**Oracle Architecture Refresher:**

[**http://www.siue.edu/~dbock/cmis565/module1-architecture.htm**](http://www.siue.edu/~dbock/cmis565/module1-architecture.htm)

**Note:**

1. Union clause does not generate unique records when datatype is char(10) and char(50).

**Foreign key exception scenarios:**

1. In a parent child table, Child table can have single column acting as foreign and primary key both.
2. Child table can have a foreign key on non-primary column (mandatory unique constraint should be there)in parent table .
3. Same table can have a primary key and another column acting as foreign key on that Primary key.
4. Table can’t be truncated if it has dependent child with foreign keys.

**Oracle constraint Disable / Enable:**

alter table

table\_name

ENABLE constraint

constraint\_name;

**Foreign key syntax:**

CREATE TABLE products

( product\_id INT PRIMARY KEY,

product\_name VARCHAR(50) NOT NULL,

category VARCHAR(25)

);

CREATE TABLE inventory

( inventory\_id INT PRIMARY KEY,

product\_id INT NOT NULL,

quantity INT,

min\_level INT,

max\_level INT,

CONSTRAINT fk\_inv\_product\_id

FOREIGN KEY (product\_id)

REFERENCES products (product\_id)

ON DELETE CASCADE

);

**What is Pragma? Types of Pragma? (**[**http://www.exforsys.com/tutorials/oracle-11g/oracle-pragma.html**](http://www.exforsys.com/tutorials/oracle-11g/oracle-pragma.html)**)**

Pragama is compiler directive.  
**Example of Autonomous transactions.**

Suppose you are updating value from table and you don't have update trigger on that table  
but still you want to maintain a log entry for this update in separate table.  
You can write a procedure and call that procedure to do this. But you cannot use "COMMIT" in this called procedure because it will save the entire transaction.  
To avoid this you can declare this procedure as autonomous transaction procedure so that the execution of this procedure will be treated as totally diff. transaction and you can issue commit in called procedure without affecting the main transaction.

Different types of Pragama are  
  
**Pragma syntax :**

CREATE OR REPLACE [FUNCTION | PROCEDURE] [NAME] IS

IS

[PRAGMA];

BEGIN

...

END;

**PRAGMA** refers to a compiler directive or "hint" it is used to provide an instruction to the compiler

1. **PRAGMA AUTONOMOUS\_TRANSACTION**: This pragma can perform an autonomous transaction within a PL/SQL block between a BEGIN and END statement without affecting the entire transaction.
2. **PRAGMA SERIALLY\_REUSABLE**: This directive tells Oracle that the package state is needed only for the duration of one call to the server. After the call is made the package may be unloaded to reclaim memory.
3. **PRAGMA EXCEPTION\_INIT**: This directive binds a user defined exception to a particular error number.
4. **PRAGMA INLINE**: (Introduced in Oracle 11g) this directive specifies that a subprogram call either is or is not to be inlined. Inlining replaces a subprogram call with a copy of the called subprogram.

**Explain Bulk Collect.**  
It is a way of fetching a very big collection of data. With the help of Oracle Bulk Collect, the PL/SQL Engine indicates the SQL Engine to collect more than one row at a single point of time and stores them into a collection. Then it switches back to the PL/SQL Engine. During the Bulk Collect, Context Switch at one point. The performance improvement would be better with the more number of rows fetched into the collection.

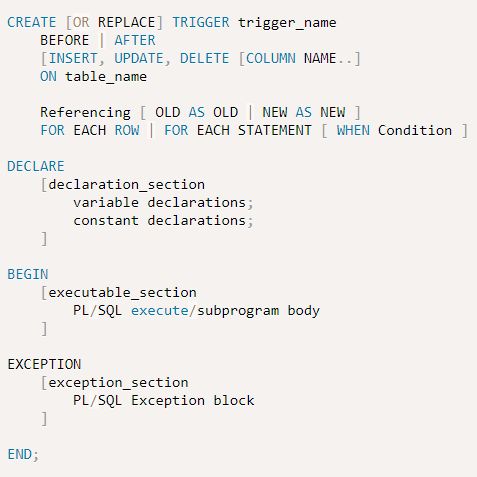
-- \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
-- create a test table for the bulk insert  
-- \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
drop table test\_objects;  
  
create table   
test\_objects   
tablespace users  
as   
select object\_name, object\_type from dba\_objects;  
  
-- \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
-- Populate the table into a array using bulk collect  
-- \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
DECLARE  
TYPE t\_tab IS TABLE OF test\_objects%ROWTYPE;  
objects\_tab t\_tab := t\_tab();  
start\_time number; end\_time number;  
BEGIN  
-- Populate a collection   
SELECT \*   
BULK COLLECT   
INTO   
objects\_tab   
FROM test\_objects;  
  
-- \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
-- Time the population of the table with a bulk insert  
-- \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
EXECUTE IMMEDIATE 'TRUNCATE TABLE test\_objects';  
Start\_time := DBMS\_UTILITY.get\_time;  
FORALL i in objects\_tab.first .. objects\_tab.last  
INSERT INTO test\_objects VALUES objects\_tab(i);  
end\_time := DBMS\_UTILITY.get\_time;  
DBMS\_OUTPUT.PUT\_LINE('Bulk Insert: '||to\_char(end\_time-start\_time));  
  
-- \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
-- Populate the table without a bulk insert  
-- \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
EXECUTE IMMEDIATE 'TRUNCATE TABLE test\_objects';  
Start\_time := DBMS\_UTILITY.get\_time;  
FOR i in objects\_tab.first .. objects\_tab.last LOOP  
INSERT INTO test\_objects (object\_name, object\_type)  
VALUES (objects\_tab(i).object\_name, objects\_tab(i).object\_type);  
END LOOP;  
end\_time := DBMS\_UTILITY.get\_time;  
DBMS\_OUTPUT.PUT\_LINE('Conventional Insert: '||to\_char(end\_time-start\_time));  
COMMIT;  
END;   
/

Use limit with bulk collect for huge number of records (Millions):

fetch c bulk collect into l\_c1, l\_c2, ....... LIMIT 1000;

Type of Triggers

1. BEFORE Trigger : BEFORE trigger execute before the triggering DML statement (INSERT, UPDATE, DELETE) execute. Triggering SQL statement is may or may not execute, depending on the BEFORE trigger conditions block.
2. AFTER Trigger : AFTER trigger execute after the triggering DML statement (INSERT, UPDATE, DELETE) executed. Triggering SQL statement is execute as soon as followed by the code of trigger before performing Database operation.
3. ROW Trigger : ROW trigger fire for each and every record which are performing INSERT, UPDATE, DELETE from the database table. If row deleting is define as trigger event, when trigger file, deletes the five rows each times from the table.
4. Statement Trigger : Statement trigger fire only once for each statement. If row deleting is define as trigger event, when trigger file, deletes the five rows at once from the table.
5. Combination Trigger : Combination trigger are combination of two trigger type,
   1. Before Statement Trigger : Trigger fire only once for each statement before the triggering DML statement.
   2. Before Row Trigger : Trigger fire for each and every record before the triggering DML statement.
   3. After Statement Trigger : Trigger fire only once for each statement after the triggering DML statement executing.
   4. After Row Trigger : Trigger fire for each and every record after the triggering DML statement executing.



**What is Mutating Table Error?**

It occurs when a Trigger tries to update a row that is currently in execution stage. So it is solved out by using temporary tables and views.

**Do Trigger can rollback?**

Yes, it will. But it is in in control of called transaction not in user or developer.

Triggers work in scope of the DML statement's transaction (either started by you explicitly or by the DML statement itself implicitly).When this transaction is rolled back, all changes made by the triggers are also rolled back. However, if you put PRAGMA autonomous\_transaction

into the trigger definition, the trigger will start its own transaction which you should commit before the trigger completes.

**Can Trigger execute DML? –** Yes

**Can Trigger call Procedure? –** Yes it can call procedure.

**Can one table have multiple triggers? –** Yes

**Cascading Triggers –** Trigger causing another triggers to execute.

**Function and DML/Rollback**

You can use DDL & DML statement in a function but remember that function cannot be used in an SQL statement. This can be called in a PL/SQL block only.

**Function and Procedure** - Yes

**Fastest way to insert a million rows in Oracle** – Select and insert in fastest

**Partitions in Oracle: (**[**https://docs.oracle.com/cd/B19306\_01/server.102/b14220/partconc.htm**](https://docs.oracle.com/cd/B19306_01/server.102/b14220/partconc.htm)**)**

* Partitioning enables data management operations such data loads, index creation and rebuilding, and backup/recovery at the partition level, rather than on the entire table
* Partitioning improves query performance. In many cases, the results of a query can be achieved by accessing a subset of partitions, rather than the entire table
* Partitioning can significantly reduce the impact of scheduled downtime for maintenance operations.

1. Range
2. Hash
3. List
4. Composite

**Partition Syntax**

CREATE TABLE sales\_range

(salesman\_id NUMBER(5),

salesman\_name VARCHAR2(30),

sales\_amount NUMBER(10),

sales\_date DATE)

**PARTITION BY RANGE(sales\_date)**

(

PARTITION sales\_jan2000 VALUES LESS THAN(TO\_DATE('02/01/2000','MM/DD/YYYY')),

PARTITION sales\_feb2000 VALUES LESS THAN(TO\_DATE('03/01/2000','MM/DD/YYYY')),

PARTITION sales\_mar2000 VALUES LESS THAN(TO\_DATE('04/01/2000','MM/DD/YYYY')),

PARTITION sales\_apr2000 VALUES LESS THAN(TO\_DATE('05/01/2000','MM/DD/YYYY'))

);

**PARTITION BY HASH(salesman\_id)**

PARTITIONS 4

STORE IN (ts1, ts2, ts3, ts4);

**PARTITION BY LIST(sales\_state)**

(

PARTITION sales\_west VALUES('California', 'Hawaii'),

PARTITION sales\_east VALUES ('New York', 'Virginia', 'Florida'),

PARTITION sales\_central VALUES('Texas', 'Illinois'),

PARTITION sales\_other VALUES(DEFAULT)

);

Composite

**PARTITION BY RANGE(sales\_date)**

SUBPARTITION BY HASH(salesman\_id)

SUBPARTITION TEMPLATE(

SUBPARTITION sp1 TABLESPACE ts1,

SUBPARTITION sp2 TABLESPACE ts2,

SUBPARTITION sp3 TABLESPACE ts3,

SUBPARTITION sp4 TABLESPACE ts4)

(PARTITION sales\_jan2000 VALUES LESS THAN(TO\_DATE('02/01/2000','MM/DD/YYYY'))

PARTITION sales\_feb2000 VALUES LESS THAN(TO\_DATE('03/01/2000','MM/DD/YYYY'))

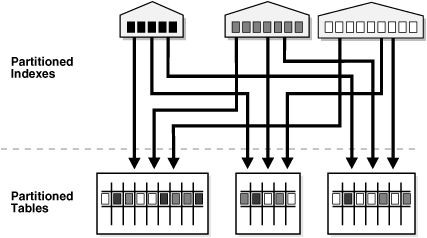
PARTITION sales\_mar2000 VALUES LESS THAN(TO\_DATE('04/01/2000','MM/DD/YYYY'))

PARTITION sales\_apr2000 VALUES LESS THAN(TO\_DATE('05/01/2000','MM/DD/YYYY'))

PARTITION sales\_may2000 VALUES LESS THAN(TO\_DATE('06/01/2000','MM/DD/YYYY'))

**Partition of indexes**

1. **Local Partitioned Indexes:** each partition of a local index is associated with exactly one partition of the table.
2. **Global Partitioned Indexes**



* 1. **Global Range Partitioned Indexes:** Global range partitioned indexes are flexible in that the degree of partitioning and the partitioning key are independent from the table's partitioning method
  2. **Global Hash Partitioned Indexes:** Global hash partitioned indexes improve performance by spreading out contention when the index is monotonically growing

**Partition Pruning**

The Oracle database server explicitly recognizes partitions and subpartitions. It then optimizes SQL statements to mark the partitions or subpartitions that need to be accessed and eliminates (prunes) unnecessary partitions or subpartitions from access by those SQL statements. In other words, partition pruning is the skipping of unnecessary index and data partitions or subpartitions in a query.

**Bitmap Indexes**

Bitmap indexes are most helpful in a data warehouse environment because they are generally great (fast) when you are only selecting data. A bitmap index is smaller than a b-tree index because it stores only the ROWID and a series of bits. In a bitmap index, if a bit is set, it means that a row in the corresponding ROWID (also stored) contains a key value.

<http://www.dba-oracle.com/oracle_tips_bitmapped_indexes.htm>

**Index-organized table (Clustered Index)**

An index-organized table (IOT) is a type of table that stores data in a B\*Tree index structure.

**Delete 100 million rows from oracle table**:

<http://www.dba-oracle.com/t_oracle_fastest_delete_from_large_table.htm>

**CTAS** (create table from select) operations in case you are trying to delete 50% of table

**Option 1**

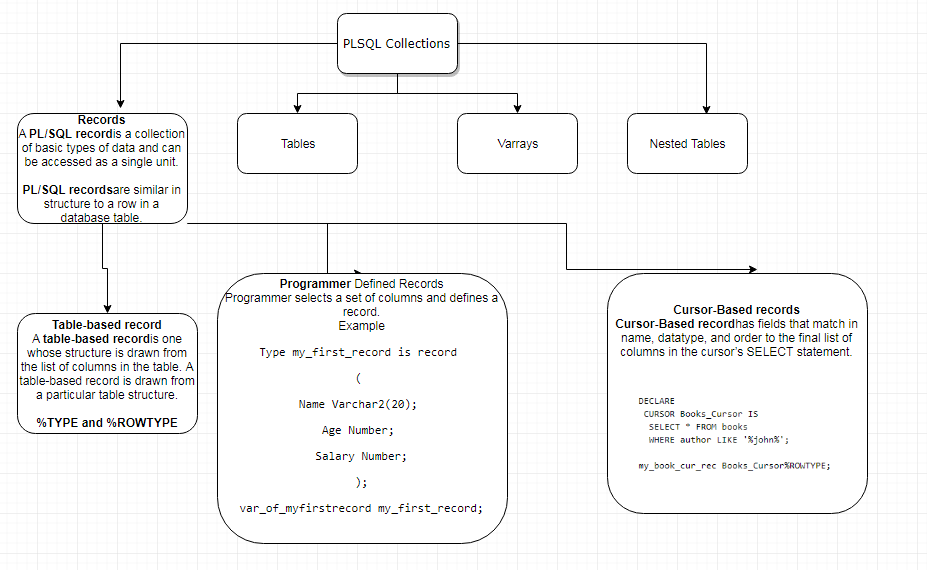
* use vanilla delete with commit; in this case issue commit with every million records so it frees up the rollback segment

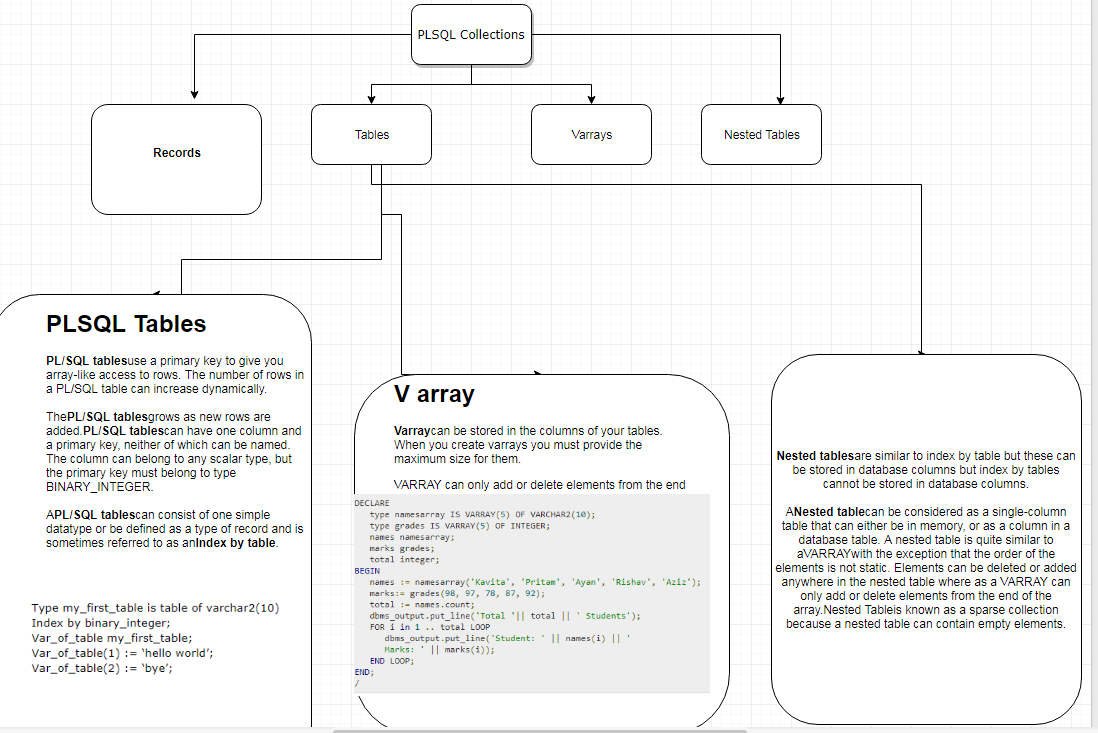
**Option 2**

* Remove index and constraints
* Create new table from old table with required data (nologgin option)
* rename tables
* readd indexes
* drop old table

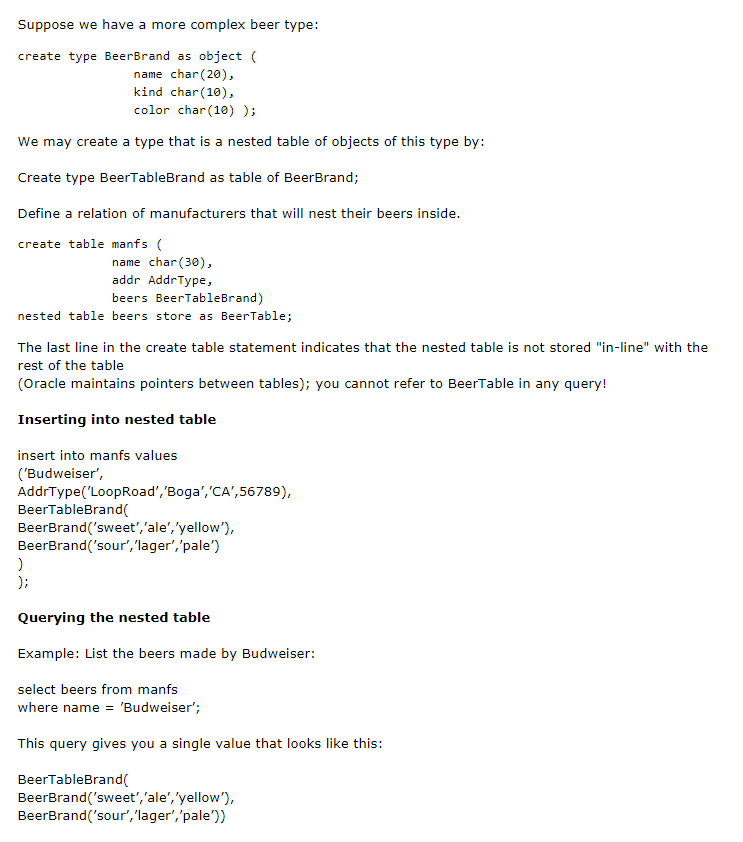
**NOLOGGING** can be used to prevent bulk operations from logging too much information to Oracle's Redo log files.

**Collections in PL SQL**





**Nested Tables:**



Index-by tables, also known as associative arrays, let you look up elements using arbitrary numbers and strings for subscript values. (They are similar to hash tables in other programming languages.)

Nested tables hold an arbitrary number of elements. They use sequential numbers as subscripts. You can define equivalent SQL types, allowing nested tables to be stored in database tables and manipulated through SQL.

Varrays (short for variable-size arrays) hold a fixed number of elements (although you can change the number of elements at runtime). They use sequential numbers as subscripts. You can define equivalent SQL types, allowing varrays to be stored in database tables. They can be stored and retrieved through SQL, but with less flexibility than nested tables.

DECLARE

TYPE nested\_type IS TABLE OF VARCHAR2(30);

TYPE varray\_type IS VARRAY(5) OF INTEGER;

TYPE assoc\_array\_num\_type

IS TABLE OF NUMBER INDEX BY PLS\_INTEGER;

TYPE assoc\_array\_str\_type

IS TABLE OF VARCHAR2(32) INDEX BY PLS\_INTEGER;

TYPE assoc\_array\_str\_type2

IS TABLE OF VARCHAR2(32) INDEX BY VARCHAR2(64);

v1 nested\_type;

v2 varray\_type;

v3 assoc\_array\_num\_type;

v4 assoc\_array\_str\_type;

v5 assoc\_array\_str\_type2;

BEGIN

-- an arbitrary number of strings can be inserted v1

v1 := nested\_type('Shipping','Sales','Finance','Payroll');

v2 := varray\_type(1, 2, 3, 4, 5); -- Up to 5 integers

v3(99) := 10; -- Just start assigning to elements

v3(7) := 100; -- Subscripts can be any integer values

v4(42) := 'Smith'; -- Just start assigning to elements

v4(54) := 'Jones'; -- Subscripts can be any integer values

v5('Canada') := 'North America';

-- Just start assigning to elements

v5('Greece') := 'Europe';

-- Subscripts can be string values

END;

/

**Cursor Attributes:**

%FOUND, %NOTFOUND, %ISOPEN, and %ROWCOUNT

Oracle Cursor can have parameters

**Example 2 of an Explicit Cursor:**  
  
DECLARE  
  
CURSOR csr\_ac (p\_name VARCHAR2) IS  
SELECT empno, name, sal  
FROM employee  
WHERE name LIKE '%p\_name%';  
  
BEGIN  
  
FOR rec\_ac IN csr\_ac ('LE')  
LOOP  
DBMS\_OUTPUT.PUT\_LINE(rec\_ac.empno || ' ' ||rec\_ac.name || ' '||v\_sal);  
END LOOP ;  
  
CLOSE csr\_ac;  
  
END;

**Types of Indexes in Oracle**

Oracle Database provides several indexing schemes, which provide complementary performance functionality. The indexes can be categorized as follows:

**B-tree indexes:** These indexes are the standard index type. They are excellent for primary key and highly-selective indexes. Used as concatenated indexes, B-tree indexes can retrieve data sorted by the indexed columns. B-tree indexes have the following subtypes:

1. **Index-organized tables:** An index-organized table differs from a heap-organized because the data is itself the index. See "Overview of Index-Organized Tables".
2. **Reverse key indexes:** In this type of index, the bytes of the index key are reversed, for example, 103 is stored as 301. The reversal of bytes spreads out inserts into the index over many blocks. See "Reverse Key Indexes".
3. **Descending indexes:** This type of index stores data on a particular column or columns in descending order. See "Ascending and Descending Indexes".
4. **B-tree cluster indexes:** This type of index is used to index a table cluster key. Instead of pointing to a row, the key points to the block that contains rows related to the cluster key. See "Overview of Indexed Clusters".

**Bitmap and bitmap join indexes:** In a bitmap index, an index entry uses a bitmap to point to multiple rows. In contrast, a B-tree index entry points to a single row. A bitmap join index is a bitmap index for the join of two or more tables. See "Bitmap Indexes".

Oracle bitmap indexes are very different from standard b-tree indexes. In bitmap structures, a two-dimensional array is created with one column for every row in the table being indexed. Each column represents a distinct value within the bitmapped index. This two-dimensional array represents each value within the index multiplied by the number of rows in the table.

At row retrieval time, Oracle decompresses the bitmap into the RAM data buffers so it can be rapidly scanned for matching values. These matching values are delivered to Oracle in the form of a Row-ID list, and these Row-ID values may directly access the required information.

The real benefit of bitmapped indexing occurs when one table includes multiple bitmapped indexes. Each individual column may have low cardinality. The creation of multiple bitmapped indexes provides a very powerful method for rapidly answering difficult SQL queries.

**Function-based indexes:** This type of index includes columns that are either transformed by a function, such as the UPPER function, or included in an expression. B-tree or bitmap indexes can be function-based. See "Function-Based Indexes".

**Application domain indexes:** This type of index is created by a user for data in an application-specific domain. The physical index need not use a traditional index structure and can be stored either in the Oracle database as tables or externally as a file. See "Application Domain Indexes".

**Oracle uses a different locking method then most other databases, Oracle locking policy consists of the following:**

Oracle locks data at the row level on modification only. There is no lock escalation to a block or table level, ever.

Oracle never locks data just to read it. There are no locks placed on rows of data by simple reads.

A writer of data does not block a reader of data.

A writer of data is blocked only when another writer of data has already locked the row it was going after.

**Steps to Information Gather During Tuning**

* Complete SQL text from V$SQLTEXT
* Structure of the tables referenced in the SQL statement, usually by describing the

Table in SQL\*Plus

* Definitions of any indexes (columns, column orderings), and whether the indexes

are unique or non-unique

* Optimizer statistics for the segments (including the number of rows each table,
* selectivity of the index columns), including the date when the segments were last

analyzed

* Definitions of any views referred to in the SQL statement
* Repeat steps two, three, and four for any tables referenced in the view definitions

found in step five

* Optimizer plan for the SQL statement (either from EXPLAIN PLAN, V$SQL\_PLAN,

or the TKPROF output)

* Any previous optimizer plans for that SQL statement

**PL SQL Tunning:**

1. Using a RESULT\_CACHE feature in function to get faster ouput from function.Its like in memory materialize view.

FUNCTION one\_employee (employee\_id\_in

IN employees.employee\_id%TYPE)

RETURN employees%ROWTYPE

RESULT\_CACHE RELIES\_ON (employees)

IS

l\_employee employees%ROWTYPE;

BEGIN

.

.

.

1. Generate extended statistics for highly skewed tables
2. Bulk Processing Feature
3. Avoid logical operators to make sure index get used
   1. <>,!=
   2. Function in where clause
   3. Implicit conversions
   4. Like ‘%a’
   5. Or operation
4. Pass parameter in Procedure by Reference.Use word NOCOPY with parameter

procedure get\_customer\_orders(  
p\_customer\_id in number,  
p\_orders out nocopy orders\_coll  
);  
theorders orders\_coll;  
get\_customer\_orders(124, theorders);

1. Proper join methods (<http://oracle-online-help.blogspot.com/2007/03/nested-loops-hash-join-and-sort-merge.html>)
   1. Nested loop Join - better for joining small result sets when column of joins are indexed
      1. In this algorithm, an outer loop is formed which consists of few entries and then for each entry, and inner loop is processed.
   2. Hash Join - larger datasets and one or more indexes are not used
      1. Hash joins are used when the joining large tables. The optimizer uses smaller of the 2 tables to build a hash table in memory and the scans the large tables and compares the hash value (of rows from large table) with this hash table to find the joined rows.
   3. Sort-Merge Join.
      1. Important point to understand is, unlike nested loop where driven (inner) table is read as many number of times as the input from outer table, in sort merge join each of the tables involved are accessed at most once. So they prove to be better than nested loop when the data set is large.

**When optimizer uses Sort merge join?**

a) When the join condition is an inequality condition (like <, <=, >=). This is because hash join cannot be used for inequality conditions and if the data set is large, nested loop is definitely not an option.

b) If sorting is anyways required due to some other attribute (other than join) like “order by”, optimizer prefers sort merge join over hash join as it is cheaper.

1. Get the cardinality of table and check the statistics
2. Optimizer mostly applies the filters first and then it applies join. So eliminate rows as early as possible in join table order
3. Understand potential memory bottlenecks
   1. Memory buffers
      1. The buffer cache – All data goes here
      2. The shared pool – Code and supporting information
      3. The PGA – Sort and other calculcations
      4. The redo log buffer – Data changes
4. Avoid unnecessary sorts in your query like group by, order by, multiple joins:
5. When writing sub-queries make use of the EXISTS operator where possible as Oracle knows that once a match has been found it can stop and avoid a full table scan (it does a SEMI JOIN).
6. While querying on a partitioned table try to use the partition key in the “WHERE” clause if possible. This will ensure partition pruning.
7. Check if the statistics for the objects used in the query are up to date. If not, use the DBMS\_STATS package to collect the same.
8. Selectivity (predicate) and Cardinality (skew) factors have a big impact on query plan. Use of Statistics and Histograms can drive the query towards a better plan.
9. Read explain plan and try to make largest restriction (filter) as the driving site for the query, followed by the next largest, this will minimize the time spent on I/O and execution in subsequent phases of the plan.
10. V$SQL\_PLAN stores the query plan

**How to Identify High-Load SQL?**

High-load SQL are poorly-performing, resource-intensive SQL statements that impact the performance of the Oracle database. High-load SQL statements can be identified by:

* Automatic Database Diagnostic Monitor
* Automatic Workload Repository
* V$SQL view
* Custom Workload
* SQL Trace

**How to generate stats in oracle?**

<http://www.dba-oracle.com/concepts/tables_optimizer_statistics.htm>

**Steps to look into Query plan:**

* The plan is such that the driving table has the best filter.
* The join order in each step means that the fewest number of rows are being

returned to the next step (that is, the join order should reflect, where possible,

going to the best not-yet-used filters).

* The join method is appropriate for the number of rows being returned. For

example, nested loop joins through indexes may not be optimal when many rows

are being returned.

* Views are used efficiently. Look at the SELECT list to see whether access to the

view is necessary.

* There are any unintentional Cartesian products (even with small tables).
* Each table is being accessed efficiently:

**What is Exchange Partition Mechanism – Oracle?**

Partition exchange feature can be useful in a number of situations. For example:

• Data cleanup activities

• Datawarehouse/table refresh

• Moving data from one schema to another schema

The idea is that you have a source table with the new data which needs to be loaded into a partitioned table and then you use the exchange partition mechanism to move data from source table into the target partitioned table without moving data. It is a data dictionary update.

Exchange partition is an operation with no actual data movement and therefore, performance improvement is significant. In addition, the biggest advantage is that, unlike delete/insert processes, the table remains available for performing queries.

Consider the predicates in the SQL statement and the number of rows in the table.

Look for suspicious activity, such as a full table scans on tables with large number

Note: The guidelines described in this section are oriented to

production SQL that will be executed frequently. Most of the

techniques that are discouraged here can legitimately be employed

in ad hoc statements or in applications run infrequently where

performance is not critical.

Developing Efficient SQL Statements

SQL Tuning Overview 11-7

of rows, which have predicates in the where clause. Determine why an index is not

used for such a selective predicate.

A full table scan does not mean inefficiency. It might be more efficient to perform a

full table scan on a small table, or to perform a full table scan to leverage a better

join method (for example, hash\_join) for the number of rows returned.

**Why you want to change job?**

I am interested in a job with more responsibility, and I am very ready for a new challenge with room for growth and opportunity for advancement.

**Why morgan?**

It is subsidiary of BNY and its one of the largest bank.I will be definitely learning new things working there with global standards and Want to be doing industry-leading work..

**Optimize a Query:**

-Correct use of where filter

-removing unnecessary tables

-Remove calculated fields from join

-Use temp tables when you are joining large tables

-Using exists when you want to just check existence

-Correct use of IN and JOIN clause

-Usage of functions in Where Clause

-Usage of different data type values in WHERE clause comparisons

-Correlated subqueries removal

- Try to avoid deadlocks

select name,

LENGTH(REGEXP\_REPLACE(name,'[a-zA-Z]','')) as num\_count,

LENGTH(REGEXP\_REPLACE(name,'[0-9]','')) as char\_count,

from test6;

**Oracle (the CBO)** still uses the 20% rule in that if the CBO thinks more than 20% of the data will be accessed that it is more efficient to use a full table scan utilizing multi-block read-ahead technology.

**cardinality** refers to the uniqueness of data values contained in a particular column (attribute) of a database table.

It is import for the cardinality estimates to be as accurate as possible as they influence all aspects of the execution plan from the access method, to the join order. However, several factors can lead to incorrect cardinality estimates even when the basic table and column statistics are up to date. Some of these factors include: » Data skew » Multiple single column predicates on a single table » Function wrapped columns in the WHERE clause predicates » Complex expressions

**GATHER\_PLAN\_STATISTICS**

you can use the GATHER\_PLAN\_STATISTICS hint in the SQL statement to automatically collect more comprehensive runtime statistics. This hint records the actual cardinality (the number of rows returned) for each operation as the statement executes.

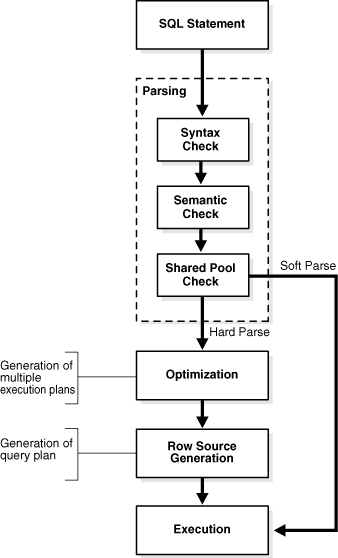
**High-cardinality** refers to columns with values that are very uncommon or unique. i.e. User ID

**Normal-cardinality** refers to columns with values that are somewhat uncommon i.e. Last NAme

**Low-cardinality** refers to columns with few unique values i.e. Code or account types

Query processing can be divided into 7 phases:

|  |  |
| --- | --- |
| [1] Syntactic | Checks the syntax of the query |
| [2] Semantic | Checks that all objects exist and are accessible |
| [3] View Merging | Rewrites query as join on base tables as opposed to using views |
| [4] Statement      Transformation | Rewrites query transforming some complex constructs into simpler ones where appropriate (e.g. subquery merging, in/or transformation) |
| [5] Optimization | Determines the optimal access path for the query to take. With the Rule Based Optimizer (RBO) it uses a set of heuristics to determine access path. With the Cost Based Optimizer (CBO) we use statistics to analyze the relative costs of accessing objects. |
| [6] QEP Generation | QEP = Query Evaluation Plan |
| [7] QEP Execution | QEP = Query Evaluation Plan |



**Can you please explain the concept of a table high water mark (HWM)?  I understand that the data is always below the high water mark.  Can you show a script to display the high water mark for a table?**

The high water mark (HWM) for an Oracle table is a construct that shows the table at its greatest size.  Just as a lake has a high-water mark after a draught, an Oracle table has a high water mark that shows the greatest size of the table, the point at which it consumed the most extents.

Also see this great script to [display all high water marks.](http://www.dba-oracle.com/t_display_high_water_marks_hwm.htm)

As a table undergoes deletes and updates, rows shrink and table data blocks become empty.  For performance reasons, Oracle keeps the high water mark for a table rather than re-calculate the high water mark after blocks at the "end" of the table (the last extent) becomes empty.

For example assume that you have a million row table that takes 30 seconds to read.  After deleting 900,000 rows, a full scan on the table will still take 30 seconds.  This is because the table high water mark is not re-set after delete operations.

The issue with the high water mark is that full-table scans will always read up to the high water mark, even thought Oracle may be reading through many empty blocks that were allocated to the table, used for rows, and then deleted.

**Explain Skewed Data and column:**

Skewed columns are columns in which the data is not evenly distributed among the rows.

For example, suppose:

* You have a table order\_lines with 100,000,000 rows
* The table has a column named customer\_id
* You have 1,000,000 distinct customers

Some (very large) customers can have hundreds of thousands or millions of order lines.

In the above example, the data in order\_lines.customer\_id is skewed. On average, you'd expect each distinct customer\_id to have 100 order lines (100 million rows divided by 1 million distinct customers). But some large customers have many, many more than 100 order lines.

This hurts performance because Oracle bases it's execution plan on statistics. So, statistically speaking, Oracle thinks it can access order\_lines based on a non-unique index on customer\_id and get only 100 records back, which it might then join to another table or whatever using a NESTED LOOP operation.

But, then when it actually gets 1,000,000 order lines for a particular customer, the index access and nested loop join are hideously slow. It would have been far better for Oracle to do a full table scan and hash join to the other table.

So, when there is skewed data, the optimal access plan depends on which particular customer you are selecting!

Oracle lets you avoid this problem by optionally gathering "histograms" on columns, so Oracle knows which values have lots of rows and which have only a few. That gives the Oracle optimizer the information it needs to generate the best plan in most cases.

Significant Query Expansions by Query Transformations:

1. **OR Expansion**: In OR expansion, the optimizer transforms a query with a WHERE clause containing OR operators into a query that uses the UNION ALL operator.
2. **View Merging:** the optimizer merges the query block representing a view into the query block that contains it.
3. **Predicate Pushing:** The optimizer "pushes" the relevant predicates from the containing query block into the view query block.
4. **Subquery Unnesting: T**he optimizer transforms a nested query into an equivalent join statement.

**PLAN\_TABLE**

PLAN\_TABLE is automatically created as a global temporary table to hold the output of an EXPLAIN PLAN statement for all users.

**What are temporal data types in Oracle?**

Oracle provides following temporal data types:

Date Data Type – Different formats of Dates

TimeStamp Data Type – Different formats of Time Stamp

Interval Data Type – Interval between dates and time

**Can we store pictures in the database and if so, how it can be done?**

Yes, we can store pictures in the database by Long Raw Data type. This datatype is used to store binary data for 2 gigabytes of length. But the table can have only on Long Raw data type.

**NVL and NVL2:**

NVL(salary,0)

NVL2(salary,what if not null,what if null)

**CASE,DECODE and COALESCE:**

CASE <colname> WHEN THEN WHEN THEN END

CASE WHEN Expression THEN result WHEN THEN END

DECODE(Columnname,condition,value,condtion,value,else part)

The COALESCE function in SQL returns the first non-NULL expression among its arguments

**NULLIF**: Return null if arguments are same else first argument

NULLIF(‘kamesh’,’kamesh’)

**LNNVL**: returns inverted Boolean value

LNNVL(commission\_pct >= .2) “If this evaluates true for all records then it will mark them as FALSE”

**NANVL**: function lets you substitute a value for a floating point number such as BINARY\_FLOAT or BINARY\_DOUBLE

**Equi Join**: which has only = condition

**Natural Join**: which ha ssame column name

**Full outer join**: All records with non matching as NULLs

**Show the cursor attributes of PL/SQL.**

%ISOPEN : Checks if the cursor is open or not

%ROWCOUNT : The number of rows that are updated, deleted or fetched.

%FOUND : Checks if the cursor has fetched any row. It is true if rows are fetched

%NOT FOUND : Checks if the cursor has fetched any row. It is True if rows are not fetched.

**First Normal Form (1NF):** This should remove all the duplicate columns from the table. Creation of tables for the related data and identification of unique columns.

**Second Normal Form (2NF):** Meeting all requirements of the first normal form. Placing the subsets of data in separate tables and Creation of relationships between the tables using primary keys.

**Third Normal Form (3NF):** This should meet all requirements of 2NF. Removing the columns which are not dependent on primary key constraints.

**Optimize a Query:**

-Correct use of where filter

-removing unnecessary tables

-Remove calculated fields from join

-Use temp tables when you are joining large tables

-Using exists when you want to just check existence

-Correct use of IN and JOIN clause

-Usage of functions in Where Clause

-Usage of different data type values in WHERE clause comparisons

-Correlated subqueries removal

- Try to avoid deadlocks

What is Junk Dimensions

use indicator field as separate dimension and later use that junk id in fact

Transformations of informatica:

1. How do you use joiner?

2. how do you use Filter Transformation?

3. 2 Different source ho you join data with non equi condition

4. Unconnected and connected filter look up

5. is filter active or passive?

6. What is router?

7. HWM and how it works?

8. do truncate drop triggers and indexses of tables.

There are 3 Informatica transformations (External Procedure, Lookup, and Stored Procedure) that can be unconnected in a valid mapping.

Query optimzation tecniques from explain plan:

 - Minimize throw away rows..if you r end results has only 1% of the rows which are accessed then probably you can optimize this at source level.

 V$SQL\_PLAN views to display the execution plan of a SQL statement.

 Plan Stability: Plan stability preserves execution plans in stored outlines. An outline is implemented as a set of optimizer hints that are associated with the SQL statement. If the use of the outline is enabled for the statement, then Oracle Database automatically considers the stored hints and tries to generate an execution plan in accordance with those hints.

 The V$SQL\_PLAN\_STATISTICS view provides the actual execution statistics for every operation in the plan, such as the number of output rows and elapsed time.

 While a PLAN\_TABLE table is automatically set up for each user, you can use the SQL script utlxplan.sql to manually create a local PLAN\_TABLE in your schema.

 With multiple statements, you can specify a statement identifier and use that to identify your specific execution plan. Before using SET STATEMENT ID, remove any existing rows for that statement ID.

 Example 12-3 Using EXPLAIN PLAN with the STATEMENT ID Clause

EXPLAIN PLAN

  SET STATEMENT\_ID = 'st1' FOR

SELECT last\_name FROM employees;

DBMS\_XPLAN.DISPLAY table function:This function accepts options for displaying the plan table output.

The DBMS\_STATS package also provides procedures for managing statistics

GATHER\_INDEX\_STATS

GATHER\_TABLE\_STATS

GATHER\_SCHEMA\_STATS

Capturing SQL Plan Baselines

During the SQL plan baseline capture phase, Oracle Database records information about SQL statement execution to detect plan changes and decide whether it is safe to use new plans. To do so, the database maintains a history of plans for individual SQL statements. Because ad-hoc SQL statements do not repeat and thus do not suffer performance degradation, plan history is maintained only for repeatable SQL statements.

A result cache is an area of memory, either in the Shared Global Area (SGA) or client application memory, that stores the results of a database query or query block for reuse. The cached rows are shared across SQL statements and sessions unless they become stale.

Parallel hints:

select /\*+ FULL(emp) PARALLEL(emp, 35) \*/

|  |  |
| --- | --- |
| LONG | Character data of variable length up to 2 gigabytes, or 231 -1 bytes. |
| LONG RAW | Raw binary data of variable length up to 2 gigabytes. |
| CLOB | A character large object containing single-byte or multibyte characters. Both fixed-width and variable-width character sets are supported, both using the database character set. Maximum size is (4 gigabytes - 1) \* (database block size).  CLOB intended to retain character-based data. |
| NCLOB | A character large object containing Unicode characters. Both fixed-width and variable-width character sets are supported, both using the database national character set. Maximum size is (4 gigabytes - 1) \* (database block size). Stores national character set data. |
| BLOB | A binary large object. Maximum size is (4 gigabytes - 1) \* (database block size).  BLOB primarily intended to hold non-traditional data, such as images,videos,voice or mixed media. |
| BFILE | Contains a locator to a large binary file stored outside the database. Enables byte stream I/O access to external LOBs residing on the database server. Maximum size is 4 gigabytes.  The BFILE datatype is used to store unstructured binary data outside the database. The column of BFILE type stores file locator that points the OS file which actually stores data. |

NCLOB and NBLOB can hold national character set where as clob and blob hold database character set

Varchar and nvarchar

**Varchar** is Non-Unicode Variable Length character data type.   
**NVarchar** is UNicode Variable Length character data type. It can store both non-Unicode and Unicode characters.

**Varchar** takes 1 byte per character  
**NVarchar** takes 2 bytes per Unicode/Non-Unicode character.

**Package Body and Specifications:**

Package have two main types of procedures, functions:   
1. Public   
2. Private   
  
1. Public - exist in body and package specification   
2. Private - exists only in body

<http://dba-blog.blogspot.in/2005/08/using-of-bulk-collect-and-forall-for.html>

<https://stackoverflow.com/questions/6042568/difference-between-strong-and-weak-ref-cursor-in-oracle>

REF Cursor: A ref cursor in Oracle PL/SQL is much like an ordinary PL/SQL cursor in that it acts as a pointer to the result set of the cursor with which it is associated. However, the difference is that a ref cursor can be assigned to different result sets whereas a cursor is always associated with the same result set.

Ref cursors also come in two variants: strongly typed and weakly typed: Weak ref cursor types can be associated with any query whereas strong ref cursor types can only be associated with cursors of the same type.

Weakly Type Ref Cursor:

DECLARE  
 TYPE wkrefcurty IS REF CURSOR;  
        -- weak ref cursor type  
 my\_cur wkrefcurty;  
 dept departments%ROWTYPE;  
 BEGIN  
  OPEN my\_cur FOR SELECT \* FROM departments;  
  FETCH my\_cur INTO dept;  
  CLOSE my\_cur;  
 END;

Strongly Type Ref Cursor:

With a strong PL/SQL ref cursor type, errors like this are reported by the Oracle PL/SQL compiler at compile time, as in the following example.  
  
DECLARE  
 TYPE myrefcurty IS REF CURSOR RETURN employees%ROWTYPE; -- strong ref cursor type  
 my\_cur myrefcurty;  
 emp employees%ROWTYPE;  
BEGIN  
 OPEN my\_cur FOR SELECT \* FROM employees; -- can't do this  
 FETCH my\_cur INTO emp;  
 CLOSE my\_cur;  
END;  
/

CREATE OR REPLACE PACKAGE ref\_cursor\_demo IS  
 TYPE wkrefcurty IS REF CURSOR;   
     -- weakly-typed ref cursor  
 PROCEDURE open\_cursor (the\_cursor OUT wkrefcurty);  
 PROCEDURE fetch\_cursor (the\_cursor IN OUT wkrefcurty);  
END ref\_cursor\_demo;  
  
  
As mentioned earlier, we can open a ref cursor in one Oracle PL/SQL procedure and fetch from it in another as in the following example.

CREATE OR REPLACE PACKAGE BODY   
ref\_cursor\_demo IS  
  
 PROCEDURE open\_cursor (the\_cursor OUT wkrefcurty) IS   
 BEGIN  
  OPEN the\_cursor FOR   
   SELECT \* FROM departments  
   ORDER BY department\_name DESC;  
END open\_cursor;  
  
 PROCEDURE fetch\_cursor (the\_cursor IN OUT wkrefcurty) IS   
  dept departments%ROWTYPE;  
BEGIN  
 FETCH the\_cursor INTO dept;  
 dbms\_output.put\_line('1st department is '||dept.department\_name);  
END fetch\_cursor;  
END ref\_cursor\_demo;  
  
DECLARE  
 my\_cur ref\_cursor\_demo.wkrefcurty;  
BEGIN  
 ref\_cursor\_demo.open\_cursor(my\_cur);  
 ref\_cursor\_demo.fetch\_cursor(my\_cur);  
END;  
/

Why Explicit Cursor:

Iterate through complex query logic

To do Bulk Collect we use

Execute multiple DML with each iteration of explicit cursor

**Example of Implicit Cursor FOR LOOP**

BEGIN

FOR vItems IN (

SELECT last\_name

FROM employees

WHERE manager\_id > 120

ORDER BY last\_name

)

LOOP

DBMS\_OUTPUT.PUT\_LINE ('Name = ' || vItems.last\_name);

END LOOP;

END;

/

**Example of Explicit Cursor FOR LOOP**

DECLARE

CURSOR c1 IS

SELECT last\_name

FROM employees

WHERE manager\_id > 120

ORDER BY last\_name;

BEGIN

FOR vItems IN c1 LOOP

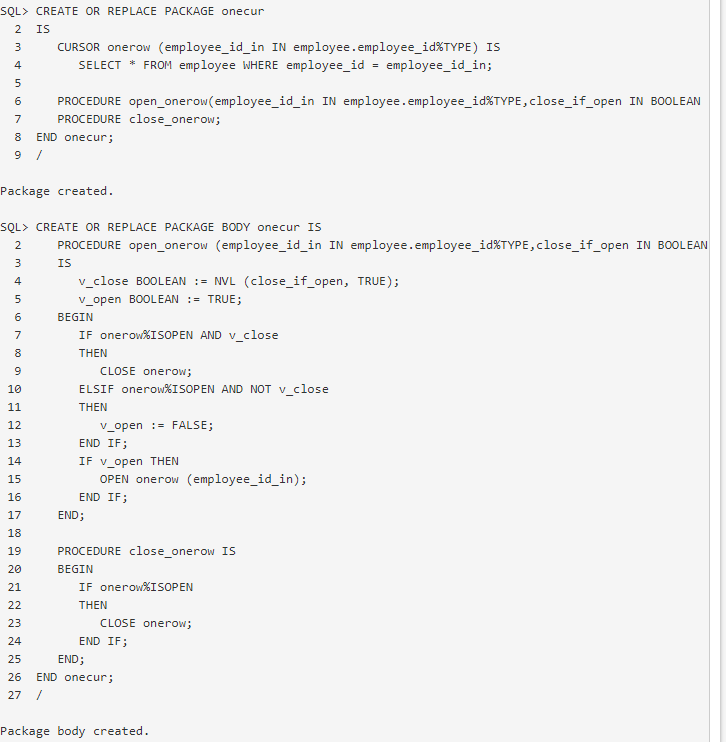
DBMS\_OUTPUT.PUT\_LINE ('Name = ' || vItems.last\_name);

END LOOP;

END;

/

Package Example:



Difference between REF CURSOR and REGULAR CURSOR

1) A ref cursor can not be used in CURSOR FOR LOOP, it must be used in simple CURSOR LOOP statement as in example.

2) A ref cursor is defined at runtime and can be opened dynamically but a regular cursor is static and defined at compile time.

3) A ref cursor can be passed to another PL/SQL routine (function or procedure) or returned to a client. A regular cursor cannot be returned to a client application and must be consumed within same routine.

4) A ref cursor incurs a parsing penalty because it cannot cached but regular cursor will be cached by PL/SQL which can lead to a significant reduction in CPU utilization.

5) A regular cursor can be defined outside of a procedure or a function as a global package variable. A ref cursor cannot be; it must be local in scope to a block of PL/SQL code.

6) A regular cursor can more efficiently retrieve data than ref cursor. A regular cursor can implicitly fetch 100 rows at a time if used with CURSOR FOR LOOP. A ref cursor must use explicit array fetching.

Use of ref cursors should be limited to only when you have a requirement of returning result sets to clients and when there is NO other efficient/effective means of achieving the goal.

Oracle Explicit Cursor- For loop-Single column fetch

DECLARE

CURSOR c1 IS SELECT ename FROM emp;

r1 c1%ROWTYPE;

BEGIN

FOR r1 IN C1

LOOP

dbms\_output.put\_line(r1.ename);

END LOOP r1;

END;

/

Oracle Explicit Cursor- For loop -Multi column fetch

DECLARE

CURSOR c1 IS SELECT ename,sal FROM emp;

r1 c1%ROWTYPE;

BEGIN

FOR r1 IN C1

LOOP

dbms\_output.put\_line(r1.ename||','||r1.sal);

END LOOP r1;

END;

/

Oracle Explicit Cursor- without For loop-Single column fetch

Oracle Explicit Cursor- without For loop-Multi column fetch

Oracle Ref Cursor- Single column fetch

Oracle Ref Cursor- Multi column fetch

<https://stackoverflow.com/questions/3187850/how-does-a-recursive-cte-run-line-by-line>