PL/SQL

Structured query language (Sql)

Join explained:

https://community.oracle.com/thread/597030

SQL> select empno,ename,job,dname,loc from emp e right outer join dept d on(e.deptno=d.deptno);

Or

SQL> select empno,ename,job,dname,loc from emp e,dept d where e.deptno(+) = d.deptno;

SQL> select empno,ename,job,dname,loc from emp e left outer join dept d on(e.deptno=d.deptno);

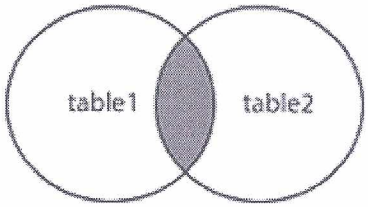
Or

SQL> select empno,ename,job,dname,loc from emp e,dept d where e.deptno=d.deptno(+);

http://dwhlaureate.blogspot.in/2012/08/joins-in-oracle.html

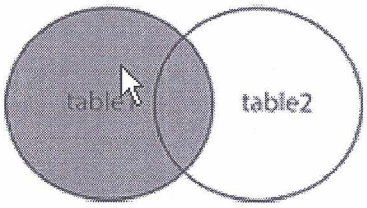
- **INNER JOIN**: Returns all rows when there is at least one match in BOTH tables

INNER JOIN



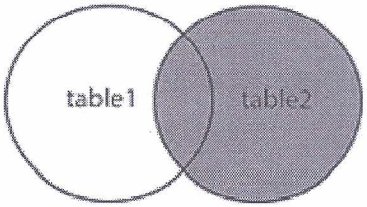
- **LEFT JOIN**: Return all rows from the left table, and the matched rows from the right table

LEFT JOIN

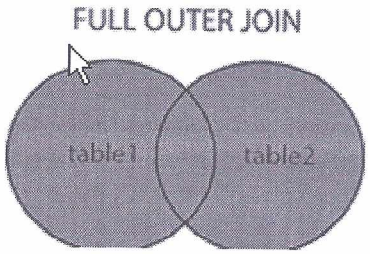


- **RIGHT JOIN**: Return all rows from the right table, and the matched rows from the left table

RIGHT JOIN



- **FULL JOIN**: Return all rows when there is a match in ONE of the tables
  Combination of left and right joins



Cross join or Cartesian join
http://www.w3resource.com/sql/joins/cross-join.php

A cross-join that does not have a 'where' clause gives the Cartesian product. Cartesian product result-set contains the number of rows in the first table, multiplied by the number of rows in second table.

Union and union all:

UNION and UNION ALL used to combine (set operation) two or more query results. UNION will eliminate duplicate rows and UNION ALL will display all rows.
The **Oracle UNION operator** is used to combine the result sets of 2 or more **Oracle** SELECT statements. It removes duplicate rows between the various SELECT statements. Each SELECT **statement** within the **UNION operator** must have the same number of fields in the result sets with similar data types

Pasted from <https://sqlandplsql.com/2012/06/22/difference-between-union-and-union-all-clause-oracle/>

Foreign key:
A FOREIGN KEY in one table points to a PRIMARY KEY in another table.

Primary key:
Primary Key is a combination of **Unique** and NOT NULL Constraints

Sql Check:
The following SQL creates a CHECK constraint on the "P_Id" column when the "Persons" table is created. The CHECK constraint specifies that the column "P_Id" must only include integers greater than 0.

Sql Default:
The DEFAULT constraint is used to insert a default value into a column.
The default value will be added to all new records, if no other value is specified.

Sql auto increment filed:
CREATE TABLE Persons
(
ID int NOT NULL **AUTO_INCREMENT**,
LastName varchar(255) NOT NULL,

City varchar(255),
PRIMARY KEY (ID)
)

NVL:
The Oracle/PLSQL NVL function lets you substitute a value when a null value is encountered.
select id, NVL(reportable, 1) as test from str_hub_transaction
where reportable is null;

DECODE:
The Oracle/PLSQL DECODE function has the functionality of an IF-THEN-ELSE statement.
select id, decode(<Column_name> ,'<Column value>', 'test',
'default') from <table name>;

COUNT(*) vs COUNT(column) vs COUNT(1)
1. Never use COUNT(*), it must read all columns and cause unnecessary reads.
2. Always use COUNT(1) because, generally, the primary key is the first column in the table and you want it to read the clustered index.
3. Always use COUNT(column_name) so that you can select which index it will scan.

The COUNT(column_name) function returns the number of values (NULL values will not be counted) of the specified column.

Having:
http://docs.oracle.com/javadb/10.8.3.0/ref/rrefsqlj14854.html
The Oracle HAVING clause is used in combination with the GROUP BY clause to restrict the groups of returned rows to only those whose the condition is TRUE.

substr function:
Substr(column_name, 3) - trim first 2 char and show rest of chars
Substr(column_name, 3,5) - trim first 2 char and show rest of chars till count 5.

The HAVING clause was added to SQL because the WHERE keyword could not be used with aggregate functions.
SELECT Employees.LastName, COUNT(Orders.OrderID) AS NumberOfOrders FROM (Orders
INNER JOIN Employees
ON Orders.EmployeeID=Employees.EmployeeID)
GROUP BY LastName
HAVING COUNT(Orders.OrderID) > 10;

Query to find 3rd highest sailary:
SELECT MIN(EmpSalary) from (SELECT EmpSalary from Employee ORDER BY EmpSalary DESC LIMIT 3); You can here just change the nth value after the LIMIT constraint. Here in this the Sub query Select EmpSalary from Employee Order by EmpSalary DESC Limit 3; would return the top 3 salaries of the Employees

```
SELECT * /*This is the outer query part */
FROM Employee Emp1
WHERE (N-1) = ( /* Subquery starts here */
SELECT COUNT(DISTINCT(Emp2.Salary))
FROM Employee Emp2
WHERE Emp2.Salary > Emp1.Salary)
```

Sub queries -
Corelated - sub query run only once on table
Uncoreleated - sub run on every iteration of output from main query.

http://www.programmerinterview.com/index.php/database-sql/find-nth-highest-salary-sql/

Between in SQL:

select * from <table_name>where id between <value1> and <value2>

Flashback query:

From Oracle Ver. 9i Oracle has introduced Flashback Query feature. It is useful to recover from accidental statement failures. For example, suppose a user accidently deletes rows from a table and commits it also then, using flash back query he can get back the rows.
Flashback feature depends upon on how much undo retention time you have specified. If you have set the **UNDO_RETENTION parameter to 2 hours then**, Oracle will not overwrite the data in undo tablespace even after committing until 2 Hours have passed. Users can recover from their mistakes made since last 2 hours only.

select * from <table name> as of timestamp sysdate-1 where id = '<value>';

Using Flashback Table to return Table to Past States.
FLASHBACK TABLE emp TO TIMESTAMP
TO_TIMESTAMP('2007-06-19 09:30:00', `YYYY-MM-DD HH24:MI:SS');

Recovering drop tables:
Flashback table emp to before drop;

TO_CHAR (number) converts *n* to a value of VARCHAR2 datatype
to_char(date_created_item, 'MM-DD-YY')
TO_CHAR (date_field, 'Month')
to_char(date_created_item, 'Month DD, YYYY HH24:MI:SS')

select * from table where to_char(date_created_item, 'Mon') = 'Sep';

To_date function coverts a string to a date

select Last_Day(ADD_MONTHS('12-May-2017',-1)) AS LAST_DAY from dual;
--30/04/2017
select Last_Day(ADD_MONTHS('12-May-2017',-2))+1 AS FIRST_DAY from dual;
--01/04/2017
select ADD_MONTHS('12-May-2017',-2) AS LAST_DAY from dual;
select last_day(sysdate) from dual;
select sysdate +1 from dual;
select extract(day from sysdate) from dual;
select (Last_Day(ADD_MONTHS('12-May-2017',-3)) - (Last_Day(ADD_MONTHS('12-May-2017',-4))+1))+1 AS days from dual;

Pasted from <http://www.oracle-dba-online.com/flash_back_features.htm>

Count of row without count function:

Select max (rownum) from <table name>;

Delete and truncate:

The DROP command removes a table from the database. All the tables' rows, indexes and privileges will also be removed. ... **DROP and TRUNCATE are DDL commands, whereas DELETE is a DML**

command. Therefore DELETE operations can be rolled back (undone), while DROP and TRUNCATE operations cannot be rolled back.
TRUNCATE is much faster than DELETE.
Reason:When you type DELETE.all the data get copied into the Rollback Tablespace first.then delete operation get performed.Thatswhy when you type ROLLBACK after deleting a table ,you can get back the data(The system get it for you from the Rollback Tablespace).All this process take time.But when you type TRUNCATE,it removes data directly without copying it into the Rollback Tablespace.Thatswhy TRUNCATE is faster.Once you Truncate you cann't get back the data.

TRUNCATE allows you to drop storage as well so releases hard storage back to the DB

Water Mark:

WATER MARK is a divided segment of used and free blocks. Blocks which are below high WATER MARK i.e. used blocks, have at least once contained some data. This data might have been deleted later. Oracle knows that blocks beyond high WATER MARK don't have data; it only reads blocks up to the high WATER MARK during a full table scan.

The high water mark level is just a line separate the used blocks and free blocks.
The blocks above the HWM level is free blocks, they are ready to use.
The blocks below the HWM level is used blocks, they are already used.
for example, if you delete some huge records from the database, that data will delete but the blocks are not ready to used, because that blocks are still below HWM level, so delete command never reset the HWM level,
At the same time you truncate the date, then the data will delete and that used blocks will goto above the HWM level, now its ready to used. now they consider has free blocks.

Hierarchy query:

http://www.orafaq.com/node/2038

```
SELECT (sysdate-1 + ROWNUM) dt
  FROM DUAL CONNECT BY ROWNUM < 5;

select fin_data  from (
SELECT DECODE (TO_CHAR (dt, 'dd'),  01, dt,  10, dt,  20, dt,  30, dt, sysdate) As fin_data
FROM (SELECT ((to_date('01-01-2015','DD-MM-YYYY')) - 1 + ROWNUM) dt
FROM DUAL
CONNECT BY ROWNUM < (select (to_date('30-05-2017','DD-MM-YYYY'))-(to_date('01-01-
2015','DD-MM-YYYY'))+1  from dual)))
where trunc(fin_data) != '22-June-2017';

        select fin_data from (
        SELECT case TO_CHAR (dt, 'dd')
        when '01' then dt
        when '10' then dt
        when '20' then dt
        when '30' then dt
        else sysdate
        end as fin_data
        FROM (SELECT (SYSDATE - 1 + ROWNUM) dt
        FROM DUAL
        CONNECT BY ROWNUM < 100))
        where trunc(fin_data) != '22-June-2017';


        SELECT
          ENAME
        FROM
          EMP
        WHERE
```

```
  LEVEL=2
CONNECT BY
  PRIOR EMPNO = MGR
START WITH
  ENAME = 'JONES';
```

http://www.dba-oracle.com/t_advanced_sql_hierarchical_queries.htm

Oracle

http://www.oracle-dba-online.com/sql/oracle_sql_tutorial.htm

Oracle internal join

- Hash join
- Sort merge join
- Nested loop join
- Cartesian join
- https://docs.oracle.com/cd/B28359_01/server.111/b28274/optimops.htm#i51523

Tuning PL/SQL Procedure Calls with the NOCOPY Compiler Hint (pass value by refrence)

By default, OUT and IN OUT parameters are passed by value. The values of any IN OUT parameters are copied before the subprogram is executed. During subprogram execution, temporary variables hold the output parameter values. If the subprogram exits normally, these values are copied to the actual parameters. If the subprogram exits with an unhandled exception, the original parameters are unchanged.
When the parameters represent large data structures such as collections, records, and instances of object types, this copying slows down execution and uses up memory. In particular, this overhead applies to each call to an object method: temporary copies are made of all the attributes, so that any changes made by the method are only applied if the method exits normally.
To avoid this overhead, you can specify the NOCOPY hint, which allows the PL/SQL compiler to pass OUT and IN OUT **parameters by reference**. If the subprogram exits normally, the behavior is the same as normal. If the subprogram exits early with an exception, the values of OUT and IN OUT parameters (or object attributes) might still change. To use this technique, ensure that the subprogram handles all exceptions.

Objects in pl/sql

An object type can represent any real-world entity. For example, an object type can represent a student, bank account, computer screen, rational number, or data structure such as a queue, stack, or list.

```
CREATE TYPE address_typ AS OBJECT (
street VARCHAR2(30),
city VARCHAR2(20),
state CHAR(2),
postal_code VARCHAR2(6) );
/
CREATE TYPE employee_typ AS OBJECT (
employee_id NUMBER(6),
first_name VARCHAR2(20),
last_name VARCHAR2(25),
email VARCHAR2(25),
phone_number VARCHAR2(20),
hire_date DATE,
job_id VARCHAR2(10),
salary NUMBER(8,2),
commission_pct NUMBER(2,2),
manager_id NUMBER(6),
department_id NUMBER(4),
address address_typ,
MAP MEMBER FUNCTION get_idno RETURN NUMBER,
MEMBER PROCEDURE display_address ( SELF IN OUT NOCOPY employee_typ ) );
/
```

```
CREATE TYPE BODY employee_typ AS
MAP MEMBER FUNCTION get_idno RETURN NUMBER IS
BEGIN
RETURN employee_id;
END;
MEMBER PROCEDURE display_address ( SELF IN OUT NOCOPY employee_typ ) IS
BEGIN
DBMS_OUTPUT.PUT_LINE(first_name || ' ' || last_name);
DBMS_OUTPUT.PUT_LINE(address.street);
DBMS_OUTPUT.PUT_LINE(address.city || ', ' || address.state || ' ' ||
address.postal_code);
END;
END;
/
CREATE TABLE employee_tab OF employee_typ;
```

The **MAP functions** - key value pair. Map have unique key
If someone sends me two employee records, one for primary key=123 and the second one for primary key = 123, then I know that these are referring to the *same* employee. But what about objects...they dont *have* a primary a key as such.
That is what a map member function does.

https://asktom.oracle.com/pls/asktom/f?p=100:11:0::::P11_QUESTION_ID:9531685800346795928

Updating and Deleting Rows in an Object Table

```
CREATE TABLE employee_tab OF employee_typ;
DECLARE
emp employee_typ;
BEGIN
INSERT INTO employee_tab VALUES (employee_typ(370, 'Robert', 'Myers', 'RMYERS',
'555.111.2277', '07-NOV-04', 'SA_REP', 8800, .12, 101, 110,
address_typ('540 Fillmore', 'San Francisco', 'CA', '94011')) );
UPDATE employee_tab e SET e.address.street = '1040 California'
WHERE e.employee_id = 370;
DELETE FROM employee_tab e WHERE e.employee_id = 310;
END;
```

Declaring a Nested Table in SQL

```
CREATE TYPE CourseList AS TABLE OF VARCHAR2(10) -- define type
/
CREATE TYPE student AS OBJECT ( -- create object
id_num INTEGER(4),
name VARCHAR2(25),
address VARCHAR2(35),
status CHAR(2),
courses CourseList); -- declare nested table as attribute
/
CREATE TABLE sophomores of student
NESTED TABLE courses STORE AS courses_nt;
```

Database link

```
CREATE DATABASE LINK <link name> CONNECT TO <database user> IDENTIFIED BY
'password' USING  <db name>;
```

Need usee_name, password and database name to create db link

```
SELECT * FROM emp@<link name>;
```

Pasted from
<https://www.enterprisedb.com/docs/en/9.4/oracompat/Database_Compatibility_for_Oracle_Developers_Guide.1.054.html>

Advantages of PL/SQL Packages

-Packages let you encapsulate logically related types, items, and subprograms in a named PL/SQL module
-With packages, you can specify which types, items, and subprograms are public (visible and accessible) or private (hidden and inaccessible).
-When you call a packaged subprogram for the first time, the whole package is loaded into memory. Later calls to related subprograms in the package require no disk I/O.
-Data hiding

Separating Cursor Specifications With Packages

```
CREATE PACKAGE emp_stuff AS
CURSOR c1 RETURN employees%ROWTYPE; -- declare cursor spec
END emp_stuff;
/
CREATE PACKAGE BODY emp_stuff AS
CURSOR c1 RETURN employees%ROWTYPE IS
SELECT * FROM employees WHERE salary > 2500; -- define cursor body
END emp_stuff;
/
```

The cursor spec has no SELECT statement because the RETURN clause specifies the datatype of the return value. However, the cursor body must have a SELECT statement and the same RETURN clause as the cursor spec. Also, the number and datatypes of items in the SELECT list and the RETURN clause must match.

subquery

A SELECT statement that provides a set of rows for processing. Its syntax is like the select_into_statement without the INTO clause.

Correlated Subqueries

While a subquery is evaluated only once for each table, a correlated subquery is evaluated once for each row. Example 6–21 returns the name and salary of each employee whose salary exceeds the departmental average. For each row in the table, the correlated subquery computes the average salary for the corresponding department.

Example 6–21 Using a Correlated Subquery

```
DECLARE
-- For each department, find the average salary. Then find all the employees in
-- that department making more than that average salary.
CURSOR c1 IS
SELECT department_id, last_name, salary FROM employees t
WHERE salary >
( SELECT AVG(salary) FROM employees WHERE t.department_id = department_id )
ORDER BY department_id;
BEGIN
FOR person IN c1
LOOP
DBMS_OUTPUT.PUT_LINE('Making above-average salary = ' || person.last_name);
END LOOP;
END;
/
```

Explicit lock

You can request data locks on specific rows or entire tables **if you need to override default locking**. Explicit locking lets you deny access to data for the duration of a

transaction.:
■ With the LOCK TABLE statement, you can explicitly lock entire tables.
- LOCK TABLE table IN ROW SHARE MODE;
- LOCK TABLE table IN EXCLUSIVE MODE;
■ With the **SELECT FOR UPDATE** statement, you can explicitly lock specific rows of a table to make sure they do not change after you have read them.

When you declare a cursor that will be referenced in the CURRENT OF clause of an UPDATE or DELETE statement, you must use the FOR UPDATE clause to acquire exclusive row locks. An example follows:

```
DECLARE
CURSOR c1 IS SELECT employee_id, salary FROM employees
WHERE job_id = 'SA_REP' AND commission_pct > .10
FOR UPDATE NOWAIT;
```

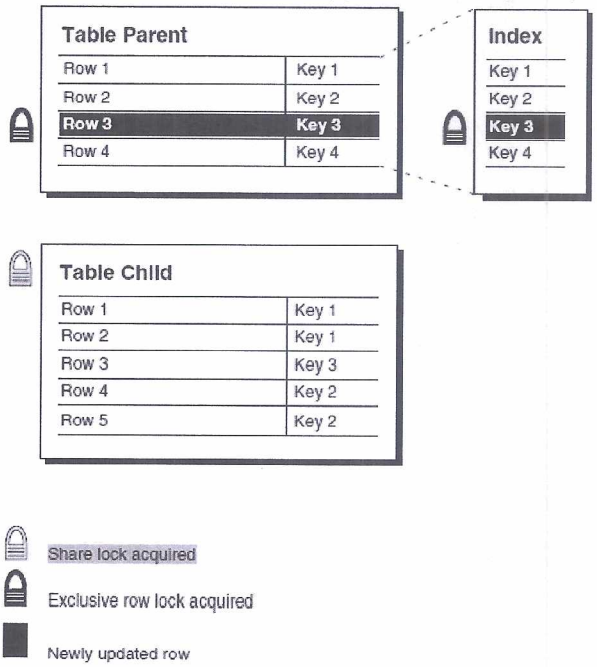
The SELECT ... FOR UPDATE statement identifies the rows that will be updated or deleted, then locks each row in the result set. This is useful when you want to base an update on the existing values in a row. In that case, you must make sure the row is not changed by another user before the update.
The optional keyword NOWAIT tells Oracle not to wait if requested rows have been locked by another user. Control is immediately returned to your program so that it can do other work before trying again to acquire the lock. If you omit the keyword NOWAIT, Oracle waits until the rows are available.

You use the LOCK TABLE statement to lock entire database tables in a specified lock mode so that you can share or deny access to them. Row share locks allow concurrent access to a table; they prevent other users from locking the entire table for exclusive use. Table locks are released when your transaction issues a commit or rollback.

```
LOCK TABLE employees IN ROW SHARE MODE NOWAIT;
```

The lock mode determines what other locks can be placed on the table

In case of primary and foreign key locking mechanism; when exclusive lock is applied on primary key; a shared lock is applied on foreign key row.



Deadlocks

A deadlock can occur when two or more users are waiting for data locked by each other. Deadlocks prevent some transactions from continuing to work.

Deadlock Detection

Oracle automatically detects deadlock situations and resolves them by rolling back one of the statements involved in the deadlock, thereby releasing one set of the conflicting row locks. A corresponding message also is returned to the transaction that undergoes statement-level rollback. The statement rolled back is the one belonging to the transaction that detects the deadlock. Usually, the signalled transaction should be rolled back explicitly, but it can retry the rolled-back statement after waiting.

Execution plan and stats gather

EXEC DBMS_STATS.gather_table_stats('<Schema name>','<Table name>');

You can influence the optimizer's choices by setting the optimizer approach and goal. Objects with stale or no statistics are automatically analyzed. You can also gather statistics for the optimizer using the PL/SQL package DBMS_STATS.
Sometimes the application designer, who has more information about a particular application's data than is available to the optimizer, can choose a more effective way to run a SQL statement. The application designer can use hints fin SQL statements to specify how the statement should be run.

PARALLEL is an example of hint

```
SELECT /* +parallel(c,2) */  *
FROM sh.customers c
ORDER BY cust_first_name, cust_last_name, cust_year_of_birth
```

Full table scan

```
Select /*+ FULL(emp) */ name
From emp
Where job = 'SALESMAN';
```

Ordered

The optimizer hint ORDERED influences the optimizer in selecting an **execution plan to drive the predicate filter in the order of how the tables are specified in the FROM clause.**

Pasted from <https://support.esri.com/en/technical-article/000010255>

First_rows(n)

```
SELECT /*+ FIRST_ROWS(10) */ empno, ename
FROM emp
WHERE deptno = 10;
```

The hints FIRST_ROWS(*n*) (where *n* is any positive integer) or FIRST_ROWS instruct Oracle to optimize an individual SQL statement for fast response. FIRST_ROWS(n) affords greater precision, because it instructs Oracle to choose the plan that returns the first *n* rows most efficiently. The FIRST_ROWS hint, which optimizes for the best plan to return the first single row, is retained for backward compatibility and plan stability.

The optimizer ignores this hint in DELETE and UPDATE statement blocks and in SELECT statement blocks that contain any of the following syntax:

- **Set operators (UNION, INTERSECT, MINUS, UNION ALL)**
- **GROUP BY clause**
- **FOR UPDATE clause**
- **Aggregate functions**
- **DISTINCT operator**
- **ORDER BY clauses, when there is no index on the ordering columns**

These statements cannot be optimized for best response time, because Oracle must retrieve all rows accessed by the statement before returning the first row. If you specify this hint in any of

these statements, then the optimizer uses the cost-based approach and optimizes for best throughput.
If you specify either the ALL_ROWS or the FIRST_ROWS hint in a SQL statement, and if the data dictionary does not have statistics about tables accessed by the statement, then the optimizer uses default statistical values (such as allocated storage for such tables) to estimate the missing statistics and, subsequently, to choose an execution plan

NO_INDEX(<*table_name*> < *index_name*>): Instructs the optimizer to specifically not use the **named index** in determining a plan.

```
SELECT /*+ NO_INDEX(emp emp_ix) */ empno, ename
FROM emp, dept
WHERE emp.deptno = dept.deptno;
```

CACHE(*table*): This hint tells Oracle to add the blocks retrieved for the table to the head of the most recently used list. This might be useful with regularly-used lookup tables, for example.

```
SELECT /*+ CACHE (d) */ deptno, dname
FROM dept d;
```

https://www.red-gate.com/simple-talk/sql/oracle/a-beginners-guide-to-optimizer-hints/
http://www.dba-oracle.com/t_sql_hints_tuning.htm

Replace :

REPLACE(string1, string_to_replace [, replacement_string])

REPLACE(old_string, NULL, v_string);

Pasted from <https://www.techonthenet.com/oracle/functions/replace.php>

Lock session

You can query the ***dba_blockers*** and ***dba_waiters*** views to locate blocking sessions, but you can also get this information from *v$lock* and *v$session*.

Pasted from <http://www.dba-oracle.com/t_find_blocking_sessions.htm>

Query plan

```
Select * from v$sql_plan;

SELECT SID,
     SERIAL#,
     SQL_ID,
     SOFAR,
     TOTALWORK,
     TIME_REMAINING,
     ELAPSED_SECONDS,
     MESSAGE,
     QCSID
 FROM gv$session_longops
WHERE TIME_REMAINING > 0 order by TIME_REMAINING desc;

select * from table(dbms_xplan.display_cursor('<Sql_id>'));

select * from V$SQLTEXT where sql_id = '99bg8qt7rs019';
Select * from v$sql where sql_id = '99bg8qt7rs019';
```

Analyze the execution plans and performance of the SQL statements, using:
- EXPLAIN PLAN statement
- **SQL Trace** facility with **TKPROF** utility
- Rewrite the SQL statements if necessary. For example, query hints can avoid problems such as unnecessary full-table scans.

There are two different methods you can use to look at the execution plan of a SQL statement:

1. **EXPLAIN PLAN command** - This displays an execution plan for a SQL statement without actually executing the statement.
2. **V$SQL_PLAN** - A dictionary view that shows the execution plan for a SQL statement that has been compiled into a cursor in the cursor cache.

Pasted from <https://blogs.oracle.com/optimizer/how-do-i-display-and-read-the-execution-plans-for-a-sql-statement>

Oracle architecture -

Index

Oracle provides several indexing schemes, which provide complementary performance functionality:
- B-tree indexes
- B-tree cluster indexes
- Hash cluster indexes
- Reverse key indexes
- Bitmap indexes
- Bitmap join indexes

```
CREATE INDEX employees_idx1 ON employees (last_name, job_id);
CREATE INDEX employees_idx2 ON employees (job_id, last_name);
```

Composite Indexes - concatenated index

A composite index (also called a concatenated index) is an index that you create on multiple columns in a table. Columns in a composite index can appear in any order and need not be adjacent in the table.
Composite indexes can speed retrieval of data for SELECT statements in which the WHERE clause references all or the leading portion of the columns in the composite index. Therefore, the order of the columns used in the definition is important. Generally, the most commonly accessed or most selective columns go first.

No more than 32 columns can form a regular composite index.

Indexes and Nulls

NULL values in indexes are considered to be distinct except when all the non-NULL values in two or more rows of an index are identical, in which case the rows are considered to be identical. Therefore, UNIQUE indexes prevent rows containing NULL values from being treated as identical. This does not apply if there are no non-NULL values—in other words, if the rows are entirely NULL.
Oracle does not index table rows in which all key columns are NULL, except in the case of bitmap indexes or when the cluster key column value is NULL.

Sort in Index

```
CREATE INDEX employees_last_name ON employees(last_name);
```

Oracle sorts the employees table on the last_name column. It then loads the index with the last_name and corresponding rowid values in this sorted order. When it uses the index, Oracle does a quick search through the sorted last_name values and

then uses the associated rowid values to locate the rows having the sought last_ name value.

Index in case of like

When we use LIKE with **preceding %** it always goes for full scan.
My question is
1. Why
2. In such case is full scan the best option. Is it possible to get a index scan with a better cost ?
1) because it would have to inspect each and every index entry - think about it, you are asking for all strings that start with *antyhing*, which means *everything* might apply
2) if you expect that very few rows will actually be returned, and an index would be appropriate, you can use TEXT to build an index that can retrieve data efficiently using leading and trailing wildcards

There are instances where Oracle will use an index to resolve a like with the pattern of '%text%'. If the query can be resolved without having to go back to the table (rowid lookup), the index may be chosen. Example:
select distinct first_nm from person where first_nm like '%EV%';
yes, if it can use the index to AVOID going to the table - in that case it would do an index fast full scan - a full scan of the smaller index.
but it isn't really using the index as an index in that case - just as a faster means to full scan the table (by using this skinnier version of it)

Reverse Key Indexes

Creating a reverse key index, compared to a standard index, reverses the bytes of each column indexed (except the rowid) while keeping the column order

Fully indexing a large table with a traditional B-tree index can be prohibitively expensive in terms of space, because the index can be several times larger than the data in the table. Bitmap indexes are typically only a fraction of the size of the indexed data in the table.

, fast access to rows. Oracle Database supports several types of index:

- [redacted]. (By default, Oracle Database creates **B-tree indexes (balanced tree)**.) B-tree indexes are most effective for high-cardinality data: that is, for data with many possible values, such as customer_name or phone_number. In a data warehouse, B-tree indexes should be used only for unique columns or other columns with very high cardinalities (that is, columns that are almost unique). The majority of indexes in a data warehouse should be bitmap indexes.

- Braches and leaf (where action data resides)
- All data in an indexed column resides at the same leaf level; which means fetching any record for a column value will take approx same time.
- High cardinal data

  Pasted from <https://docs.oracle.com/cd/B10500_01/server.920/a96520/indexes.htm>

  http://searchsqlserver.techtarget.com/definition/B-tree
  https://docs.oracle.com/cd/E11882_01/server.112/e40540/indexiot.htm#CNCPT1170 - explained

- [redacted], which store rowids associated with a key value as a bitmap
  The advantages of using bitmap indexes are greatest for columns in which the ratio of the number of distinct values to the number of rows in the table is under 1% (**Cardinality**).

A gender column, which has only two distinct values (male and female), is ideal for a bitmap index. However, data warehouse administrators also build bitmap indexes on columns with higher cardinalities.

An index provides pointers to the rows in a table that contain a given key value. A regular index stores a list of **rowids** for each key corresponding to the rows with that key value. In a bitmap index, a bitmap for each key value replaces a list of rowids.

```
CREATE INDEX index_name
ON table_name (column_name)
```

degree of cardinality
The number of unique values of a column divided by the total number of rows in the table.
This is particularly important when deciding which indexes to build. You typically want to use bitmap indexes on low degree of cardinality columns and B-tree indexes on high degree of cardinality columns. As a general rule, a cardinality of under 1% makes a good candidate for a bitmap index.

The following shows a portion of a com	pany's customers table.

```
SELECT cust_id, cust_gender, cust_marital_status, cust_income_level
FROM customers;
CUST_ID    C CUST_MARITAL_STATUS  CUST_INCOME_LEVEL
---------- - -------------------- ---------------------
...
     70 F                 D: 70,000 - 89,999
     80 F married           H: 150,000 - 169,999
     90 M single           H: 150,000 - 169,999
    100 F                 I: 170,000 - 189,999
    110 F married           C: 50,000 - 69,999
    120 M single           F: 110,000 - 129,999
    130 M                 J: 190,000 - 249,999
    140 M married           G: 130,000 - 149,999
...
```

Because cust_gender, cust_marital_status, and cust_income_level are all low-cardinality columns (there are only three possible values for marital status and region, two possible values for gender, and 12 for income level), bitmap indexes are ideal for these columns. Do not create a bitmap index on cust_id because this is a unique column. Instead, a unique B-tree index on this column provides the most efficient representation and retrieval.

https://docs.oracle.com/cd/B10500_01/server.920/a96520/indexes.htm

Cluster and non cluster
A **table cluster** is a group of tables that share common columns and store related data in the same blocks. **When tables are clustered, a single data block can contain rows from multiple tables**. For example, a block can store rows from both the employees and departments tables rather than from only a single table.
The **cluster key** is the column or columns that the clustered tables have in common. For example, the employees and departments tables share thedepartment_id column. You specify the cluster key when creating the table cluster and when creating every table added to the table cluster.
The **cluster key value** is the value of the cluster key columns for a particular set of rows. All data that contains the same cluster key value, such as department_id=20, is physically stored together. Each cluster key value is stored only once in the cluster and the cluster index, no matter how many rows of different tables contain the value.

```
CREATE CLUSTER employees_departments_cluster
  (department_id NUMBER(4))
SIZE 512;
CREATE INDEX idx_emp_dept_cluster ON CLUSTER employees_departments_cluster;

      CREATE TABLE employees ( ... )
        CLUSTER employees_departments_cluster (department_id);
      CREATE TABLE departments ( ... )
        CLUSTER employees_departments_cluster (department_id);
```

Pasted from <https://docs.oracle.com/cloud/latest/db112/CNCPT/tablecls.htm#CNCPT88825>

Hash Clusters

Hash clusters group table data in a manner similar to regular index clusters (clusters keyed with an index rather than a hash function). However, a row is stored in a hash cluster based on the result of applying a hash function to the row's cluster key value. All rows with the same key value are stored together on disk.
Hash clusters are a better choice than using an indexed table or index cluster when a table is queried frequently with equality queries (for example, return all rows for department 10). For such queries, the specified cluster key value is hashed. The resulting hash key value points directly to the area on disk that stores the rows.

Oracle uses a hash function to generate a distribution of numeric values, called hash values, which are based on specific cluster key values.

https://www.relationaldbdesign.com/extended-database-features/module3/clustered-tables-advantages.php

Clustered index (A table can have only one clustered index):

The primary key created for the StudId column will create a clustered index for the Studid column. A table can have only one clustered index on it.
When creating the clustered index, Oracle reads the Studid column and forms a Binary tree on it. This binary tree information is then stored separately in the disc

Pasted from <http://oracle-surya.blogspot.in/2012/05/cluster-and-non-cluster-index-with.html>

Non Clustered Index:

A non-clustered index is useful for columns that have some repeated values. Say for example, AccountType column of a bank database may have 10 million rows. But, the distinct values of account type may be 10-15. A clustered index is automatically created when we create the primary key for the table. We need to take care of the creation of the non-clustered index.

A table can have more than one Non-Clustered index. But, it should have only one clustered index that works based on the Binary tree concept. Non-Clustered column always depends on the Clustered column on the database.

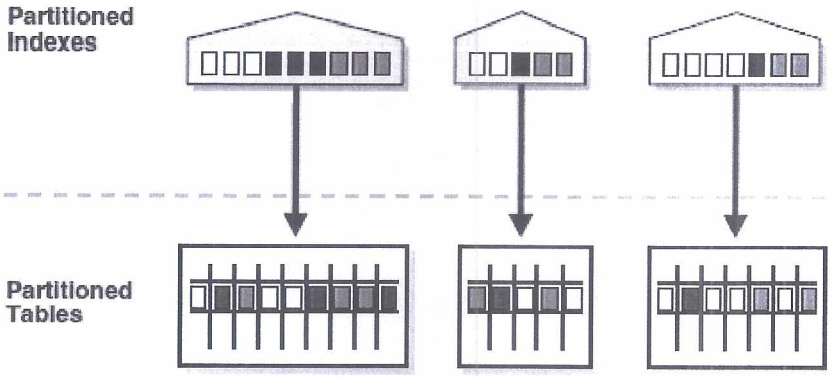
Pasted from <http://oracle-surya.blogspot.in/2012/05/cluster-and-non-cluster-index-with.html>

- Partitioned indexes, which consist of partitions containing an entry for each value that appears in the indexed column(s) of the table

Local partition index -

each partition of a local index is associated with exactly one partition of the table. This enables Oracle to automatically keep the index partitions in sync with the table partitions, and makes each table-index pair independent.

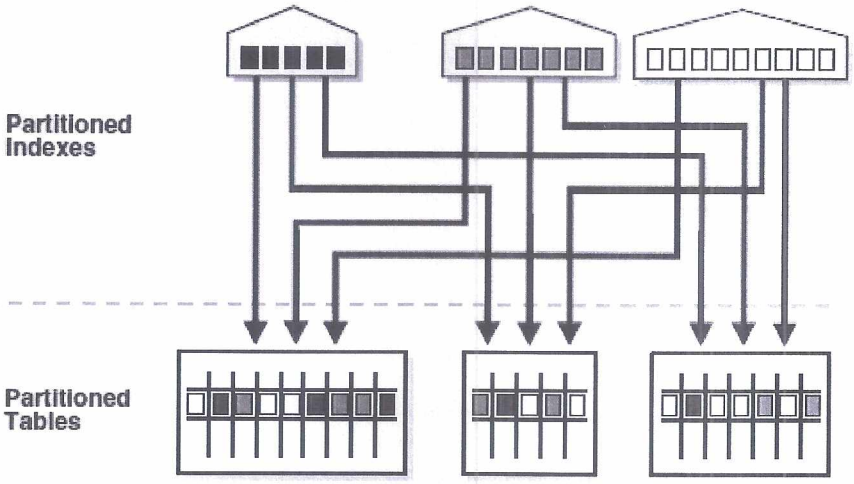
Figure 18–6 Local Partitioned Index



Global partition index -

- Global Range Partitioned Indexes
- Global Hash Partitioned Indexes

Figure 18–7 Global Partitioned Index



- https://oracle-base.com/articles/8i/partitioned-tables-and-indexes
  Function-based indexes(FBI), which are based on expressions. They enable you to construct queries that evaluate the value returned by an expression, which in turn may include built-in or user-defined functions.
  http://www.dba-oracle.com/oracle_tips_index_scan_fbi_sql.htm
  For example, these WHERE clause predicates might invoke an unexpected full-table scan:

  WHERE trunc(ship_date) > trunc(sysdate-7);

  WHERE to_char(ship_date,'YYYY-MM-DD') = '2004-01-04';

  Even though the ship_date column may have an index, the trunc and to_char built-in functions will invalidate the index, causing sub-optimal execution with unnecessary I/O:
  1. A less selective index would be used.
  2. A full-table scan might be performed.

  To avoid the index invalidation issues you have several options:

1. **Fix Predicate** - You can rewrite the query to leave the ship_date predicate alone:
   WHERE ship_date >= trunc(ship_date-7) + 1;

   WHERE ship_date = to_date(?2004-01-04?,'YYYY-MM-DD');

2. **Call FBI** - Create a function-based index on the predicate:
   Create index trunc_ship_date on product(trunc(ship_date));

   Create index char_ship_date on product(to_char(ship_date,'YYYY-MM-DD');

Pasted from <http://www.dba-oracle.com/oracle_tips_index_scan_fbi_sql.htm>

- **Domain indexes**, which are instances of an application-specific index of type ***indextype*** Oracle provides extensible indexing to accommodate indexes on customized complex datatypes such as documents, spatial data, images, and video clips and to make use of specialized indexing techniques.

Parallel hint:

http://www.toadworld.com/platforms/oracle/w/wiki/383.oracle-parallel-sql
SELECT /*+
parallel(c,2) */ *
FROM sh.customers c
ORDER BY cust_first_name, cust_last_name, cust_year_of_birth

TableSpace and datafile:

Database -> table space -> datafile

Databases, tablespaces, and datafiles are closely related, but they have important differences:

- An Oracle database consists of one or more logical storage units called tablespaces, which collectively store all of the database's data.
- Each tablespace in an Oracle database consists of one or more files called datafiles, which are physical structures that conform to the operating system in which Oracle is running.
- A database's data is collectively stored in the datafiles that constitute each tablespace of the database. For example, the simplest Oracle database would have one tablespace and one datafile. Another database can have three tablespaces, each consisting of two datafiles (for a total of six datafiles).

Pasted from <https://docs.oracle.com/cd/B19306_01/server.102/b14220/physical.htm>

Tablespaces are the bridge between certain physical and logical components of the Oracle database. Tablespaces are where you store Oracle database objects such as tables, indexes and rollback segments. You can think of a tablespace like a shared disk drive in Windows. You can store files on your shared drive, move files around and remove files. The same is true with tablespaces.
A tablespace is made up of one or more database datafiles. The total amount of storage space available in a tablespace is the sum of the physical disk size of all the datafiles associated with that tablespace (less some system overhead). The datafiles are created automatically when the tablespace is defined. In most cases, all datafile space is pre-allocated; that is, the space is set aside when the datafile is created. Thus, when you create a tablespace, you define the initial size of the associated datafile. We will discuss the specifics of creating tablespaces later in this book.

Pasted from <http://www.dba-oracle.com/concepts/tablespaces_concepts.htm>

Partition (Recreate table with partition information):

http://www.oracle-dba-online.com/sql/oracle_table_partition.htm

```
create table sales (year number(4),
                product varchar2(10),
               amt number(10,2))
    partition by range (year)
    partition p1 values less than (1992) tablespace u1,
    partition p2 values less than (1993) tablespace u2,
    partition p3 values less than (1994) tablespace u3,
    partition p4 values less than (1995) tablespace u4,
  partition p5 values less than (MAXVALUE) tablespace u5;
```

In Oracle you can partition a table by

- Range Partitioning
- Hash Partitioning
- List Partitioning
- Composite Partitioning

To see how many partitioned tables are there in your schema give the following statement

- select * from user_part_tables;

To see on partition level partitioning information

- select * from user_tab_partitions;

Space used by table in oracle:

```
select segment_name,sum(bytes)/(1024*1024*1024) GB from dba_segments
where owner = '<owner name>
and segment_type = 'TABLE'
group by segment_name
order by GB desc;
```

Queris used to get relation info on table_name, table_space and data_file

```
select t.tablespace,  t.totalspace as " Totalspace(MB)",round((t.totalspace-
fs.freespace),2) as "Used Space(MB)",
fs.freespace as "Freespace(MB)",round(((t.totalspace-fs.freespace)/t.totalspace)*100,2)
as "% Used",
round((fs.freespace/t.totalspace)*100,2) as "% Free" from
(select round(sum(d.bytes)/(1024*1024)) as totalspace, d.tablespace_name tablespace
from dba_data_files d group by d.tablespace_name) t,
(select round(sum(f.bytes)/(1024*1024)) as freespace, f.tablespace_name tablespace
from dba_free_space f group by f.tablespace_name) fs where t.tablespace=fs.tablespace
and t.tablespace like '<table_space>' order by t.tablespace;
```

```
select T.NAME , sum(BYTES)/(1024*3) from V$TABLESPACE T, V$DATAFILE D
where T.TS#=D.TS#
and T.NAME in (<table space>) group by T.NAME;
```

Views:

https://docs.oracle.com/cd/E18283_01/server.112/e16508/schemaob.htm

```
CREATE VIEW staff AS
  SELECT employee_id, last_name, job_id, manager_id, department_id
  FROM   employees;
```

- Views are logical representation of one or more tables.
- Unlike table, view is not allocated space because views are based on other db objects (tables)
  Materialized views replicate data at distributed sites and synchronize updates performed at several sites

Updatable join view:
An updatable join view, also called a modifiable join view, involves two or more base tables or views and permits DML operations. An updatable view contains multiple tables in the top-level FROM clause of the SELECT statement and is not restricted by the WITH READ ONLY clause.
To be inherently updatable, a view must meet several criteria. For example, a general rule is that an INSERT, UPDATE, or DELETE operation on a join view can modify only one base table at a time.

Key Preserver:
http://www.dba-oracle.com/t_key_preserved_table.htm
An updatable join view, also called a modifiable join view, involves two or more base tables or views and permits DML operations. An updatable view contains multiple tables in the top-level FROM clause of the SELECT statement and is not restricted by the WITH READ ONLY clause.
To be inherently updatable, a view must meet several criteria. For example, a general rule is that an INSERT, UPDATE, or DELETE operation on a join view can modify only one base table at a time. The following query of the USER_UPDATABLE_COLUMNS data dictionary view shows that the view created in Example 4-7 is updatable:

```
SQL> SELECT TABLE_NAME, COLUMN_NAME, UPDATABLE
  2  FROM   USER_UPDATABLE_COLUMNS
  3  WHERE  TABLE_NAME = 'STAFF_DEPT_10_30';

TABLE_NAME                     COLUMN_NAME                    UPD
------------------------------ ------------------------------ ---
STAFF_DEPT_10_30               EMPLOYEE_ID                    YES
STAFF_DEPT_10_30               LAST_NAME                      YES
STAFF_DEPT_10_30               JOB_ID                         YES
STAFF_DEPT_10_30               DEPARTMENT_ID                  YES
```

All updatable columns of a join view must map to columns of a key-preserved table.
A key-preserved table in a join query is a table in which each row of the underlying table appears at most one time in the output of the query. In Example 4-7, department_id is the primary key of the departments table, so each row from the employees table appears at most once in the result set, making the employees table key-preserved.
The departments table is not key-preserved because each of its rows may appear many times in the result set. Update will be done on emp_id column

Important point to be note in modifiable view:
- The table columns to be updated must be listed as updatable in user_modufiable_table
- A general rule is that an INSERT, UPDATE, or DELETE operation on a join view can modify only one base table at a time
- The table to be updated must be key preserve. department_id is the primary key of the departments table, so each row from the employees table appears at most once in

the result set, making the employees table key-preserved. **The departments table is not key-preserved because each of its rows may appear many times in the result set.**

Pasted from
<https://docs.oracle.com/cd/E18283_01/server.112/e16508/schemaob.htm#i20690>

An updatable join view is a join view that involves two or more base tables or views, where UPDATE, INSERT, and DELETE operations are permitted. The data dictionary views ALL_UPDATABLE_COLUMNS, DBA_UPDATABLE_COLUMNS, and USER_UPDATABLE_COLUMNS contain information that indicates which of the view columns are updatable. In order to be inherently updatable, a view cannot contain any of the following constructs:

- A set operator
- A DISTINCT operator
- An aggregate or analytic function
- A GROUP BY, ORDER BY, CONNECT BY, or START WITH clause
- A collection expression in a SELECT list
- A subquery in a SELECT list
- Joins (with some exceptions)

Views that are not updatable can be modified using INSTEAD OF triggers.

Object View:

Just as a view is a virtual table, an object view is a virtual object table. Each row in the view is an object, which is an instance of an object type. An object type is a user-defined data type.

Materialized Views:

- Materialized views are query results that have been stored or "materialized" in advance as schema objects. The FROM clause of the query can name tables, views, and materialized views. Collectively these objects are called master tables (a replication term) or detail tables (a data warehousing term).
-
- Materialised view are used to summarize, compute, replicate, and distribute data.
- In mobile computing environments, you can use materialized views to download a data subset from central servers to mobile clients, with periodic refreshes from the central servers and propagation of updates by clients to the central servers.
- They contain actual data and consume storage space.
- They can be refreshed when the data in their master tables changes.
- They can improve performance of SQL execution when used for query rewrite operations.
- The refresh method can be incremental, known as fast refresh, or a complete refresh.

Materialized View Logs

A materialized view **log is a schema object** that records changes to a master table's data so that a materialized view defined on the master table can be refreshed incrementally.

```
CREATE MATERIALIZED VIEW cust_sales_mv
PCTFREE 0 TABLESPACE demo
STORAGE (INITIAL 8M)
PARALLEL
BUILD IMMEDIATE
REFRESH COMPLETE
ENABLE QUERY REWRITE AS
SELECT  c.cust_last_name, SUM(amount_sold) AS sum_amount_sold
FROM customers c, sales s WHERE s.cust_id = c.cust_id
GROUP BY c.cust_last_name;
```

Pasted from <https://docs.oracle.com/cd/E11882_01/server.112/e25554/basicmv.htm#DWHSG0083>

Query rewrite:
When base tables contain large amount of data, it is expensive and time-consuming to compute the required aggregates or to compute joins between these tables. In such cases, queries can take minutes or even hours. Because materialized views contain already precomputed aggregates and joins, Oracle Database employs an extremely powerful process called query rewrite to quickly answer the query using materialized views.
One of the major benefits of creating and maintaining materialized views is the ability to take advantage of query rewrite, which transforms a SQL statement expressed in terms of tables or views into a statement accessing one or more materialized views that are defined on the detail tables. The transformation is transparent to the end user or application, requiring no intervention and no reference to the materialized view in the SQL statement.

PCTFREE : database Block level parameter
PCTFREE is a block storage parameter used to specify how much space should be left in a database block for future updates. For example, for PCTFREE=10, Oracle will keep on adding new rows to a block until it is 90% full. This leaves 10% for future updates (row expansion)

PCTUSED: database Block level parameter
PCTUSED is a block storage parameter used to specify when Oracle should consider a database block to be empty enough to be added to the freelist. Oracle will only insert new rows in blocks that is enqueued on the freelist. For example, if PCTUSED=40, Oracle will not add new rows to the block unless sufficient rows are deleted from the block so that it falls below 40% used.

Pasted from <http://www.orafaq.com/wiki/PCTUSED>

With clause (temporary table):
The WITH clause, or subquery factoring clause, is part of the SQL-99 standard and was added into the Oracle SQL syntax in Oracle 9.2. **The WITH clause may be processed as an inline view or resolved as a temporary table. The advantage of the latter is that repeated references to the subquery may be more efficient as the data is easily retrieved from the temporary table, rather than being requeried by each reference.** You should assess the performance implications of the WITH clause on a case-by-case basis.
This article shows how the WITH clause can be used to reduce repetition and simplify complex SQL statements. I'm not suggesting the following queries are the best way to retrieve the required information. They merely demonstrate the use of the WITH clause. Using the SCOTT schema, for each employee we want to know how many other people are in their department. Using an inline view we might do the following.

```
SELECT e.ename AS employee_name,
     dc.dept_count AS emp_dept_count
FROM   emp e,
     (SELECT deptno, COUNT(*) AS dept_count
      FROM   emp
      GROUP BY deptno) dc
WHERE  e.deptno = dc.deptno;
```

Using a WITH clause this would look like the following.

```
WITH dept_count AS (
 SELECT deptno, COUNT(*) AS dept_count
 FROM   emp
 GROUP BY deptno)
```

```
SELECT e.ename AS employee_name,
       dc.dept_count AS emp_dept_count
FROM   emp e,
       dept_count dc
WHERE  e.deptno = dc.deptno;
```

Pasted from <https://oracle-base.com/articles/misc/with-clause>

Procedures:

A **procedure** is a group of PL/SQL statements that you can call by name. A **call specification** (sometimes called call spec) declares a Java method or a third-generation language (3GL) routine so that it can be called from SQL and PL/SQL. The call spec tells Oracle Database which Java method to invoke when a call is made. It also tells the database what type conversions to make for the arguments and return value.
Stored procedures offer advantages in the areas of **development, integrity, security, performance, and memory allocation.**

```
DECLARE
  -- Global variables
  num1 number := 95;
  num2 number := 85;
BEGIN
  dbms_output.put_line('Outer Variable num1: ' || num1);
  dbms_output.put_line('Outer Variable num2: ' || num2);
  DECLARE
    -- Local variables
    num1 number := 195;
    num2 number := 185;
  BEGIN
    dbms_output.put_line('Inner Variable num1: ' || num1);
    dbms_output.put_line('Inner Variable num2: ' || num2);
  END;
END;
/

create or replace procedure test
begin
select * from strm_owner.trade;
end;
/
```

Curser:

https://www.tutorialspoint.com/plsql/plsql_cursors.htm
Purpose of CURSOR is same as that in PL/SQL or any other programming language, to reserve an area in memory and populate it with the result set, which can then later be fetched one by one or multiple rows together, in a loop.

In oracle forms, most of the time you can avoid writing cursors as oracle forms does most of the related work on behalf of programmer.

Explicit Cursor which are construct/manage by user itself call explicit cursor.

User itself to declare the cursor,

open cursor to reserve the memory and populate data.
Opening the cursor executes the query and identifies the result set, which consists of all rows that meet the query search criteria.

For cursors declared using the **FOR UPDATE clause**, the OPEN statement also **locks those rows**

fetch the records from the active data set one at a time.
Unless you use the BULK COLLECT clause, the FETCH statement retrieves the rows in the result set one at a time. Each
Managing Cursors in PL/SQL fetch retrieves the current row and advances the cursor to the next row in the result
set.
apply logic and last close the cursor.
The CLOSE statement disables the cursor, and the result set becomes undefined. Once a
cursor is closed, you can reopen it, which runs the query again with the latest values of any cursor parameters and variables referenced in the WHERE clause. Any other operation on a closed cursor raises the predefined exception INVALID_CURSOR.

Whenever a DML statement (INSERT, UPDATE and DELETE) is issued, an implicit cursor is associated with this statement. For INSERT operations, the cursor holds the data that needs to be inserted. For UPDATE and DELETE operations, the cursor identifies the rows that would be affected.

You can not directly assign value to an explicit cursor variable you have to use expression or create subprogram for assign value to explicit cursor variable.

cursor attributes.
There are five types of cursors attributes
1- %isopen (Verify whether this cursor is open or not)
2- %found (If cursor fetch the data then %found return true)
%FOUND yields TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows, or a SELECT INTO statement returned one or more rows.

3- %notfound (If cursor fetches not data then %notfound return true)
4- %rowcount (It return no. of rows that are in cursor and also give position of record)
5- %bulk_rowcount (%bulk_rowcount is same like %rowcount but it is used in bulk)

%BULK_ROWCOUNT
Designed for use with the FORALL statement, this is a composite attribute of the implicit cursor SQL.

If a SELECT INTO statement returns more than one row, PL/SQL raises the predefined exception TOO_MANY_ROWS and sets %ROWCOUNT to 1, not the actual number of rows
that satisfy the query.
Before the first fetch, %NOTFOUND evaluates to NULL. If FETCH never executes successfully, the EXIT WHEN condition is never TRUE and the loop is never exited. To be safe, you might want to use the following EXIT statement instead:
EXIT WHEN c1%NOTFOUND OR c1%NOTFOUND IS NULL;

Fetching Bulk Data with a Cursor
The BULK COLLECT clause lets you fetch all rows from the result set at once.

```
DECLARE
TYPE IdsTab IS TABLE OF employees.employee_id%TYPE;
TYPE NameTab IS TABLE OF employees.last_name%TYPE;
```

```
ids IdsTab;
names NameTab;
CURSOR c1 IS
SELECT employee_id, last_name FROM employees WHERE job_id = 'ST_CLERK';
BEGIN
OPEN c1;
FETCH c1 BULK COLLECT INTO ids, names;
CLOsE c1;
-- Here is where you process the elements in the collections
FOR i IN ids.FIRST .. ids.LAST
LOOP
IF ids(i) > 140 THEN
DBMS_OUTPUT.PUT_LINE( ids(i) );
END IF;
END LOOP;
FOR i IN names.FIRST .. names.LAST
LOOP
IF names(i) LIKE '%Ma%' THEN
DBMS_OUTPUT.PUT_LINE( names(i) );
END IF;
END LOOP;
```

Limiting the Rows for a Bulk FETCH Operation with the LIMIT Clause

The optional LIMIT clause, allowed only in bulk FETCH statements, limits the number of rows fetched from the database. With each iteration of the loop, the FETCH statement fetches ten rows (or less) into index-by table empids. The previous values are overwritten. Note the use of empids.COUNT to determine when to exit the loop.

```
TYPE numtab IS TABLE OF NUMBER INDEX BY PLS_INTEGER;
CURSOR c1 IS SELECT employee_id FROM employees WHERE department_id = 80;
empids numtab;
rows PLS_INTEGER := 10;
BEGIN
OPEN c1;
LOOP -- the following statement fetches 10 rows or less in each iteration
FETCH c1 BULK COLLECT INTO empids LIMIT rows;
EXIT WHEN empids.COUNT = 0;
-- EXIT WHEN c1%NOTFOUND; -- incorrect, can omit some data
DBMS_OUTPUT.PUT_LINE('------- Results from Each Bulk Fetch --------');
FOR i IN 1..empids.COUNT LOOP
DBMS_OUTPUT.PUT_LINE( 'Employee Id: ' || empids(i));
END LOOP;
END LOOP;
CLOSE c1;
END;
```

Using FORALL With BULK COLLECT

```
CREATE TABLE emp_temp AS SELECT * FROM employees;
DECLARE
TYPE NumList IS TABLE OF NUMBER;
depts NumList := NumList(10,20,30);
TYPE enum_t IS TABLE OF employees.employee_id%TYPE;
TYPE dept_t IS TABLE OF employees.department_id%TYPE;
e_ids enum_t;
d_ids dept_t;
BEGIN
FORALL j IN depts.FIRST..depts.LAST
DELETE FROM emp_temp WHERE department_id = depts(j)
RETURNING employee_id, department_id BULK COLLECT INTO e_ids, d_ids;
```