COLLECTIONS:

The syntax for defining a nested table in SQL is like this:

CREATE OR REPLACE TYPE nestedtable\_type AS TABLE OF element\_type;

The syntax for defining a nested table in PL/SQL is like this:

TYPE nestedtable\_type IS TABLE OF element\_type;

After it has been defined in SQL the nested table can be used as a column in the table definition, just like you would use one of the simple types.

CREATE TABLE table\_name

(

field1 [VARCHAR2|NUMBER|DATE]

, field2 nestedtable\_type

)

NESTED TABLE field2 STORE AS storage\_name;

| **Index-By Table** | **Nested Table** | **VARRAY** |
| --- | --- | --- |
| Dimensionality | Single | Single | Single |
| Usable in SQL? | No | Yes | Yes |
| Usable as column datatype in a table? | No | Yes; data stored "out of line" (in separate table) | Yes; data stored "in line" (in same table) |
| Uninitialized state | Empty (cannot be null); elements undefined | Atomically null; illegal to reference elements | Atomically null; illegal to reference elements |
| Initialization | Automatic, when declared | Via constructor, fetch, assignment | Via constructor, fetch, assignment |
| In PL/SQL, elements referenced via | BINARY\_INTEGER  (-2,147,483,647 .. 2,147,483,647) | Positive integer between 1 and 2,147,483,647 | Positive integer between 1 and 2,147,483,647 |
| Sparse? | Yes | Initially, no; after deletions, yes | No |
| Bounded? | No | Can be extended | Yes |
| Can assign value to any element at any time? | Yes | No; may need to EXTEND first | No; may need to EXTEND first, and cannot EXTEND past upper bound |
| Means of extending | Assign value to element with a new subscript | Use built-in EXTEND procedure (or TRIM to condense), with no predefined maximum | EXTEND (or TRIM), but only up to declared maximum size |
| Can be compared for equality? | No | No | No |
| Retains ordering and subscripts when stored in and retrieved from database? | N/A | No | Yes |

**Use index by tables when:**

* Your program needs small lookups
* The collection can be made at runtime in the memory when the package/ procedure is initialized
* The data volume is unknown beforehand
* The subscript values are flexible (e.g. strings, negative numbers, non-sequential)
* You do not need to store the collection in the database

**Use nested tables when:**

* The data needs to be stored in the database
* The number of elements in the collection is not known in advance
* The elements of the collection may need to be retrieved out of sequence
* Updates and deletions affect only some elements, at arbitrary locations
* Your program does not expect to rely on the subscript remaining stable, as their order may change when nested tables are stored in the database.

**Use varrays when:**

* The data needs to be stored in the database
* The number of elements of the varray is known in advance
* The data from the varray is accessed in sequence
* Updates and deletions happen on the varray as a whole and not on arbitrarily located elements in the varray

A Table is a basic unit of storage in oracle.  
  
A nested table is a collection type. The main advantage of collections is instead of processing data sequentially, we may process all the date  in one step.  
- It is a collection of rows stored as a column in a table.   
- It is a table withing a table.  
- There is no restriction on number of rows in a nested table  
- There is no restriction on number of columns in a nested table.  
- We can not index nested table.  
- The nested table is stored as a seperate table and main table maintains a pointer to the nested table as reference.  
- Nested table stored data in no particular order   
- But when you retrieve data into pl/sql variable it assigns serial number starts with 1  
- Intially they are dense you may sparse later.  
- Nested tables are unbounded. There is no upper limit.

Nested table Unbounded   
Individual element can be deleted   
Stored out of line in stored table.   
Support indexes.   
More flexible.   
Data is stored in separate table.   
Stored outside the table   
DML are allowed.   
  
VArrays  
Bounded  
Individual element cannot be deleted.  
Stored by Oracle in- line  
Do not support indexes.  
Not flexible than nested table.  
Data is stored as single object in database.  
Stored within table.  
DML are not allowed

Bulk binds can improve the performance when loading collections from a query. The BULK COLLECT INTO construct binds the output of the query to the collection.SELECT statements that retrieve multiple rows with a single fetch, improving the speed of data retrieval

The FORALL syntax allows us to bind the contents of a collection to a single DML statement, allowing the DML to be run for each row in the collection without requiring a context switch each time. INSERTs, UPDATEs, and DELETEs that use collections to change multiple rows of data very quickly.

When the PL/SQL runtime engine processes a block of code, it executes the procedural statements within its own engine, but passes the SQL statements on to the SQL engine. The SQL layer executes the SQL statements and then returns information to the PL/SQL engine.

This transfer of control between the PL/SQL and SQL engines is called a context switch. Each time a switch occurs, there is additional overhead. There are a number of scenarios in which many switches occur and performance degrades.

With FORALL and BULK COLLECT, however, you can fine-tune the way these two engines communicate, effectively telling the PL/SQL engine to compress multiple context switches into a single switch, thereby improving the performance of your applications.

Using BULK COLLECT, we replace "INTO" with "BULK COLLECT INTO" (whether it is SELECT INTO, FETCH INTO or EXECUTE IMMEDIATE INTO) and then after the INTO keyword, you will supply one or more collections, rather than a record or list of variables.

LOOP

FETCH c BULK COLLECT INTO l\_Data LIMIT 100;

FORALL i IN 1..l\_Data.COUNT

INSERT INTO Test\_Object VALUES l\_Data(i);

EXIT WHEN c%NOTFOUND;

END LOOP;

CLOSE c;

All exceptions raised during execution are saved in %BULK\_EXCEPTION attribute. It also stores a collection of records similar to BULK COLLECT.

%BULK\_EXCEPTIONS(i).ERROR\_CODE holds the corresponding Oracle error code.

%BULK\_EXCEPTIONS(i).ERROR\_INDEX holds the iteration number of the FORALL statement.

%BULK\_EXCEPTIONS.COUNT holds total number of exceptions encountered.

In order to bulk collect exceptions, we have to use FORALL clause with SAVE EXCEPTIONS keyword

eXAMPLE:

FORALL i IN v\_test\_ID.FIRST..v\_test\_ID.LAST SAVE EXCEPTIONS

UPDATE TEST

SET NEW\_DESC = TO\_CHAR(TEST\_ID)||TEST\_DESC

WHERE TEST\_ID = v\_test\_ID(i);

Commit;

EXCEPTION

WHEN OTHERS THEN

v\_Err\_Count := SQL%BULK\_EXCEPTIONS.COUNT;

DBMS\_OUTPUT.PUT\_LINE('Number of statements that failed: ' || v\_Err\_Count);

FOR i IN 1..v\_Err\_Count

LOOP

DBMS\_OUTPUT.PUT\_LINE('Error #' || i || ' occurred during '||'Iteration #' || SQL%BULK\_EXCEPTIONS(i).ERROR\_INDEX);

DBMS\_OUTPUT.PUT\_LINE('Error message is ' ||

SQLERRM(-SQL%BULK\_EXCEPTIONS(i).ERROR\_CODE));

You can also use Bulk fetch from Native Dynamic SQL statements. In the following example, we will fetch a collection of keys from a table based on a different set of criteria passed in as a parameter (Parameter in this case will be represented by a sqlplus variable).

EXAMPLE:

DECLARE

TYPE V\_Dynamic\_Collection IS TABLE OF User\_Objects%ROWTYPE

INDEX BY PLS\_INTEGER;

V\_Rec V\_Dynamic\_Collection;

V\_clause VARCHAR2(100) := :Where\_clause;

BEGIN

/\* Execute the statement and bulk fetch the results.\*/

EXECUTE IMMEDIATE 'SELECT \*

FROM User\_Objects

WHERE ' || V\_clause

BULK COLLECT INTO V\_Rec;

/\* Check the Fetched Records\*/

DBMS\_OUTPUT.PUT\_LINE (V\_Rec.COUNT ||

' Records Fetched from USER\_OBJECTS' );

END;

Mutating error normally occurs when we are performing some DML operations and we are trying to select the affected record from the same trigger. So basically we are trying to select records in the trigger from the table that owns the trigger. This creates inconsistency and Oracle throws a mutating error.

Now, Let's create a Mutating Trigger (Row Level) on Update of Status column of above table.

**SQL> CREATE OR REPLACE TRIGGER MutatingTrigger**  **/\*Row Level TRIGGER\*/**

2 AFTER UPDATE OF STATUS ON TEST

3 FOR EACH ROW

4 **DECLARE**

5 V\_count NUMBER;

6 **BEGIN**

7 Select count(\*) Into V\_count From TEST

8 Where STATUS='Active';

9 DBMS\_OUTPUT.PUT\_LINE('Total Number of Active Records: '|| V\_count);

10 **END;**

**Trigger created.**

Now if we try to change status of any record to "Active", Oracle will throw a **Mutating Table Error**as we are trying to update the records and trigger is trying to select affected records in the same trigger.

EXTERNAL TABLE:

CREATE TABLE EMPLOYEES\_EXT

2 (

3 EMP\_NO NUMBER,

4 ENAME VARCHAR2(30),

5 DEPTNO NUMBER,

6 HIREDATE DATE,

7 SALARY NUMBER (8,2)

8 )

9 ORGANIZATION EXTERNAL

10 (

11 TYPE ORACLE\_LOADER

12 DEFAULT Directory EXTTABDIR

13 Access Parameters

14 (

15 Records Delimited By Newline

16 Fields Terminated By ','

17 )

18 Location('EMPLOYEES\_DATA.csv')

19 )

20 REJECT LIMIT 10;

Table created.

**DEFAULT Directory EXTTABDIR:**The Directory where the File resides.

**RECORDS DELIMITED BY NEWLINE:**The New Line Character.

**FIELDS TERMINATED BY ',':**The Column termination character.

**LOCATION('EMPLOYEES\_DATA.csv'):**The name of the External File.

Following is procedure to load the data from Third Party Database into Oracle using SQL Loader.

1. Convert the Data into Flat file using third party database command.
2. Create the Table Structure in Oracle Database using appropriate datatypes
3. Write a Control File, describing how to interpret the flat file and options to load the data.
4. Execute SQL Loader utility specifying the control file in the command line argument

To understand it better let us see the following case study.

**CASE STUDY (Loading Data from MS-ACCESS to Oracle)**

Suppose you have a table in MS-ACCESS by name EMP, running under Windows O/S, with the following structure

EMPNO INTEGER

NAME TEXT(50)

SAL CURRENCY

JDATE DATE

This table contains some 10,000 rows. Now you want to load the data from this table into an Oracle Table. Oracle Database is running in LINUX O/S.

**Solution**

**Step 1**

Start MS-Access and convert the table into comma delimited flat (popularly known as csv) , by clicking on File/Save As menu. Let the delimited file name be emp.csv  
  
Now transfer this file to Linux Server using FTP command

1. Go to Command Prompt in windows
2. At the command prompt type FTP followed by IP address of the server running Oracle.   
     
   FTP will then prompt you for username and password to connect to the Linux Server. Supply a valid username and password of Oracle User in Linux  
     
   For example:-
3. C:\> ftp 200.200.100.111   
   Name: oracle

Password:oracle  
FTP>

1. Now give PUT command to transfer file from current Windows machine to Linux machine
2. FTP>put  
   Local file:C:\>emp.csv  
   remote-file:/u01/oracle/emp.csv
3. File transferred in 0.29 Seconds

FTP>

1. Now after the file is transferred quit the FTP utility by typing bye command.

FTP>bye  
Good-Bye

**Step 2**

Now come to the Linux Machine and create a table in Oracle with the same structure as in MS-ACCESS by taking appropriate datatypes. For example,  create a table like this

$ sqlplus scott/tiger

SQL> CREATE TABLE emp (empno number(5),

name varchar2(50),

sal  number(10,2),

jdate date);

**Step 3**

After creating the table, you have to write a control file describing the actions which SQL Loader should do. You can use any text editor to write the control file. Now let us write a controlfile for our case study

$ vi emp.ctl

1 LOAD DATA

2 INFILE ‘/u01/oracle/emp.csv’  
3 BADFILE ‘/u01/oracle/emp.bad’  
4 DISCARDFILE ‘/u01/oracle/emp.dsc’  
5 INSERT INTO TABLE emp   
6 FIELDS TERMINATED BY “,” OPTIONALLY ENCLOSED BY ‘”’ TRAILING NULLCOLS  
7 (empno,name,sal,jdate date ‘mm/dd/yyyy’)

Notes: (Do not write the line numbers, they are meant for explanation purpose)

1.       The LOAD DATA statement is required at the beginning of the control file.  
2.       The INFILE option specifies where the input file is located   
3.       Specifying BADFILE is optional. If you specify,  then bad records found during loading will be stored in this file.  
4.       Specifying DISCARDFILE is optional. If you specify, then records which do not meet a WHEN condition will be written to this file.   
5.       You can use any of the following loading option  
    i.       INSERT : Loads rows only if the target table is empty  
    ii.       APPEND: Load rows if the target table is empty or not.  
    iii.      REPLACE: First deletes all the rows in the existing table and then, load rows.  
    iv.      TRUNCATE: First truncates the table and then load rows.  
6.       This line indicates how the fields are separated in input file. Since in our case the fields are separated by “,” so we have specified “,” as the terminating char for fields. You can replace this by any char which is used to terminate fields. Some of the popularly use terminating characters are semicolon “;”, colon “:”, pipe “|” etc. TRAILING NULLCOLS means if the last column is null then treat this as null value, otherwise,  SQL LOADER will treat the record as bad if the last column is null.  
7.        In this line specify the columns of the target table. Note how do you specify format for Date columns

**Step 4**

After you have wrote the control file save it and then, call SQL Loader utility by typing the following command

$sqlldr userid=scott/tiger control=emp.ctl log=emp.log

After you have executed the above command SQL Loader will shows you the output describing how many rows it has loaded.   
  
The LOG option of sqlldr specifies where the log file of this sql loader session should be created. The log file contains all actions which SQL loader has performed i.e. how many rows were loaded, how many were rejected and how much time is taken to load the rows and etc. You have to view this file for any errors encountered while running SQL Loader.

Pipeline function:

Pipelined functions, a very classic example is where you do a SELECT \* FROM table name in SQL\*Plus. What happens is, Oracle streams the data from the table..

Like watching a video in youtube.

Please note the word, '*Streaming*'.. And in our function we define how many rows we stream.. Every *streamed* row is immediately available to the caller. Pipelining means in lay man terms, dont make me wait till you complete, give me what ever you have, and keep processing and updating me simultaneously.

In your last procedure, after piping every row, you initiate a sleep call for 10s , so the record is *Streamed* to the caller every 10s.

And, a normal table function will keep waiting until all processing work is done, and then it will return the reference to the result set cursor.

pipelined functions , they claim to save memory, is by flushing the content immediately, and hence the buffer being used is always minimal, whereas the round trips count get higher.

CREATE OR REPLACE FUNCTION FN\_RET\_COL RETURN TY\_1\_TABLE

PIPELINED IS

BEGIN

PIPE ROW(TY\_1('A',1,'10-JUN-2013'));

DBMS\_LOCK.sleep(seconds => 10);

PIPE ROW(TY\_1('B',2,'11-JUN-2013'));

DBMS\_LOCK.sleep(seconds => 10);

PIPE ROW(TY\_1('C',3,'12-JUN-2013'));

END;

**ORACLE UTL\_FILE**

**How UTL\_FILE PACKAGE used for File I/O Operations?**

In Oracle PL/SQL, **UTL\_FILE** is an Oracle supplied package which is used for file operations (read and write). **UTL\_FILE** works for both server and client machine systems. A directory has to be created on the server, which points to the target file. For the files located on the server machine, the actual path can be given while creating the directory. For the files, which are located on the client machines, however, the relative path is required along with the client machine name.

In addition, the relative file path must be in shared mode with read and write access for the required users. A DBA must create the directory and then grant Read/Write access to the required users.

**Process Flow:**

**In order to write to a file, you will (in most cases) perform the following steps:**

* Declare a file handle. This handle serves as a pointer to the file for subsequent calls to modules in the UTL\_FILE package to manipulate the contents of this file.
* Open the file with a call to FOPEN, which returns a file handle to the file. You can open a file to read, replace, or append text.
* Write data to the file using the PUT, PUTF, or PUT\_LINE procedures.
* Close the file with a call to FCLOSE. This releases resources associated with the file.

**In order to read data from a file you will (in most cases) perform the following steps:**

* Declare a file handle.
* Declare a VARCHAR2 string buffer that will receive the line of data from the file.
* Open the file using FOPEN in read mode.
* Use the GET\_LINE procedure to read data from the file and into the buffer. To read all the     lines from a file, you would execute GET\_LINE in a loop.
* Close the file with a call to FCLOSE.

DECLARE

2 Fhandle Utl\_File.File\_Type;

3 Begin

4 Fhandle := Utl\_File.Fopen( 'TESTDIR','FILE\_DATA.txt','W');

5

6 /\* TESTDIR: File Location | FILE\_DATA.txt: File Name | W: Open Mode: W- Write \*/

7

8 Utl\_File.Put(Fhandle, 'How Are You Doing Today'|| CHR(10));

9 Utl\_File.Put(Fhandle, 'I am Doing Good');

10 Utl\_File.Fclose(Fhandle);

11 Exception

12 When Others Then

13 Dbms\_Output.Put\_Line('ERROR: ' || SQLCODE|| ' - ' || SQLERRM);

14 Raise;

15 END;

16 /

PL/SQL procedure successfully completed.

Some UTL\_FILE subprograms are listed as below.

* **FOPEN** - Opens a file for input or output. FOPEN takes the following parameters:  
  the File Location, Filename, OPEN\_MODE and the Max\_Linesize.
* **FCLOSE** - Closes a file.
* **FCLOSE\_ALL** - Closes all open file handles
* **FCOPY** - Copies a contiguous portion of a file to a newly created file. Takes the following parameters:  
  src\_location, src\_filename, dest\_location, dest\_filename, start\_line, and end\_line.
* **FFLUSH** - Physically writes all pending output to a file.
* **FGETATTR** - Reads and returns the attributes of a disk file. Returns the following items about the file:  
  location, filename, fexists (a boolean), file\_length (in bytes), and block\_size.  
  The *location* must be either an existing directory on the server AND be in the utl\_file\_dir parameter, or it may be a directory.

Local and Global index usage comes during the Partitioning of the table.

When using Oracle partitioning, you can specify the "global" or "local" parameter in the create index syntax.

**Global Index**

A global index is a one-to-many relationship, allowing one index partition to map to many table partitions.  The docs say that a "global index can be partitioned by the range or hash method, and it can be defined on any type of partitioned, or non-partitioned, table".

**SYNTAX: -**

      CREATE INDEX item\_idx

on all\_fact (item\_nbr)

GLOBAL

(PARTITION city\_idx1 VALUES LESS THAN (100)),

(PARTITION city\_idx1 VALUES LESS THAN (200)),

(PARTITION city\_idx1 VALUES LESS THAN (300)),

(PARTITION city\_idx1 VALUES LESS THAN (400)),

(PARTITION city\_idx1 VALUES LESS THAN (500));

**Local Index**

A local index is a one-to-one mapping between an index partition and a table partition.  In general, local indexes allow for a cleaner "divide and conquer" approach for generating fast SQL execution plans with partition pruning.

Temporary table:

Data stored temporarily. Data stored as long as the session or transaction lasts and is private for each  session.

* The Definition is visible to all sessions.
* Data stored in RAM so processing will be faster. Logical read from RAM.
* After Commit or Disconnection, the data is lost but table definition remains in DB.
* Foreign key constraints are not applicable in case of Temporary table.
* Temporary tables are useful while doing runtime operations.
* We can store the data at runtime and after completing the operation the data will be deleted.

**ORACLE Table Locking:**

When a transaction updates a row, it puts a lock so that no one can update the same row until it commits. When another transaction issues an update to the same row, it waits until the first one either commits or rolls back. After the first transaction performs a commit or rollback, the update by the second transaction is executed immediately, since the lock placed by the first transaction is now gone.

**Types of locking**

**1. Implicit locks:** Oracle automatically locks the rows whenever user performs DML operations.

**2. Explicit locks:** Provided by user and can be Row and Table Level.

**Row level Locks (TX):** Used to lock selected rows of table. It is imposed by "for update" clause in select.

**Select for Update**

The Select for Update statement allows you to lock the records in the cursor result set.

You are not required to make changes to the records in order to use this statement.

The record locks are released when the next commit or rollback statement is issued.

**Table level Locks (TM):** Used to lock complete table and can be done in below modes

**Shared Lock**:This type is placed on a record when the record is being viewed (Read Operation using Select).So, it permits other user to query the data on the tables but doesn’t allow any change. There can be multiple shared lock on a record at a time.  
 **Exclusive lock**This is placed when Insert, Update or Delete command is performed (Write Operation).  There can be only one exclusive lock on a record at a time.

**Deadlocks**

A deadlock occurs when two or more sessions are waiting for data locked by each other, resulting in all the sessions being blocked. Oracle automatically detects and resolves deadlocks by rolling back the statement associated with the transaction that detects the deadlock.

How to Kill the Session:**How to find Table Lock – Release Table lock – Find Session ID and kill the Session?**

Select Session\_ID  
From   
Dba\_Dml\_Locks  
where Name = 'EMP\_SAL';