as the same type as variable 'commission' */
```

Note:

1. Anchored variables allow for the automatic synchronization of the type of anchored variable with the type of <obj> when there is a change to the <obj> type.
2. Anchored types are evaluated at compile time, so recompile the program to reflect the change of <obj> type in the anchored variable.

Collections

A collection is an ordered group of elements, all of the same type. It is a general concept that encompasses lists, arrays, and other familiar datatypes. Each element has a unique subscript that determines its position in the collection.

```
--Define a PL/SQL record type representing a book:
TYPE book_rec IS RECORD
   (title                   book.title%TYPE,
    author                  book.author_last_name%TYPE,
    year_published          book.published_date%TYPE);

--define a PL/SQL table containing entries of type book_rec:
Type book_rec_tab IS TABLE OF book_rec%TYPE
     INDEX BY BINARY_INTEGER;

my_book_rec  book_rec%TYPE;
my_book_rec_tab book_rec_tab%TYPE;
...
my_book_rec := my_book_rec_tab(5);
find_authors_books(my_book_rec.author);
...
```

There are many good reasons to use collections.

- Dramatically faster execution speed, thanks to transparent performance boosts including a new optimizing compiler,

better integrated native compilation, and new datatypes that help out with number-crunching applications.

- The FORALL statement, made even more flexible and useful. For example, FORALL now supports nonconsecutive indexes.
- Regular expressions are available in PL/SQL in the form of three new functions (REGEXP_INSTR, REGEXP_REPLACE, and REGEXP_SUBSTR) and the REGEXP_LIKE operator for comparisons. (For more information, see "First Expressions" by Jonathan Gennick in this issue.)
- Collections, improved to include such things as collection comparison for equality and support for set operations on nested tables.

see also:

- Taking Up Collections (http://www.oracle.com/technology/oramag/oracle/03-sep/o53plsql.html)
- Oracle Programming with PL/SQL Collections (http://www.developer.com/db/article.php/3379271)

Stored logic

Functions

A function must return a value to the caller.

The syntax for a function is

```
CREATE [OR REPLACE] FUNCTION function_name [ (parameter [,parameter]) ]
RETURN [return_datatype]
IS
    [declaration_section]
BEGIN
    executable_section
    return [return_value]

    [EXCEPTION
        exception_section]
END [function_name];
```

For example:

```
CREATE OR REPLACE  FUNCTION to_date_check_null(dateString IN VARCHAR2, dateFormat IN VARCHAR2)
RETURN DATE IS
BEGIN
    IF dateString IS NULL THEN
        return NULL;
    ELSE
        return to_date(dateString, dateFormat);
    END IF;
END;
```

Procedures

A procedure differs from a function in that it must not return a value to the caller.

The syntax for a procedure is:

```
CREATE [OR REPLACE] PROCEDURE procedure_name [ (parameter [,parameter]) ]
IS
    [declaration_section]
BEGIN
    executable_section
    [EXCEPTION
        exception_section]
END [procedure_name];
```

When you create a procedure or function, you may define parameters. There are three types of parameters that can be declared:

1. **IN -** The parameter can be referenced by the procedure or function. The value of the parameter can not be overwritten

by the procedure or function.
2. **OUT** - The parameter can not be referenced by the procedure or function, but the value of the parameter can be overwritten by the procedure or function.
3. **IN OUT** - The parameter can be referenced by the procedure or function and the value of the parameter can be overwritten by the procedure or function.

Also you can declare a **DEFAULT** value;

```
CREATE [OR REPLACE] PROCEDURE procedure_name [ (parameter [IN|OUT|IN OUT] [DEFAULT value] [,parameter]) ]
```

The following is a simple example of a procedure:

```
  /* purpose: shows the students in the course specified by courseId */

  CREATE OR REPLACE Procedure GetNumberOfStudents
     ( courseId IN number, numberOfStudents OUT number )
  IS

      /* although there are better ways to compute the number of students,
         this is a good opportunity to show a cursor in action            */

      cursor student_cur is
      select studentId, studentName
          from course
          where course.courseId = courseId;
      student_rec    student_cur%ROWTYPE;

  BEGIN
      OPEN student_cur;
      LOOP
          FETCH student_cur INTO student_rec;
          EXIT WHEN student_cur%NOTFOUND;
          numberOfStudents := numberOfStudents + 1;
      END LOOP;
      CLOSE student_cur;

  EXCEPTION
  WHEN OTHERS THEN
        raise_application_error(-20001,'An error was encountered - '||SQLCODE||' -ERROR- '||SQLERRM);
  END GetNumberOfStudents;
```

anonymous block

```
DECLARE
 x NUMBER(4) := 0;
BEGIN
  x := 1000;
  BEGIN
    x := x + 100;
  EXCEPTION
    WHEN OTHERS THEN
      x := x + 2;
  END;
  x := x + 10;
  dbms_output.put_line(x);
EXCEPTION
  WHEN OTHERS THEN
    x := x + 3;
END;
```

Passing parameters to stored logic

There are three basic syntaxes for passing parameters to a stored procedure: positional notation, named notation and mixed notation.

The following examples call this procedure for each of the basic syntaxes for parameter passing:

```
CREATE OR REPLACE PROCEDURE create_customer( p_name IN varchar2,
                                             p_id IN number,
                                             p_address IN varchar2,
                                             p_phone IN varchar2 ) IS
BEGIN
    INSERT INTO customer ( name, id, address, phone )
    VALUES ( p_name, p_id, p_address, p_phone );
END create_customer;
```

Positional notation

Specify the same parameters in the same order as they are declared in the procedure. This notation is compact, but if you specify the parameters (especially literals) in the wrong order, the bug can be hard to detect. You must change your code if the procedure's parameter list changes.

```
create_customer('James Whitfield', 33, '301 Anystreet', '251-222-3154');
```

Named notation

Specify the name of each parameter along with its value. An arrow (=>) serves as the association operator. The order of the parameters is not significant. This notation is more verbose, but makes your code easier to read and maintain. You can sometimes avoid changing code if the procedure's parameter list changes, for example if the parameters are reordered or a new optional parameter is added. Named notation is a good practice to use for any code that calls someone else's API, or defines an API for someone else to use.

```
create_customer(p_address => '301 Anystreet', p_id => 33, p_name => 'James Whitfield', p_phone => '251-222-3154');
```

Mixed notation

Specify the first parameters with positional notation, then switch to named notation for the last parameters. You can use this notation to call procedures that have some required parameters, followed by some optional parameters.

```
create_customer(v_name, v_id, p_address=> '301 Anystreet', p_phone => '251-222-3154');
```

Table functions

```
CREATE TYPE object_row_type as OBJECT (
  object_type VARCHAR(18),
  object_name VARCHAR(30)
);

CREATE TYPE object_table_type as TABLE OF object_row_type;

CREATE OR REPLACE FUNCTION get_all_objects
  RETURN object_table_type PIPELINED AS
BEGIN
    FOR cur IN (SELECT * FROM all_objects)
    LOOP
      PIPE ROW(object_row_type(cur.object_type, cur.object_name));
    END LOOP;
    RETURN;
END;

SELECT * FROM TABLE(get_all_objects);
```

Flow control

Conditional Operators

- and: AND
- or: OR
- not: NOT

Example

IF salary > 40000 AND salary <= 70000 THEN() ELSE IF salary>70000 AND salary<=100000 THEN() ELSE()

If/then/else

```
IF [condition] THEN
    [statements]
ELSEIF [condition] THEN
```

```
    [statements}
ELSEIF [condition] THEN
    [statements}
ELSEIF [condition] THEN
    [statements}
ELSEIF [condition] THEN
    [statements}
ELSEIF [condition] THEN
    [statements}
ELSEIF [condition] THEN
    [statements}
ELSEIF [condition] THEN
    [statements}
ELSE
    [statements}
END IF;
```

Arrays

Associative arrays

- Strongly typed arrays, useful as in-memory tables

Example

- Very simple example, the index is the key to accessing the array so there is no need to loop through the whole table unless you intend to use data from every line of the array.
- The index can also be a numeric value.

```
DECLARE
    -- Associative array indexed by string:

    -- Associative array type
    TYPE population IS TABLE OF NUMBER
        INDEX BY VARCHAR2(64);
    -- Associative array variable
    city_population  population;
    i                VARCHAR2(64);
BEGIN
    -- Add new elements to associative array:
    city_population('Smallville')  := 2000;
    city_population('Midland')     := 750000;
    city_population('Megalopolis') := 1000000;

    -- Change value associated with key 'Smallville':
    city_population('Smallville') := 2001;

    -- Print associative array by looping through it:
    i := city_population.FIRST;

    WHILE i IS NOT NULL LOOP
        DBMS_OUTPUT.PUT_LINE
            ('Population of ' || i || ' is ' || TO_CHAR(city_population(i)));
        i := city_population.NEXT(i);
    END LOOP;

    -- Print selected value from a associative array:
    DBMS_OUTPUT.PUT_LINE('Selected value');
    DBMS_OUTPUT.PUT_LINE('Population of
END;
/

-- Printed results:
Population of Megalopolis is 1000000
Population of Midland is 750000
Population of Smallville is 2001
```

- More complex example, using a record

```
DECLARE
    -- Record type
    TYPE apollo_rec IS RECORD
    (
        commander   VARCHAR2(100),
        launch      DATE
    );
    -- Associative array type
    TYPE apollo_type_arr IS TABLE OF apollo_rec INDEX BY VARCHAR2(100);
    -- Associative array variable
    apollo_arr apollo_type_arr;
BEGIN
```

```
    apollo_arr('Apollo 11').commander := 'Neil Armstrong';
    apollo_arr('Apollo 11').launch :=   TO_DATE('July 16, 1969','Month dd, yyyy');
    apollo_arr('Apollo 12').commander := 'Pete Conrad';
    apollo_arr('Apollo 12').launch :=   TO_DATE('November 14, 1969','Month dd, yyyy');
    apollo_arr('Apollo 13').commander := 'James Lovell';
    apollo_arr('Apollo 13').launch :=   TO_DATE('April 11, 1970','Month dd, yyyy');
    apollo_arr('Apollo 14').commander := 'Alan Shepard';
    apollo_arr('Apollo 14').launch :=   TO_DATE('January 31, 1971','Month dd, yyyy');

    DBMS_OUTPUT.PUT_LINE(apollo_arr('Apollo 11').commander);
    DBMS_OUTPUT.PUT_LINE(apollo_arr('Apollo 11').launch);
end;
/

-- Printed results:
Neil Armstrong
16-JUL-69
```

APEX

String substitution

```
   * In SQL: :VARIABLE
   * In PL/SQL: V('VARIABLE') or NV('VARIABLE')
   * In text: &VARIABLE.
```

External links

PSOUG reference (http://www.psoug.org/reference/)

More Wikibooks

Introduction to SQL

Retrieved from "https://en.wikibooks.org/w/index.php?title=Oracle_Database/Print_version&oldid=3087283"
