SECTION III: Structured Query Language (SQL)

7. INTERACTIVE SQL PART - I

TABLE FUNDAMENTALS

A **table** is database object that holds user data. The simplest analogy is to think of a table as a spreadsheet. The cells of the spreadsheet equate to the columns of a table having a specific **data type** associated with them. If the spreadsheet cell has a number data type associated with it, then storing letters (i.e. characters) in the same cell is not allowed. The same logic is applied to a table's column. Each column of the table will have a specific data type bound to it. Oracle ensures that only data, which is identical to the data type of the column, will be stored within the column.

Oracle Data Types

Basic Data Types

Data types come in several forms and sizes, allowing the programmer to create tables suited to the scope of the project. The decisions made in choosing proper data types greatly influence the performance of a database, so it is wise to have a detailed understanding of these concepts.

Oracle is capable of many of the data types that even the novice programmer has probably already been exposed to. Refer to table 7.1 for some of the more commonly used include:

Data Type	Description
CHAR(size)	This data type is used to store character strings values of fixed length. The size in brackets determines the number of characters the cell can hold. The maximum number of characters (i.e. the size) this data type can hold is 255 characters. The data held is right-padded with spaces to whatever length specified. For example: In case of **Name CHAR(60)**, if the data held in the variable Name is only 20 characters in length, then the entry will be padded with 40 characters worth of spaces. These spaces will be removed when the value is retrieved though. These entries will be sorted and compared by MySQL in case-insensitive fashions unless the BINARY keyword is associated with it. The BINARY attribute means that column values are sorted and compared in case-sensitive fashion using the underlying character code values rather then a lexical ordering. BINARY doesn't affect how the column is stored or retrieved.
VARCHAR (size) / VARCHAR 2(size)	This data type is used to store variable length alphanumeric data. It is a more flexible form of the CHAR data type. The maximum this data type can hold upto 4000 characters. One difference between this data type and the CHAR data type is ORACLE compares VARCHAR values using non-padded comparison semantics i.e. the inserted values will not be padded with spaces. It also represents data of type String, yet stores this data in variable length format. VARCHAR can hold **1** to **255** characters. VARCHAR is usually a wiser choice than CHAR, due to it's variable length format characteristic. But, keep in mind, that CHAR is much faster than VARCHAR, sometimes up to **50%**.

Table 7.1

Data Type	Description
DATE	This data type is used to represent date and time. The standard format is DD-MON-YY as in 21-JUN-04. To enter dates other than the standard format, use the appropriate functions. DateTime stores date in the 24-hour format. By default, the time in a date field is 12:00:00 am, if no time portion is specified. The default date for a date field is the first day of the current month. Valid dates range from January 1, 4712 B.C. to December 31, 4712 A.D.
NUMBER (P, S)	The NUMBER data type is used to store numbers (fixed or floating point). Numbers of virtually any magnitude maybe stored up to 38 digits of precision. Valid values are 0, and positive and negative numbers with magnitude 1.0E-130 to 9.9…E125. Numbers may be expressed in two ways: first, with the numbers 0 to 9, the signs + and -, and a decimal point (.); second, in scientific notation, such as, 1.85E3 for 1850. The precision (P), determines the maximum length of the data, whereas the scale (S), determines the number of places to the right of the decimal. If scale is omitted then the default is zero. If precision is omitted, values are stored with their original precision upto the maximum of 38 digits.
LONG	This data type is used to store variable length character strings containing upto 2 GB. LONG data can be used to store arrays of binary data in ASCII format. Only one LONG value can be defined per table. LONG values cannot be used in subqueries, functions, expressions, where clauses or indexes and the normal character functions such as SUBSTR cannot be applied to LONG values. A table containing a LONG value cannot be clustered.
RAW/ LONG RAW	The RAW /LONG RAW data types are used to store binary data, such as digitized picture or image. Data loaded into columns of these data types are stored without any further conversion. RAW data type can have a maximum length of 255 bytes. LONG RAW data type can contain up to 2 GB. Values stored in columns having LONG RAW data type cannot be indexed.

Table 7.1 (Continued)

Comparison Between Oracle 8i/9i For Various Oracle Data Types

Data Type	Oracle 8i	Oracle 9i	Explanation
dec(p, s)	The maximum precision is 38 digits.	The maximum precision is 38 digits.	Where *p* is the precision and *s* is the scale. For example, dec(3,1) is a number that has 2 digits before the decimal and 1 digit after the decimal.
decimal(p, s)	The maximum precision is 38 digits.	The maximum precision is 38 digits.	Where *p* is the precision and *s* is the scale. For example, decimal(3,1) is a number that has 2 digits before the decimal and 1 digit after the decimal.
double precision			
float			
int			
integer			
real			
smallint			

Table 7.2

Data Type	Oracle 8i	Oracle 9i	Explanation
numeric(p, s)	The maximum precision is 38 digits.	The maximum precision is 38 digits.	Where *p* is the precision and *s* is the scale. For example, numeric(7,2) is a number that has 5 digits before the decimal and 2 digits after the decimal.
number(p, s)	The maximum precision is 38 digits.	The maximum precision is 38 digits.	Where *p* is the precision and *s* is the scale. For example, number(7,2) is a number that has 5 digits before the decimal and 2 digits after the decimal.
char (size)	Up to 32767 bytes in PLSQL. Up to 2000 bytes in Oracle 8i.	Up to 32767 bytes in PLSQL. Up to 2000 bytes in Oracle 9i.	Where *size* is the number of characters to store. Fixed-length strings. Space padded.
varchar2 (size)	Up to 32767 bytes in PLSQL. Up to 4000 bytes in Oracle 8i.	Up to 32767 bytes in PLSQL. Up to 4000 bytes in Oracle 9i.	Where *size* is the number of characters to store. Variable-length strings.
long	Up to 2 gigabytes.	Up to 2 gigabytes.	Variable-length strings. (backward compatible)
raw	Up to 32767 bytes in PLSQL. Up to 2000 bytes in Oracle 8i.	Up to 32767 bytes in PLSQL. Up to 2000 bytes in Oracle 9i.	Variable-length binary strings
long raw	Up to 2 gigabytes.	Up to 2 gigabytes.	Variable-length binary strings. (backward compatible)
date	A date between Jan 1, 4712 BC and Dec 31, 9999 AD.	A date between Jan 1, 4712 BC and Dec 31, 9999 AD.	
timestamp (*fractional seconds precision*)	Not supported in Oracle 8i.	*fractional seconds precision* must be a number between 0 and 9. (default is 6)	Includes year, month, day, hour, minute, and seconds. For example: timestamp(6)
timestamp (*fractional seconds precision*) with time zone	Not supported in Oracle 8i.	*fractional seconds precision* must be a number between 0 and 9. (default is 6)	Includes year, month, day, hour, minute, and seconds; with a time zone displacement value. For example: timestamp(5) with time zone
timestamp (*fractional seconds precision*) with local time zone	Not supported in Oracle 8i.	*fractional seconds precision* must be a number between 0 and 9. (default is 6)	Includes year, month, day, hour, minute, and seconds; with a time zone expressed as the session time zone. For example: timestamp(4) with local time zone
interval year (*year precision*) to month	Not supported in Oracle 8i.	*year precision* must be a number between 0 and 9. (default is 2)	Time period stored in years and months. For example: interval year(4) to month
urowid [size]	Up to 2000 bytes.	Up to 2000 bytes.	Universal rowid. Where *size* is optional.

Table 7.2 (Continued)

Data Type	Oracle 8i	Oracle 9i	Explanation
interval day (*day precision*) to second (*fractional seconds precision*)	Not supported in Oracle 8i.	*day precision* must be a number between 0 and 9. (default is 2) *fractional seconds precision* must be a number between 0 and 9. (default is 6)	Time period stored in days, hours, minutes, and seconds. For example: interval day(2) to second(6)
rowid	The format of the rowid is: BBBBBBB.RRRR.FFFFF Where BBBBBBB is the block in the database file; RRRR is the row in the block; FFFFF is the database file.	The format of the rowid is: BBBBBBB.RRRR.FFFFF Where BBBBBBB is the block in the database file; RRRR is the row in the block; FFFFF is the database file.	Fixed-length binary data. Every record in the database has a physical address or rowid.
boolean	Valid in PLSQL, but this datatype does not exist in Oracle 8i.	Valid in PLSQL, but this datatype does not exist in Oracle 9i.	
nchar (size)	Up to 32767 bytes in PLSQL. Up to 2000 bytes in Oracle 8i	Up to 32767 bytes in PLSQL. Up to 2000 bytes in Oracle 9i.	Where *size* is the number of characters to store. Fixed-length NLS string
nvarchar2 (size)	Up to 32767 bytes in PLSQL. Up to 4000 bytes in Oracle 8i.	Up to 32767 bytes in PLSQL. Up to 4000 bytes in Oracle 9i.	Where *size* is the number of characters to store. Variable-length NLS string
bfile	Up to 4 gigabytes.	Up to 4 gigabytes.	File locators that point to a read-only binary object outside of the database
blob	U p to 4 gigabytes.	Up to 4 gigabytes.	LOB locators that point to a large binary object within the database
clob	Up to 4 gigabytes.	Up to 4 gigabytes.	LOB locators that point to a large character object within the database
nclob	Up to 4 gigabytes.	Up to 4 gigabytes.	LOB locators that point to a large NLS character object within the database

Table 7.2 (Continued)

Prior using a table to store user data it needs to be created. Table creation is done using the **Create Table** syntax. When Oracle creates a table in response to a create table command, it stores table structure information within its Data Dictionary.

The CREATE TABLE Command

The **CREATE TABLE** command defines each column of the table uniquely. Each column has a minimum of three attributes, a name, datatype and size (**i.e.** column width). Each table column definition is a single clause in the create table syntax. Each table column definition is separated from the other by a comma. Finally, the SQL statement is terminated with a semi colon.

Rules For Creating Tables

1. A name can have maximum upto 30 characters
2. Alphabets from A-Z, a-z and numbers from 0-9 are allowed
3. A name should begin with an alphabet
4. The use of the special character like _ is allowed and also recommended. (Special characters like $, # are allowed **only in Oracle**).
5. SQL reserved words **not** allowed. For Example: create, select, and so on.

Syntax:

```
CREATE TABLE <TableName>
    (<ColumnName1> <DataType>(<size>), <ColumnName2> <DataType>(<Size>));
```

Note

Each column must have a datatype. The column should either be defined as null or not null and if this value is left blank, the database assumes "null" as the default.

A Brief Checklist When Creating Tables

The following provides a small checklist for the issues that need to be considered before creating a table:

- ❑ What are the attributes of the rows to be stored?
- ❑ What are the data types of the attributes?
- ❑ Should varchar2 be used instead of char?
- ❑ Which columns should be used to build the primary key?
- ❑ Which columns do (not) allow null values? Which columns do / do not, allow duplicates?
- ❑ Are there default values for certain columns that **also** allow null values?

Example 1:
Create the **BRANCH_MSTR** table as shown in the **Chapter 6** along with the structure for other table belonging to the **Bank System**.

```
CREATE TABLE "DBA_BANKSYS"."BRANCH_MSTR"(
    "BRANCH_NO" VARCHAR2(10),
    "NAME" VARCHAR2(25));
```

Output:

```
Table created.
```

Note

All table columns belong to a **single record**. Therefore all the table column definitions are enclosed within parenthesis.

Inserting Data Into Tables

Once a table is created, the most natural thing to do is load this table with data to be manipulated later.

When inserting a single row of data into the table, the insert operation:

- Creates a new row (empty) in the database table
- Loads the values passed (by the SQL insert) into the columns specified

Syntax:

```
INSERT INTO <tablename> (<columnname1>, <columnname2>)
    VALUES (<expression>, <expression2>);
```

Example 2:
Insert the values into the BRANCH_MSTR table (For values refer to 6th chapter under Test Records)

```
INSERT INTO BRANCH_MSTR (BRANCH_NO, NAME) VALUES('B1', 'Vile Parle (HO)');
INSERT INTO BRANCH_MSTR (BRANCH_NO, NAME) VALUES('B2', 'Andheri');
INSERT INTO BRANCH_MSTR (BRANCH_NO, NAME) VALUES('B3', 'Churchgate');
INSERT INTO BRANCH_MSTR (BRANCH_NO, NAME) VALUES('B4', 'Sion');
INSERT INTO BRANCH_MSTR (BRANCH_NO, NAME) VALUES('B5', 'Borivali');
INSERT INTO BRANCH_MSTR (BRANCH_NO, NAME) VALUES('B6', 'Matunga');
```

Output for each of the above INSERT INTO statements:

```
1 row created.
```

Tip

Character expressions placed within the **INSERT INTO** statement must be enclosed in **single quotes** (').

In the **INSERT INTO** SQL sentence, table columns and values have a one to one relationship, (**i.e.** the first value described is inserted into the first column, and the second value described is inserted into the second column and so on).

Hence, in an **INSERT INTO** SQL sentence if there are exactly the same numbers of values as there are columns and the values are sequenced in exactly in accordance with the data type of the table columns, there is **no need** to indicate the column names.

However, if there are less values being described than there are columns in the table then it is **mandatory** to indicate **both** the table **column name** and its corresponding **value** in the **INSERT INTO** SQL sentence.

In the absence of mapping a table column name to a value in the **INSERT INTO** SQL sentence, the Oracle engine will not know which columns to insert the data into. This will generally cause a loss of data integrity. Then the data held within the table will be largely useless.

Note

Refer to the file **Chap07_Adtn.pdf**, for the **INSERT INTO** statement belonging to the remaining tables as mentioned in **Chapter 6**. These statements are built on the test data mentioned in **Chapter 6: Test Records For Retail Banking**.

VIEWING DATA IN THE TABLES

Once data has been inserted into a table, the next most logical operation would be to view what has been inserted. The **SELECT** SQL verb is used to achieve this. The SELECT command is used to retrieve rows selected from one or more tables.

All Rows And All Columns

In order to view global table data the **syntax** is:

```
SELECT <ColumnName 1> TO <ColumnName N> FROM TableName;
```

Note

Here, **ColumnName1 to ColumnName N** represents table column names.

Syntax:

```
SELECT * FROM <TableName>;
```

Example 3:
Show all employee numbers, first name, middle name and last name who work in the bank.

SELECT EMP_NO, FNAME, MNAME, LNAME **FROM** EMP_MSTR;

Output:

EMP_NO	FNAME	MNAME	LNAME
E1	Ivan	Nelson	Bayross
E2	Amit		Desai
E3	Maya	Mahima	Joshi
E4	Peter	Iyer	Joseph
E5	Mandhar	Dilip	Dalvi
E6	Sonal	Abdul	Khan
E7	Anil	Ashutosh	Kambli
E8	Seema	P.	Apte
E9	Vikram	Vilas	Randive
E10	Anjali	Sameer	Pathak

10 rows selected.

Example 4:
Show all the details related to the Fixed Deposit Slab

SELECT * FROM FDSLAB_MSTR;

Output:

FDSLAB_NO	MINPERIOD	MAXPERIOD	INTRATE
1	1	30	5
2	31	92	5.5
3	93	183	6
4	184	365	6.5
5	366	731	7.5

Output: (Continuted)

```
6          732        1097
7         1098        1829         8.5
7 rows selected.                      10
```

Tip

When data from all rows and columns from the table are to be viewed the syntax of the SELECT statement will be: **SELECT * FROM <TableName>;**

Oracle allows the use of the Meta character asterisk (*), this is expanded by Oracle to mean all rows and all columns in the table.

The Oracle Server parses and compiles the SQL query, executes it, and retrieved data from all rows/columns from the table.

Filtering Table Data

While viewing data from a table it is rare that **all the data** from the table will be required **each time**. Hence, SQL provides a method of filtering table data that is not required.

The ways of filtering table data are:

- Selected columns and all rows
- Selected rows and all columns
- Selected columns and selected rows

Selected Columns And All Rows

The retrieval of specific columns from a table can be done as shown below:

Syntax:

```
SELECT <ColumnName1>, <ColumnName2> FROM <TableName>;
```

Example 5:
Show the first name and the last name of the bank employees

```
SELECT FNAME, LNAME FROM EMP_MSTR;
```

Output:

```
FNAME      LNAME
---------- ----------
Ivan       Bayross
Amit       Desai
Maya       Joshi
Peter      Joseph
Mandhar    Dalvi
Sonal      Khan
Anil       Kambli
Seema      Apte
Vikram     Randive
Anjali     Pathak
10 rows selected.
```

Selected Rows And All Columns

If information of a particular client is to be retrieved from a table, its retrieval must be based on a **specific condition**.

The SELECT statement used until now displayed all rows. This is because there was no condition set that informed Oracle about how to choose a specific set of rows (or a specific row) from any table. Oracle provides the option of using a **WHERE Clause** in an SQL query to apply a filter on the rows retrieved.

When a where clause is added to the SQL query, the Oracle engine compares each record in the table with the condition specified in the where clause. The Oracle engine displays only those records that satisfy the specified condition.

Syntax:

```
SELECT * FROM <TableName> WHERE <Condition>;
```

Here, **<Condition>** is always quantified as **<ColumnName = Value>**

Example 6:
Display the branch details of the branch named Vile Parle (HO)

SELECT * **FROM** BRANCH_MSTR **WHERE** NAME = 'Vile Parle (HO)';

Output:

```
BRANCH_NO       NAME
--------------------------------------
B1              Vile Parle (HO)
```

Note

When specifying a condition in the **where clause** all standard operators such as logical, arithmetic, predicates and so on, can be used.

Selected Columns And Selected Rows

To view a specific set of rows and columns from a table the syntax will be as follows:

Syntax:

```
SELECT <ColumnName1>, <ColumnName2> FROM <TableName>
   WHERE <Condition>;
```

Example 7:
List the savings bank account numbers and the branch to which they belong.

SELECT ACCT_NO, BRANCH_NO **FROM** ACCT_MSTR **WHERE** TYPE = 'SB';

Selected Rows And All Columns

If information of a particular client is to be retrieved from a table, its retrieval must be based on a **specific condition**.

The SELECT statement used until now displayed all rows. This is because there was no condition set that informed Oracle about how to choose a specific set of rows (**or** a specific row) from any table. Oracle provides the option of using a **WHERE Clause** in an SQL query to apply a filter on the rows retrieved.

When a where clause is added to the SQL query, the Oracle engine compares each record in the table with the condition specified in the where clause. The Oracle engine displays only those records that satisfy the specified condition.

Syntax:

```
SELECT * FROM <TableName> WHERE <Condition>;
```

Here, **<Condition>** is always quantified as **<ColumnName = Value>**

Example 6:
Display the branch details of the branch named Vile Parle (HO)

SELECT * **FROM** BRANCH_MSTR **WHERE** NAME = 'Vile Parle (HO)';

Output:

```
BRANCH_NO       NAME
--------------------------------------
B1              Vile Parle (HO)
```

Note

When specifying a condition in the **where clause** all standard operators such as logical, arithmetic, predicates and so on, can be used.

Selected Columns And Selected Rows

To view a specific set of rows and columns from a table the syntax will be as follows:

Syntax:

```
SELECT <ColumnName1>, <ColumnName2> FROM <TableName>
   WHERE <Condition>;
```

Example 7:
List the savings bank account numbers and the branch to which they belong.

SELECT ACCT_NO, BRANCH_NO **FROM** ACCT_MSTR **WHERE** TYPE = 'SB';

Output:

```
ACCT_NO    BRANCH_NO
SB1        B1
SB3        B1
SB5        B6
SB6        B4
SB8        B2
SB9        B4
7 rows selected.
```

ELIMINATING DUPLICATE ROWS WHEN USING A SELECT STATEMENT

A table could hold duplicate rows. In such a case, to view only unique rows the distinct clause can be used. The DISTINCT clause allows removing duplicates from the result set. The DISTINCT clause can only be used with select statements.

The **DISTINCT** clause scans through the values of the column/s specified and displays only unique values from amongst them.

Syntax:

```
SELECT DISTINCT <ColumnName1>, <ColumnName2> FROM <TableName>;
```

The **SELECT DISTINCT** * SQL syntax scans through **entire rows**, and eliminates rows that have exactly the same contents in each column.

Syntax:

```
SELECT DISTINCT * FROM <TableName>;
```

Example 8:
Show different types of occupations of the bank customers by eliminating the repeated occupations

```
SELECT DISTINCT OCCUP FROM CUST_MSTR;
```

Output:

```
OCCUP
------------------------------------
Business
Community Welfare
Executive
Information Technology
Retail Business
Self Employed
Service
7 rows selected.
```

First insert one more record in the table **BRANCH_MSTR** so as to see the output for the next query example.

```
INSERT INTO BRANCH_MSTR (BRANCH_NO, NAME) VALUES('B6', 'Matunga');
```

Example 9:
Show only unique branch details.

SELECT DISTINCT * FROM BRANCH_MSTR;

The following output shows the entry for B6 only once even though entered twice in the table.

Output:

```
BRANCH_NO      NAME
---------------------------------
B1             Vile Parle (HO)
B2             Andheri
B3             Churchgate
B4             Sion
B5             Borivali
B6             Matunga
6 rows selected.
```

SORTING DATA IN A TABLE

Oracle allows data from a table to be viewed in a sorted order. The rows retrieved from the table will be sorted in either **ascending** or **descending** order depending on the condition specified in the **SELECT** sentence. The syntax for viewing data in a sorted order is as follows:

Syntax:

```
SELECT * FROM <TableName>
    ORDER BY <ColumnName1>, <ColumnName2> <[Sort Order]>;
```

The ORDER BY clause sorts the result set based on the columns specified. The ORDER BY clause can only be used in SELECT statements.

Example 10:
Show details of the branch according to the branch's name.

SELECT * FROM BRANCH_MSTR **ORDER BY** NAME;

Output:

```
BRANCH_NO  NAME
---------------------------------
B2         Andheri
B5         Borivali
B3         Churchgate
B6         Matunga
B6         Matunga
B4         Sion
B1         Vile Parle (HO)
7 rows selected.
```

Tip

For viewing data in descending sorted order the word **DESC** must be mentioned **after** the column name and before the semi colon in the **order by** clause. In case there is no mention of the sort order, the Oracle engine sorts in **ascending order by default**.

Example 11:
Show the details of the branch according to the branch's name in descending order.

SELECT * FROM BRANCH_MSTR ***ORDER BY*** NAME ***DESC;***

Output:

```
BRANCH_NO      NAME
---------      ------------------------------
B1             Vile Parle (HO)
B4             Sion
B6             Matunga
B6             Matunga
B3             Churchgate
B5             Borivali
B2             Andheri

7 rows selected.
```

CREATING A TABLE FROM A TABLE

Syntax:

```
CREATE TABLE <TableName> (<ColumnName>, < ColumnName>)
   AS SELECT <ColumnName>, <ColumnName> FROM <TableName>
```

Example 12:
Create a table named ACCT_DTLS having three fields i.e. ACCT_NO, BRANCH_NO and CURBAL from the source table named ACCT_MSTR and rename the field CURBAL to BALANCE.

CREATE TABLE ACCT_DTLS (ACCT_NO, BRANCH_NO, BALANCE)
AS SELECT ACCT_NO, BRANCH_NO, CURBAL ***FROM*** ACCT_MSTR;

Output:

```
Table created.
```

Note

If the Source Table **Acct_Mstr** was populated with records then the target table **Acct_Dtls** will also be populated with the same.

The **Source** table is the table identified in the **SELECT** section of this SQL sentence. The **Target** table is one identified in the **CREATE** section of this SQL sentence. This SQL sentence populates the Target table with data from the Source table.

To create a Target table without the records from the source table (i.e. create the structure only), the select statement must have a **WHERE clause**. The **WHERE clause** must specify a condition that **cannot** be satisfied.

This means the **SELECT** statement in the CREATE TABLE definition **will not retrieve** any rows from the source table, it will just retrieve the table structure thus the target table will be created empty.

Example 13:
Create a table named ACCT_DTLS having three fields i.e. ACCT_NO, BRANCH_NO and CURBAL from the source table named ACCT_MSTR and rename the field CURBAL to BALANCE. The table ACCT_DTLS should not be populated with any records.

CREATE TABLE ACCT_DTLS (ACCT_NO, BRANCH_NO, BALANCE)
AS SELECT ACCT_NO, BRANCH_NO, CURBAL **FROM** ACCT_MSTR **WHERE** 1=2;

Output:
```
Table created.
```

INSERTING DATA INTO A TABLE FROM ANOTHER TABLE

In addition to inserting data one row at a time into a table, it is quite possible to populate a table with data that already exists in another table. The syntax for doing so is as follows:

Syntax:
INSERT INTO <TableName>
SELECT <ColumnName 1>, <ColumnName N> **FROM** <TableName>;

Example 14:
Insert data in the table ACCT_DTLS using the table ACCT_MSTR as a source of data.

INSERT INTO ACCT_DTLS **SELECT** ACCT_NO, BRANCH_NO, CurBal **FROM** ACCT_MSTR;

Output:
```
10 rows created.
```

Insertion Of A Data Set Into A Table From Another Table

Syntax:
INSERT INTO <TableName> **SELECT** <ColumnName 1>, <ColumnName N>
FROM <TableName> **WHERE** <Condition>;

Example 15:
Insert only the savings bank accounts details in the target table ACCT_DTLS.

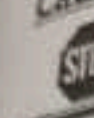
INSERT INTO ACCT_DTLS **SELECT** ACCT_NO, BRANCH_NO, CurBal **FROM** ACCT_MSTR
WHERE ACCT_NO **LIKE** 'SB%';

Output:
```
6 rows created.
```

DELETE OPERATIONS

The DELETE command deletes rows from the table that satisfies the condition provided by its where clause, **and** returns the number of records deleted.

Caution



If a DELETE statement without a WHERE clause is issued then, all rows are deleted.

The verb **DELETE** in SQL is used to remove either:

- All the rows from a table

OR

- A set of rows from a table

Removal Of All Rows

Syntax:

```
DELETE FROM <TableName>;
```

Example 16:
Empty the ACCT_DTLS table

```
DELETE FROM ACCT_DTLS;
```

Output:

```
16 rows deleted.
```

Removal Of Specific Row(s)

Syntax:

```
DELETE FROM <TableName> WHERE <Condition>;
```

Example 17:
Remove only the savings bank accounts details from the ACCT_DTLS table.

```
DELETE FROM ACCT_DTLS WHERE ACCT_NO LIKE 'SB%';
```

Output:

```
6 rows deleted.
```

Removal Of Specific Row(s) Based On The Data Held By The Other Table

Sometimes it is desired to delete records in one table based on values in another table. Since it is not possible to list more than one table in the FROM clause while performing a delete, the EXISTS clause can be used.

Example 18:
Remove the address details of the customer named **Ivan**.

```
DELETE FROM ADDR_DTLS WHERE EXISTS(SELECT FNAME FROM CUST_MSTR
    WHERE CUST_MSTR.CUST_NO = ADDR_DTLS.CODE_NO
        AND CUST_MSTR.FNAME = 'Ivan');
```

Output:
```
1 row deleted.
```

Explanation:
This will delete all records in the ADDR_DTLS table where there is a record in the CUST_MSTR table whose FNAME is Ivan, and the CUST_NO field belonging to the table CUST_MSTR is the same as the CODE_NO belonging to the table ADDR_DTLS.

UPDATING THE CONTENTS OF A TABLE

The **UPDATE** command is used to change or modify data values in a table.

The verb update in SQL is used to either update:
- All the rows from a table

OR
- A select set of rows from a table

Updating All Rows

The **UPDATE** statement updates columns in the existing table's rows with new values. The **SET** clause indicates which column data should be modified and the new values that they should hold. The **WHERE** clause, if given, specifies which rows should be updated. Otherwise, **all** table rows are updated.

Syntax:
```
UPDATE <TableName>
    SET <ColumnName1> = <Expression1>, <ColumnName2> = <Expression2>;
```

Example 19:
Update the address details by changing its city name to Bombay

```
UPDATE ADDR_DTLS SET City = 'Bombay';
```

Output:
```
44 rows updated.
```

Updating Records Conditionally

Syntax:
```
UPDATE <TableName>
    SET <ColumnName1> = <Expression1>, <ColumnName2> = <Expression2>
        WHERE <Condition>;
```

Example 20:
Update the branch details by changing the Vile Parle (HO) to head office.

```
UPDATE BRANCH_MSTR SET NAME = 'Head Office'
    WHERE NAME = 'Vile Parle (HO)';
```

Output:
```
1 row updated.
```

MODIFYING THE STRUCTURE OF TABLES

The structure of a table can be modified by using the **ALTER TABLE** command. **ALTER TABLE** allows **changing the structure of an existing table**. With **ALTER TABLE** it is possible to **add** or **delete** columns, **create** or **destroy** indexes, **change the data type** of existing columns, or **rename columns** or the **table** itself.

ALTER TABLE works by making a temporary copy of the original table. The alteration is performed on the copy, then the original table is deleted and the new one is renamed. While **ALTER TABLE** is executing, the original table is still readable by users of Oracle.

Updates and **writes** to the table are **stalled** until the new table is ready, and then are automatically redirected to the new table **without any failed updates**.

Note

To use **ALTER TABLE**, the **ALTER**, **INSERT**, and **CREATE** privileges for the table are required.

Adding New Columns

Syntax:

```
ALTER TABLE <TableName>
   ADD(<NewColumnName> <Datatype> (<Size>),
       <NewColumnName> <Datatype> (<Size>)...);
```

Example 21:
Enter a new field called City in the table BRANCH_MSTR.

```
ALTER TABLE BRANCH_MSTR ADD (CITY VARCHAR2(25));
```

Output:
```
Table altered.
```

Dropping A Column From A Table

Syntax:

```
ALTER TABLE <TableName> DROP COLUMN <ColumnName>;
```

Example 22:
Drop the column city from the BRANCH_MSTR table.

```
ALTER TABLE BRANCH_MSTR DROP COLUMN CITY;
```

Output:
```
Table altered.
```

Modifying Existing Columns

Syntax:

```
ALTER TABLE <TableName>
        MODIFY (<ColumnName> <NewDatatype>(<NewSize>));
```

Example 23:
Alter the BRANCH_MSTR table to allow the NAME field to hold maximum of 30 characters

```
ALTER TABLE BRANCH_MSTR MODIFY (NAME varchar2(30));
```

Output:

```
Table altered.
```

Restrictions on the ALTER TABLE

The following tasks **cannot** be performed when using the **ALTER TABLE** clause:

- Change the name of the table
- Change the name of the column
- Decrease the size of a column if table data exists

RENAMING TABLES

Oracle allows renaming of tables. The rename operation is done **atomically**, which means that **no other thread** can access any of the tables while the rename process is running.

Note

To rename a table the ALTER and DROP privileges on the original table, and the CREATE and INSERT privileges on the new table are required.

To rename a table, the syntax is

Syntax:

```
RENAME <TableName> TO <NewTableName>
```

Example 24:
Change the name of branches table to branch table

```
RENAME BRANCH_MSTR TO BRANCHES;
```

Output:

```
Table renamed.
```

TRUNCATING TABLES

TRUNCATE TABLE empties a table completely. Logically, this is equivalent to a **DELETE** statement that deletes all rows, but there are practical differences under some circumstances.

TRUNCATE TABLE differs from **DELETE** in the following ways:

- Truncate operations drop and re-create the table, which is much faster than deleting rows one by one
- Truncate operations are not transaction-safe (i.e. an error will occur if an active transaction or an active table lock exists)
- The number of deleted rows are not returned

Syntax:

```
TRUNCATE TABLE <TableName>;
```

Example 25:
Truncate the table BRANCH_MSTR

```
TRUNCATE TABLE BRANCH_MSTR;
```

Output:

```
Table truncated.
```

DESTROYING TABLES

Sometimes tables within a particular database become obsolete and need to be discarded. In such situation using the DROP TABLE statement with the table name can destroy a specific table.

Syntax:

```
DROP TABLE <TableName>;
```

Caution

If a table is dropped all records held within it are lost and cannot be recovered.

Example 26:
Remove the table BRANCH_MSTR along with the data held.

```
DROP TABLE BRANCH_MSTR;
```

Output:

```
Table dropped.
```

CREATING SYNONYMS

A synonym is an alternative name for objects such as tables, views, sequences, stored procedures, and other database objects.

Syntax:

```
CREATE [OR REPLACE]  [PUBLIC] SYNONYM [SCHEMA .]
    SYNONYM_NAME FOR [SCHEMA .]
    OBJECT_NAME [@ DBLINK];
```

In the syntax,

- The **OR** replace phrase allows to recreate the synonym (if it already exists) without having to issue a DROP synonym command.
- The **PUBLIC** phrase means that the synonym is a public synonym and is accessible to all users. Remember though that the user must first have the appropriate privileges to the object to use the synonym.
- The **SCHEMA** phrase is the appropriate schema. If this phrase is omitted, Oracle assumes that a reference is made to the user's own schema.
- The **OBJECT_NAME** phrase is the name of the object for which you are creating the synonym. It can be one of the following:
  - Table
  - Package
  - View
  - Materialized View
  - Sequence
  - Java Class Schema Object
  - Stored Procedure
  - User-Defined Object
  - Function
  - Synonym

Example 27:
Create a synonym to a table named EMP held by the user SCOTT.

```
CREATE PUBLIC SYNONYM EMPLOYEES FOR SCOTT.EMP;
```

Output:

```
Synonym created.
```

Explanation:
Now, users of other schemas can reference the table EMP, which is now called as EMPLOYEES without having to prefix the table name with the schema named SCOTT. For example:

```
SELECT * FROM EMPLOYEES;
```

Dropping Synonyms

Syntax:

```
DROP [PUBLIC] SYNONYM [SCHEMA.]SYNONYM_NAME [FORCE];
```

In the syntax,

- The **PUBLIC** phrase allows to drop a public synonym. If public is specified, then there is no need to specify a schema.
- The **FORCE** phrase will force Oracle to drop the synonym even if it has dependencies. It is probably not a good idea to use the force phrase as,it can cause invalidation of Oracle objects

Example 28:
Drop the public synonym named EMPLOYEES

```
DROP PUBLIC SYNONYM EMPLOYEES;
```

Output:

```
Synonym dropped.
```

EXAMINING OBJECTS CREATED BY A USER

Finding Out The Table/s Created By A User

The command shown below is used to determine the tables to which a user has access. The tables created under the **currently selected** tablespace are displayed.

Example 29:

```
SELECT * FROM TAB;
```

Output:

```
TNAME                          TABTYPE  CLUSTERID
------------------------------ -------  ----------
ACCT_FD_CUST_DTLS              TABLE
ACCT_MSTR                      TABLE
ADDR_DTLS                      TABLE
BRANCH_MSTR                    TABLE
CNTC_DTLS                      TABLE
CUST_MSTR                      TABLE
EMP_MSTR                       TABLE
FDSLAB_MSTR                    TABLE
FD_DTLS                        TABLE
FD_MSTR                        TABLE
NOMINEE_MSTR                   TABLE
SPRT_DOC                       TABLE
TRANS_DTLS                     TABLE
TRANS_MSTR                     TABLE
14 rows selected.
```

Displaying The Table Structure

To display information about the columns defined in a table use the following syntax

Syntax:

```
DESCRIBE <TableName>;
```

This command displays the column names, the data types and the special attributes connected to the table.

Example 30:
Show the table structure of table BRANCH_MSTR

```
DESCRIBE BRANCH_MSTR;
```

Output:

```
Name                      Null?    Type
------------------------- -------- ----------
BRANCH_NO                          VARCHAR2(10)
NAME                               VARCHAR2(25)
```

SELF REVIEW QUESTIONS

FILL IN THE BLANKS

1. A _____ is a database object that holds user data.
2. Table creation is done using the _____ syntax.
3. Character expressions placed within the insert into statement must be enclosed in _____ quotes.
4. Oracle provides the option of using a _____ _____ in an SQL query to apply a filter on the rows retrieved.
5. The _____ _____ SQL syntax scans through the values of the column/s specified and displays only unique values from amongst them.
6. The SQL sentence populates the _____ table with data from the _____ table.
7. The name of the column cannot be changed using the _____ _____ clause.
8. The _____ command is used to change or modify data values in a table.
9. All table columns belong to a _____ _____.

TRUE OR FALSE

10. If a spreadsheet has a number data type associated with, then it can store characters as well.
11. Each table column definition is separated from the other by a colon.
12. All table columns belong to a single record.
13. In the insert into SQL sentence table columns and values have a one to many relationship.
14. The SELECT DISTINCT SQL syntax scans through entire rows, and eliminates rows that have exactly the same contents in each column.
15. When specifying a condition in the where clause only logical standard operators can be used.
16. Oracle allows data from a table to be viewed in a sorted order.
17. In order to view the data in descending sorted order the word 'desc' must be mentioned after the column name and before the semi colon in the order by clause.
18. The MODIFY command is used to change or modify data values in a table.
19. The name of the table cannot be changed using the ALTER TABLE clause.

HANDS ON EXERCISES

1. *Create the tables described below:*

Table Name: CLIENT_MASTER
Description: Used to store client information.

Column Name	Data Type	Size	Default	Attributes
CLIENTNO	Varchar2	6		
NAME	Varchar2	20		
ADDRESS1	Varchar2	30		
ADDRESS2	Varchar2	30		
CITY	Varchar2	15		
PINCODE	Number	8		
STATE	Varchar2	15		
BALDUE	Number	10,2		

Table Name: PRODUCT_MASTER
Description: Used to store product information.

Column Name	Data Type	Size	Default	Attributes
PRODUCTNO	Varchar2	6		
DESCRIPTION	Varchar2	15		
PROFITPERCENT	Number	4,2		
UNITMEASURE	Varchar2	10		
QTYONHAND	Number	8		
REORDERLVL	Number	8		
SELLPRICE	Number	8,2		
COSTPRICE	Number	8,2		

Table Name: SALESMAN_MASTER
Description: Used to store salesman information working for the company.

Column Name	Data Type	Size	Default	Attributes
SALESMANNO	Varchar2	6		
SALESMANNAME	Varchar2	20		
ADDRESS1	Varchar2	30		
ADDRESS2	Varchar2	30		
CITY	Varchar2	20		
PINCODE	Number	8		
STATE	Varchar2	20		
SALAMT	Number	8,2		
TGTTOGET	Number	6,2		
YTDSALES	Number	6,2		
REMARKS	Varchar2	60		

2. Insert the following data into their respective tables:

a) Data for **CLIENT_MASTER** table:

ClientNo	Name	City	Pincode	State	BalDue
C00001	Ivan Bayross	Mumbai	400054	Maharashtra	15000
C00002	Mamta Muzumdar	Madras	780001	Tamil Nadu	0
C00003	Chhaya Bankar	Mumbai	400057	Maharashtra	5000
C00004	Ashwini Joshi	Bangalore	560001	Karnataka	0
C00005	Hansel Colaco	Mumbai	400060	Maharashtra	2000
C00006	Deepak Sharma	Mangalore	560050	Karnataka	0

b) Data for **PRODUCT_MASTER** table:

ProductNo	Description	Profit Percent	Unit Measure	QtyOn Hand	ReorderLvl	SellPrice	CostPrice
P00001	T-Shirts	5	Piece	200	50	350	250
P0345	Shirts	6	Piece	150	50	500	350
P06734	Cotton Jeans	5	Piece	100	20	600	450
P07865	Jeans	5	Piece	100	20	750	500
P07868	Trousers	2	Piece	150	50	850	550
P07885	Pull Overs	2.5	Piece	80	30	700	450
P07965	Denim Shirts	4	Piece	100	40	350	250
P07975	Lycra Tops	5	Piece	70	30	300	175
P08865	Skirts	5	Piece	75	30	450	300

c) Data for **SALESMAN_MASTER** table:

SalesmanNo	Name	Address1	Address2	City	PinCode	State
S00001	Aman	A/14	Worli	Mumbai	400002	Maharashtra
S00002	Omkar	65	Nariman	Mumbai	400001	Maharashtra
S00003	Raj	P-7	Bandra	Mumbai	400032	Maharashtra
S00004	Ashish	A/5	Juhu	Mumbai	400044	Maharashtra

SalesmanNo	SalAmt	TgtToGet	YtdSales	Remarks
S00001	3000	100	50	Good
S00002	3000	200	100	Good
S00003	3000	200	100	Good
S00004	3500	200	150	Good

3. Exercise on retrieving records from a table

a. Find out the names of all the clients.
b. Retrieve the entire contents of the Client_Master table.
c. Retrieve the list of names, city and the sate of all the clients.
d. List the various products available from the Product_Master table.
e. List all the clients who are located in Mumbai.
f. Find the names of salesmen who have a salary equal to Rs.3000.

4. Exercise on updating records in a table

a. Change the city of ClientNo 'C00005' to 'Bangalore'.
b. Change the BalDue of ClientNo 'C00001' to Rs. 1000.
c. Change the cost price of 'Trousers' to Rs. 950.00.
d. Change the city of the salesman to Pune.

5. Exercise on deleting records in a table
a. Delete all salesmen from the Salesman_Master whose salaries are equal to Rs. 3500.
b. Delete all products from Product_Master where the quantity on hand is equal to 100.
c. Delete from Client_Master where the column state holds the value 'Tamil Nadu'.

6. Exercise on altering the table structure
a. Add a column called 'Telephone' of data type 'number' and size ='10' to the Client_Master table.
b. Change the size of SellPrice column in Product_Master to 10,2.

7. Exercise on deleting the table structure along with the data
a. Destroy the table Client_Master along with its data.

8. Exercise on renaming the table
a. Change the name of the Saleśman_Master table to sman_mast.

8. INTERACTIVE SQL PART – II

DATA CONSTRAINTS

All businesses of the world run on business data being gathered, stored and analyzed. Business managers determine a set of business rules that must be applied to their data prior to it being stored in the database/table to ensure its integrity.

For instance, no employee in the sales department can have a salary of less than Rs.1000/-.

Such rules have to be enforced on data stored. Only data, which satisfies the conditions set, should be stored for future analysis. If the data gathered fails to satisfy the conditions set, it must be rejected. This ensures that the data stored in a table will be valid, and have integrity.

Business rules that are applied to data are completely **System dependent**. The rules applied to data gathered and processed by a **Savings bank system** will be very different, to the business rules applied to data gathered and processed by an **Inventory system**, which in turn will be very different, to the business rules applied to data gathered and processed by a **Personnel management system**.

Business rules, which are enforced on data being stored in a table, are called **Constraints**. Constraints, **super control** the data being entered into a table for permanent storage.

To understand the concept of data constraints, several tables will be created and different types of constraints will be applied to table columns **or** the table itself. The set of tables are described below. Appropriate examples of data constraints are bound to these tables.

Applying Data Constraints

Oracle permits data constraints to be attached to table columns via SQL syntax that checks data for integrity prior storage. Once data constraints are part of a table column construct, the Oracle database engine checks the data being entered into a table column against the data constraints. If the data passes this check, it is stored in the table column, else the data is rejected. Even if a single column of the record being entered into the table fails a constraint, the **entire record is rejected and not stored in the table**.

Both the **Create Table** and **Alter Table** SQL verbs can be used to write SQL sentences that attach constraints (i.e. Business / System rules) to a table column.

Caution

Until now tables created in this material have **not** had any data constraints attached to their table columns. Hence the tables have **not** been given any instructions to filter what is being stored in the table. This situation **can** and **does**, result in erroneous data being stored in the table.

Once a constraint is attached to a table column, any SQL **INSERT** or **UPDATE** statement automatically causes these constraints to be applied to data prior it is being inserted into the table column for storage.

Note

Oracle also permits applying data constraints at **Table level**. More on table level constraints later in this material.

TYPES OF DATA CONSTRAINTS

There are two types of data constraints that can be applied to data being inserted into a Oracle table. One type of constraint is called an **I/O** constraint (**I**nput / **O**utput). This data constraint determines the speed at which data can be inserted or extracted from a Oracle table. The other type of constraint is called a **business rule** constraint.

I/O Constraints

The input/output data constraints are further divided into **two** distinctly different constraints.

The PRIMARY KEY Constraint

A primary key is one or more column(s) in a table used to uniquely identify **each row** in the table. None of the fields that are part of the primary key can contain a null value. A table can have only one primary key.

A **primary key column** in a table has special attributes:

- It defines the column, as a mandatory column (**i.e.** the column cannot be left blank). As the NOT NULL attribute is active
- The data held across the column MUST be UNIQUE

A single column primary key is called a **Simple** key. A multicolumn primary key is called a **Composite** primary key. The only function of a primary key in a table is to **uniquely identify a row**. When a record cannot be uniquely identified using a value in a simple key, a composite key must be defined. A primary key can be defined in either a **CREATE TABLE** statement or an **ALTER TABLE** statement.

For example, a **SALES_ORDER_DETAILS** table will hold multiple records that are sales orders. Each such sales order will have multiple products that have been ordered. Standard business rules do not allow multiple entries for the same product. However, multiple orders will definitely have multiple entries of the same product.

Under these circumstances, the only way to uniquely identify a row in the **SALES_ORDER_DETAILS** table is via a composite primary key, consisting of **ORDER_NO** and **PRODUCT_NO**. Thus the combination of order number and product number will uniquely identify a row.

Features of Primary key

1. Primary key is a column or a set of columns that uniquely identifies a row. Its main purpose is the, **Record Uniqueness**
2. Primary key will not allow duplicate values
3. Primary key will also not allow null values
4. Primary key is not compulsory but it is recommended
5. Primary key helps to identify one record from another record and also helps in relating tables with one another
6. Primary key cannot be LONG or LONG RAW data type
7. Only one Primary key is allowed per table
8. Unique Index is created automatically if there is a Primary key
9. One table can combine upto 16 columns in a Composite Primary key

PRIMARY KEY Constraint Defined At Column Level

Syntax:

```
<ColumnName> <Datatype>(<Size>) PRIMARY KEY
```

Example 1:
Drop the CUST_MSTR table, if it already exists. Create a table CUST_MSTR such that the contents of the column CUST_NO is unique and not null.

```
DROP TABLE CUST_MSTR;
CREATE TABLE CUST_MSTR (
    "CUST_NO" VARCHAR2(10) PRIMARY KEY,
    "FNAME" VARCHAR2(25), "MNAME" VARCHAR2(25),
    "LNAME" VARCHAR2(25), "DOB_INC" DATE,
    "OCCUP" VARCHAR2(25), "PHOTOGRAPH" VARCHAR2(25),
    "SIGNATURE"  VARCHAR2(25), "PANCOPY" VARCHAR2(1),
    "FORM60" VARCHAR2(1));
```

Output:
```
Table created.
```

For testing purpose, execute the following **INSERT INTO** statement:

```
INSERT INTO CUST_MSTR (CUST_NO, FNAME, MNAME, LNAME, DOB_INC, OCCUP, PHOTOGRAPH,
                       SIGNATURE, PANCOPY, FORM60)
    VALUES('C1', 'Ivan', 'Nelson', 'Bayross', '25-JUN-1952', 'Self Employed', 'D:/ClntPht/C1.gif',
           'D:/ClntSgnt/C1.gif', 'Y', 'Y');
```

Output:
```
1 row created.
```

To verify whether the Primary Key Constraint is functional, reissue the same **INSERT INTO** statement. The result is the following error:

Output:
```
INSERT INTO CUST_MSTR (CUST_NO, FNAME, MNAME, LNAME, DOB_INC, OCCUP,
PHOTOGRAPH, SIGNATURE, PANCOPY,
*
ERROR at line 1:
ORA-00001: unique constraint (DBA_BANKSYS.SYS_C003009) violated
```

PRIMARY KEY Constraint Defined At Table Level

Syntax:

```
PRIMARY KEY (<ColumnName>, <ColumnName>)
```

Example 2:
Drop the FD_MSTR table, if it already exists. Create a table FD_MSTR where there is a composite primary key mapped to the columns FD_SER_NO and CORP_CUST_NO.
Since this constraint spans across columns, it must be described at table level.

```
DROP TABLE FD_MSTR;
CREATE TABLE "DBA_BANKSYS"."FD_MSTR"(
    "FD_SER_NO" VARCHAR2(10), "SF_NO" VARCHAR2(10),
    "BRANCH_NO" VARCHAR2(10), "INTRO_CUST_NO" VARCHAR2(10),
    "INTRO_ACCT_NO" VARCHAR2(10), "INTRO_SIGN" VARCHAR2(1),
    "ACCT_NO" VARCHAR2(10), "TITLE" VARCHAR2(30),
    "CORP_CUST_NO" VARCHAR2(10), "CORP_CNST_TYPE" VARCHAR(4),
    "VERI_EMP_NO" VARCHAR2(10), "VERI_SIGN" VARCHAR2(1),
    "MANAGER_SIGN" VARCHAR2(1), PRIMARY KEY(FD_SER_NO, CORP_CUST_NO));
```

Output:
```
Table created.
```

For testing purpose, execute the following **INSERT INTO** statement:

```
INSERT INTO FD_MSTR (FD_SER_NO, SF_NO, BRANCH_NO, ACCT_NO, TITLE, CORP_CUST_NO,
                     CORP_CNST_TYPE, INTRO_CUST_NO, INTRO_ACCT_NO, INTRO_SIGN,
                     VERI_EMP_NO, VERI_SIGN, MANAGER_SIGN)
    VALUES ('FS1', 'SF-0011', 'B1', 'CA2', 'Uttam Stores', 'O11', '1C', null, null, 'N', 'E1', 'Y', 'Y');
```

Output:
```
1 row created.
```

To verify whether the Composite Primary Key Constraint is functional, reissue the same **INSERT INTO** statement. The result is the following error:

Output:
```
INSERT   INTO   FD_MSTR   (FD_SER_NO,   SF_NO,   BRANCH_NO,   ACCT_NO,   TITLE,
CORP_CUST_NO, CORP_CNST_TYPE, INTR
*
ERROR at line 1:
ORA-00001: unique constraint (DBA_BANKSYS.SYS_C003010) violated
```

Now, simply modify the INSERT INTO statement as show below, to allow the record to pass the composite primary key constraint:

```
INSERT INTO FD_MSTR (FD_SER_NO, SF_NO, BRANCH_NO, ACCT_NO, TITLE, CORP_CUST_NO,
                     CORP_CNST_TYPE, INTRO_CUST_NO, INTRO_ACCT_NO, INTRO_SIGN,
                     VERI_EMP_NO, VERI_SIGN, MANAGER_SIGN)
    VALUES ('FS2', 'SF-0012', 'B1', 'CA4', 'Sun"s Pvt. Ltd.', 'C12', '4C', null, null, 'N', 'E1', 'Y', 'Y');
```

Output:
```
1 row created.
```

The Foreign Key (Self Reference) Constraint

Foreign keys represent relationships between tables. A foreign key is a column (or a group of columns) whose values are derived from the **primary key** or **unique key** of some other table.

The table in which the foreign key is defined is called a **Foreign table** or **Detail table**. The table that defines the **primary** or **unique** key and is referenced by the **foreign key** is called the **Primary table** or **Master table**. A Foreign key can be defined in either a CREATE TABLE statement or an ALTER TABLE statement.

The master table can be referenced in the foreign key definition by using the clause **REFERENCES TableName.ColumnName** when defining the foreign key, column attributes, in the detail table.

Features of Foreign Keys

1. Foreign key is a column(s) that references a column(s) of a table and it can be the same table also
2. Parent that is being referenced has to be unique or Primary key
3. Child may have duplicates and nulls but unless it is specified
4. Foreign key constraint can be specified on child but not on parent
5. Parent record can be delete provided no child record exist
6. Master table cannot be updated if child record exist

This constraint establishes a relationship between records (i.e. column data) across a Master and a Detail table.

This relationship ensures:

- Records cannot be **inserted** into a **detail table** if corresponding records in the master table do not exist
- Records of the master **table** cannot be **deleted** if corresponding records in the detail table actually exist

Insert Or Update Operation In The Foreign Key Table

The existence of a foreign key implies that the table with the foreign key is **related** to the master table from which the foreign key is derived. A foreign key must have a corresponding primary key or unique key value in the master table.

For example a personnel information system includes two tables (i.e. department and employee). An employee cannot belong to a department that does not exist. Thus the department number specified in the employee table must be present in the department table.

Delete Operation On The Primary Key Table

Oracle **displays an error message** when a record in the master table is deleted and corresponding records exists in a detail table and **prevents the delete operation** from going through.

Note

The default behavior of the foreign key can be changed, by using the **ON DELETE CASCADE** option. When the **ON DELETE CASCADE** option is specified in the foreign key definition, if a record is deleted in the master table, all corresponding records in the detail table along with the record in the master table will be deleted.

Principles of **Foreign Key/References** constraint:

- Rejects an **INSERT** or **UPDATE** of a value, if a corresponding value does not currently exist in the master key table
- If the **ON DELETE CASCADE** option is set, a **DELETE** operation in the master table will trigger a **DELETE** operation for corresponding records **in all** detail tables
- If the **ON DELETE SET NULL** option is set, a **DELETE** operation in the master table will set the value held by the foreign key of the detail tables to null
- **Rejects** a **DELETE** from the Master table if **corresponding records** in the DETAIL table **exist**
- Must reference a **PRIMARY KEY** or **UNIQUE** column(s) in primary table
- Requires that the **FOREIGN KEY** column(s) and the **CONSTRAINT** column(s) have **matching** data types
- Can reference the same table named in the **CREATE TABLE** statement

FOREIGN KEY Constraint Defined At The Column Level

Syntax:

```
<ColumnName> <DataType>(<Size>)
    REFERENCES <TableName> [(<ColumnName>)]
        [ON DELETE CASCADE]
```

Example 3:
Drop the table EMP_MSTR, if it already exists. Create a table EMP_MSTR with its primary as EMP_NO referencing the foreign key BRANCH_NO in the BRANCH_MSTR table.

```
DROP TABLE EMP_MSTR;
CREATE TABLE "DBA_BANKSYS"."EMP_MSTR"(
    "EMP_NO" VARCHAR2(10) PRIMARY KEY,
    "BRANCH_NO" VARCHAR2(10) REFERENCES BRANCH_MSTR,
    "FNAME" VARCHAR2(25), "MNAME" VARCHAR2(25),
    "LNAME" VARCHAR2(25), "DEPT" VARCHAR2(30),
    "DESIG" VARCHAR2(30));
```

Output:
```
Table created.
```

The **REFERENCES** key word points to the table **BRANCH_MSTR**. The table **BRANCH_MSTR** has the column **BRANCH_NO** as its primary key column. Since no column is specified in the foreign key definition, Oracle applies an automatic (default) link to the primary key column i.e. **BRANCH_NO** of the table **BRANCH_MSTR**.

The foreign key definition is specified as

```
"BRANCH_NO" VARCHAR2(10) REFERENCES BRANCH_MSTR
```

FOREIGN KEY Constraint Defined At The Table Level

Syntax:

```
FOREIGN KEY (<ColumnName> [,<ColumnName>] )
    REFERENCES <TableName> [(<ColumnName>,<ColumnName>)
```

Example 4:
Drop the table ACCT_FD_CUST_DTLS, if it already exists. Create a table ACCT_FD_CUST_DTLS with CUST_NO as foreign key referencing column CUST_NO in the CUST_MSTR table

```
DROP TABLE ACCT_FD_CUST_DTLS;
CREATE TABLE "DBA_BANKSYS"."ACCT_FD_CUST_DTLS"(
   "ACCT_FD_NO" VARCHAR2(10), "CUST_NO" VARCHAR2(10),
   FOREIGN KEY (CUST_NO) REFERENCES CUST_MSTR(CUST_NO));
```

Output:
```
Table created.
```

FOREIGN KEY Constraint Defined With ON DELETE CASCADE

Example 5:
Drop the table FD_MSTR, if it already exists. Create a table FD_MSTR with its primary key as FD_SER_NO.

Drop the table FD_DTLS, if it already exists. Create a table FD_DTLS with its foreign key as FD_SER_NO with the ON DELETE CASCADE option. The foreign key is FD_SER_NO and is available as a primary key column named FD_SER_NO in the FD_MSTR table.

Insert some records into both the tables.

```
DROP TABLE FD_MSTR;
CREATE TABLE "DBA_BANKSYS"."FD_MSTR"(
        "FD_SER_NO" VARCHAR2(10) PRIMARY KEY,
        "SF_NO" VARCHAR2(10), "BRANCH_NO" VARCHAR2(10),
        "INTRO_CUST_NO" VARCHAR2(10), "INTRO_ACCT_NO" VARCHAR2(10),
        "INTRO_SIGN" VARCHAR2(1), "ACCT_NO" VARCHAR2(10),
        "TITLE" VARCHAR2(30), "CORP_CUST_NO" VARCHAR2(10),
        "CORP_CNST_TYPE" VARCHAR(4), "VERI_EMP_NO" VARCHAR2(10),
        "VERI_SIGN" VARCHAR2(1), "MANAGER_SIGN" VARCHAR2(1));
```

Output:
```
Table created.
```

```
DROP TABLE FD_DTLS;
CREATE TABLE "DBA_BANKSYS"."FD_DTLS"(
    "FD_SER_NO" VARCHAR2(10), "FD_NO" VARCHAR2(10),
    "TYPE" VARCHAR2(1), "PAYTO_ACCTNO" VARCHAR2(10),
    "PERIOD" NUMBER(5), "OPNDT" DATE,
    "DUEDT" DATE, "AMT" NUMBER(8,2),
    "DUEAMT" NUMBER(8,2), "INTRATE" NUMBER(3),
    "STATUS" VARCHAR2(1) DEFAULT 'A', "AUTO_RENEWAL" VARCHAR2(1) ,
    CONSTRAINT f_FDSerNoKey
        FOREIGN KEY (FD_SER_NO) REFERENCES FD_MSTR(FD_SER_NO)
        ON DELETE CASCADE);
```

Output:

```
Table created.
```

Now delete a record from the FD_MSTR table as:

```
DELETE FROM FD_MSTR WHERE FD_SER_NO = 'FS1';
```

Output:

```
1 row deleted.
```

Query the table FD_DTLS for records:

```
SELECT * FROM FD_DTLS;
```

Notice the deletion of the records belonging to **FS1**.

Explanation:
In this example, a primary key is created in the FD_MSTR table. It consists of only **one field** i.e. FD_SER_NO field. Then a foreign key is created in the FD_DTLS table that references the FD_MSTR table based on the contents of the FD_SER_NO field.

Because of the **cascade delete**, when a record in the FD_MSTR table is deleted, all records in the FD_DTLS table will also be deleted that have the same FD_SER_NO value.

FOREIGN KEY Constraint Defined With ON DELETE SET NULL:

A FOREIGN key with a **SET NULL ON DELETE** means that if a record in the parent table is deleted, then the corresponding records in the child table will have the foreign key fields set to **null**. The records in the child table **will not** be deleted.

A FOREIGN key with a **SET NULL ON DELETE** can be defined in either a CREATE TABLE statement or an ALTER TABLE statement.

Example 6:
Drop the table FD_MSTR, if it already exists. Create a table FD_MSTR with its primary key as FD_SER_NO.

Drop the table FD_DTLS, if it already exists. Create a table FD_DTLS with its foreign key as FD_SER_NO with the ON DELETE SET NULL option. The foreign key is FD_SER_NO and is available as a primary key column named FD_SER_NO in the FD_MSTR table.

Insert some records into both the tables.

```
DROP TABLE FD_MSTR;
```

```
CREATE TABLE "DBA_BANKSYS"."FD_MSTR"(
    "FD_SER_NO" VARCHAR2(10) PRIMARY KEY, "SF_NO" VARCHAR2(10),
    "BRANCH_NO" VARCHAR2(10), "INTRO_CUST_NO" VARCHAR2(10),
    "INTRO_ACCT_NO" VARCHAR2(10), "INTRO_SIGN" VARCHAR2(1),
    "ACCT_NO" VARCHAR2(10), "TITLE" VARCHAR2(30), "CORP_CUST_NO" VARCHAR2(10),
    "CORP_CNST_TYPE" VARCHAR(4), "VERI_EMP_NO" VARCHAR2(10),
    "VERI_SIGN" VARCHAR2(1), "MANAGER_SIGN" VARCHAR2(1));
```

Output:

```
Table created.
```

```
DROP TABLE FD_DTLS;
CREATE TABLE "DBA_BANKSYS"."FD_DTLS"(
  "FD_SER_NO" VARCHAR2(10), "FD_NO" VARCHAR2(10), "TYPE" VARCHAR2(1),
  "PAYTO_ACCTNO" VARCHAR2(10), "PERIOD" NUMBER(5), "OPNDT" DATE,
  "DUEDT" DATE, "AMT" NUMBER(8,2), "DUEAMT" NUMBER(8,2), "INTRATE" NUMBER(3),
  "STATUS" VARCHAR2(1) DEFAULT 'A', "AUTO_RENEWAL" VARCHAR2(1) ,
  CONSTRAINT f_FDSerNoKey
     FOREIGN KEY (FD_SER_NO) REFERENCES FD_MSTR(FD_SER_NO)
     ON DELETE SET NULL);
```

Output:

```
Table created.
```

Now delete a record from the FD_MSTR table as:

```
DELETE FROM FD_MSTR WHERE FD_SER_NO = 'FS1';
```

Output:

```
1 row deleted.
```

Query the table FD_DTLS for records:

```
SELECT * FROM FD_DTLS;
```

Notice the value held by the field FD_SER_NO.

Explanation:

In this example, a primary key is created in the FD_MSTR table. It consists of only one field i.e. FD_SER_NO field. Then a foreign key is created in the FD_DTLS table that references the FD_MSTR table based on the FD_SER_NO field.

Because of the **cascade set null**, when a record in the FD_MSTR table is deleted, all corresponding records in the FD_DTLS table will have the FD_SER_NO values set to **null**.

Assigning User Defined Names To Constraints

When constraints are defined, Oracle assigns a **unique name** to each constraint. The convention used by Oracle is

SYS_C**n**

where **n** is a **numeric value** that makes the constraint name **unique**.

Constraints can be given a unique user-defined name along with the constraint definition. A constraint can then, be dropped by referring to the constraint by its name. Under these circumstances a user-defined constraint name becomes very convenient.

User named constraints simplifies the task of dropping constraints. A constraint can be given a user-defined name by preceding the constraint definition with the reserved word **CONSTRAINT** and a **user-defined name**.

Syntax:

CONSTRAINT <Constraint Name> <Constraint Definition>

Example 7:
Drop the CUST_MSTR table, if it already exists. Create a table CUST_MSTR with a primary key constraint on the column CUST_NO and also define its constraint name.

```
DROP TABLE CUST_MSTR;
CREATE TABLE CUST_MSTR (
    "CUST_NO" VARCHAR2(10) CONSTRAINT p_CUSTKey PRIMARY KEY,
    "FNAME" VARCHAR2(25), "MNAME" VARCHAR2(25),
    "LNAME" VARCHAR2(25), "DOB_INC" DATE,
    "OCCUP" VARCHAR2(25), "PHOTOGRAPH" VARCHAR2(25),
    "SIGNATURE" VARCHAR2(25), "PANCOPY" VARCHAR2(1),
    "FORM60" VARCHAR2(1));
```

Output:

```
Table created.
```

Example 8:
Drop the table EMP_MSTR, if it already exists. Create a table EMP_MSTR with its foreign key as BRANCH_NO. The foreign key is BRANCH_NO available and as a primary key in the BRANCH_MSTR table. Also define the name of the foreign key.

```
DROP TABLE EMP_MSTR;
CREATE TABLE "DBA_BANKSYS"."EMP_MSTR"(
    "EMP_NO" VARCHAR2(10), "BRANCH_NO" VARCHAR2(10),
    "FNAME" VARCHAR2(25), "MNAME" VARCHAR2(25),
    "LNAME" VARCHAR2(25), "DEPT" VARCHAR2(30),
    "DESIG" VARCHAR2(30),
    CONSTRAINT f_BranchKey
        FOREIGN KEY (BRANCH_NO) REFERENCES BRANCH_MSTR);
```

Output:

```
Table created.
```

The Unique Key Constraint

The **Unique** column constraint **permits multiple entries** of NULL into the column. These NULL values are clubbed at the top of the column in the order in which they were entered into the table. This is the **essential difference** between the Primary Key and the Unique constraints when applied to table column(s).

Key point about Unique Constraint:

1. Unique key will not allow duplicate values
2. Unique index is created automatically
3. A table can have more than one Unique key which is not possible in Primary key
4. Unique key can combine upto 16 columns in a Composite Unique key
5. Unique key can not be LONG or LONG RAW data type

UNIQUE Constraint Defined At The Column Level

Syntax:

```
<ColumnName> <Datatype>(<Size>) UNIQUE
```

Example 9:
Drop the CUST_MSTR table, if it already exists. Create a table CUST_MSTR such that the contents of the column CUST_NO are unique across the entire column.

```
DROP TABLE CUST_MSTR;
CREATE TABLE CUST_MSTR (
   "CUST_NO" VARCHAR2(10) UNIQUE, "FNAME" VARCHAR2(25), "MNAME" VARCHAR2(25),
   "LNAME" VARCHAR2(25), "DOB_INC" DATE, "OCCUP" VARCHAR2(25),
   "PHOTOGRAPH" VARCHAR2(25), "SIGNATURE" VARCHAR2(25), "PANCOPY" VARCHAR2(1),
   "FORM60" VARCHAR2(1));
```

Output:

```
Table created.
```

For testing the Unique constraint execute the following INSERT INTO statements:

```
INSERT INTO CUST_MSTR (CUST_NO, FNAME, MNAME, LNAME, DOB_INC, OCCUP, PHOTOGRAPH,
                       SIGNATURE, PANCOPY, FORM60)
   VALUES('C1', 'Ivan', 'Nelson', 'Bayross', '25-JUN-1952', 'Self Employed', 'D:/ClntPht/C1.gif',
          'D:/ClntSgnt/C1.gif', 'Y', 'Y');
INSERT INTO CUST_MSTR (CUST_NO, FNAME, MNAME, LNAME, DOB_INC, OCCUP, PHOTOGRAPH,
                       SIGNATURE, PANCOPY, FORM60)
   VALUES('C1', 'Chriselle', 'Ivan', 'Bayross', '29-OCT-1982', 'Service', 'D:/ClntPht/C2.gif', 'D:/ClntSgnt/C2.gif', 'N',
          'Y');
INSERT INTO CUST_MSTR (CUST_NO, FNAME, MNAME, LNAME, DOB_INC, OCCUP, PHOTOGRAPH,
                       SIGNATURE, PANCOPY, FORM60)
   VALUES('C2', 'Mamta', 'Arvind', 'Muzumdar', '28-AUG-1975', 'Service', 'D:/ClntPht/C3.gif', 'D:/ClntSgnt/C3.gif',
          'Y', 'Y');
```

Output:
The first INSERT INTO statement will execute without any errors as show below:

```
1 row created.
```

When the second INSERT INTO statement is executed an errors occurs as show below:

```
INSERT INTO CUST_MSTR (CUST_NO, FNAME, MNAME, LNAME, DOB_INC, OCCUP,
PHOTOGRAPH, SIGNATURE, PANCOPY,
*
ERROR at line 1:
ORA-00001: unique constraint (DBA_BANKSYS.SYS_C003007) violated
```

The third INSERT INTO statement rectifies this and the result is as show below:

```
1 row created.
```

When a SELECT statement is executed on the Client_Master table the records reterived are:

SELECT CUST_NO, FNAME, MNAME, LNAME **FROM** CUST_MSTR;

Output:

```
CUST_NO    FNAME     MNAME     LNAME
---------- --------- --------- ---------
C1         Ivan      Nelson    Bayross
C2         Mamta     Arvind    Muzumdar
```

UNIQUE Constraint Defined At The Table Level

Syntax:

```
CREATE TABLE TableName
    (<ColumnName1> <Datatype>(<Size>), <ColumnName2> <Datatype>(<Size>),
      UNIQUE (<ColumnName1>, <ColumnName2>));
```

Example 10:
Drop the CUST_MSTR table, if it already exists. Create a table CUST_MSTR such that the contents of the column CUST_NO are unique across the entire column.

```
DROP TABLE CUST_MSTR;
CREATE TABLE CUST_MSTR (
   "CUST_NO" VARCHAR2(10), "FNAME" VARCHAR2(25),
   "MNAME" VARCHAR2(25), "LNAME" VARCHAR2(25),
   "DOB_INC" DATE, "OCCUP" VARCHAR2(25),
   "PHOTOGRAPH" VARCHAR2(25), "SIGNATURE"  VARCHAR2(25),
   "PANCOPY" VARCHAR2(1), "FORM60" VARCHAR2(1),
   UNIQUE(CUST_NO));
```

Output:

```
Table created.
```

In the case of the table level unique constraints, the result for the following INSERT INTO statement will remain the same as explained earlier.

```
INSERT INTO CUST_MSTR (CUST_NO, FNAME, MNAME, LNAME, DOB_INC, OCCUP, PHOTOGRAPH,
                       SIGNATURE, PANCOPY, FORM60)
   VALUES('C1', 'Ivan', 'Nelson', 'Bayross', '25-JUN-1952', 'Self Employed', 'D:/ClntPht/C1.gif',
          D:/ClntSgnt/C1.gif', 'Y', 'Y');
INSERT INTO CUST_MSTR (CUST_NO, FNAME, MNAME, LNAME, DOB_INC, OCCUP, PHOTOGRAPH,
                       SIGNATURE, PANCOPY, FORM60)
   VALUES('C1', 'Chriselle', 'Ivan', 'Bayross', '29-OCT-1982', 'Service', 'D:/ClntPht/C2.gif', 'D:/ClntSgnt/C2.gif', 'N',
          'Y');
INSERT INTO CUST_MSTR (CUST_NO, FNAME, MNAME, LNAME; DOB_INC, OCCUP, PHOTOGRAPH,
                       SIGNATURE, PANCOPY, FORM60)
   VALUES('C2', 'Mamta', 'Arvind', 'Muzumdar', '28-AUG-1975', 'Service', 'D:/ClntPht/C3.gif', 'D:/ClntSgnt/C3.gif',
          'Y', 'Y');
```

Business Rule Constraints

Oracle allows the application of **business rules** to table columns. Business managers determine business rules, they vary from system to system as mentioned earlier. These rules are applied to data, **prior** the data is being inserted into table columns. This ensures that the data (**records**) in the table have integrity.

For example, the rule that no employee in the company shall get a salary less than Rs.1000/- is a business rule. This means that no cell in the **salary** column of the employee table should hold a **value** less than 1000. If an attempt is made, to insert a value less than 1000 into the salary column, the database engine rejects the entire record automatically.

Business rules can be implemented in Oracle by using **CHECK** constraints. Check Constraints can be bound to a **column** or a **table** using the **CREATE TABLE** or **ALTER TABLE** command.

Business rule validation checks are performed when any table **write** *operation is carried out. Any insert or update statement causes the relevant Check constraint to be evaluated. The Check constraint must be satisfied for the write operation to succeed. Thus* **Check constraints** *ensure the integrity of the data in tables.*

Conceptually, data constraints are connected to a column, by the Oracle engine, as **flags**. *Whenever, an attempt is made to load the column with data, the Oracle engine observes the flag and recognizes the presence of a constraint. The Oracle engine then retrieves the Check constraint definition and then applies the Check constraint definition, to the data being loaded into the table column. If the data being entered into a column fails any of the data constraint checks, the* **entire** *record is rejected. The Oracle engine will then flash an appropriate* **error message**.

Oracle allows programmers to define constraints at:

- *Column Level*
- *Table Level*

Column Level Constraints

If data constraints are defined as an attribute of a column definition when creating or altering a table structure, they are **column level constraints**.

Caution

Column level constraints are applied to the **current column**. *The current column is the column that immediately* **precedes** *the constraint (***i.e.** *they are local to a specific column). A column level constraint* **cannot** *be applied if the data constraint spans* **across multiple columns** *in a table.*

Table Level Constraints

If data constraints are defined **after defining all table column attributes** *when creating or altering a table structure, it is a* **table level constraint**.

Note

A table level constraint **must** *be applied if the data constraint* **spans across multiple columns** *in a table.*

Constraints are stored as a part of the global table definition by the Oracle engine in its **system tables.** *The SQL syntax used to attach the constraint will change depending upon whether it is a column level or table level constraint.*

NULL Value Concepts

Often there may be records in a table that do not have values for every field. This could be because the information is not available at the time of data entry or because the field is not applicable in every case. If the column was created as **NULLABLE**, Oracle will place a NULL value in the column in the **absence** of a user-defined value.

A NULL value is **different from** a blank or a zero. A **NULL value** can be inserted into **columns** of **any data type**.

Principles Of NULL Values

- Setting a NULL value is appropriate when the actual value is unknown, or when a value would not be meaningful
- A NULL value is **not equivalent** to a value of **zero** if the data type is **number** and is not equivalent to **spaces** if the data type is **character**
- A NULL value will evaluate to NULL in any expression (**e.g.** NULL multiplied by 10 is NULL)
- NULL value can be inserted into columns of **any data type**
- If the column has a NULL value, Oracle ignores any UNIQUE, FOREIGN KEY, CHECK constraints that may be attached to the column

Difference Between An Empty String And A NULL Value

Oracle has changed its rules about empty strings and null values in newer versions of Oracle. Now, an empty string is treated as a null value in Oracle.

To understand this go through the following example:

Example 11:
Drop the BRANCH_MSTR table, if it already exists. Create a table BRANCH_MSTR such that the contents of the column CUST_NO are unique across the entire column.

```
DROP TABLE BRANCH_MSTR;
CREATE TABLE BRANCH_MSTR (
    "BRANCH_NO" VARCHAR2(10), "NAME" VARCHAR2(25));
```

Output:

```
Table created.
```

Next, insert two records into this table.

```
INSERT INTO BRANCH_MSTR (BRANCH_NO, NAME) VALUES('B1', null);
```

Output:

```
1 row created.
```

```
INSERT INTO BRANCH_MSTR (BRANCH_NO, NAME) VALUES('B2', '');
```

Output:

```
1 row created.
```

The first statement inserts a record with a branch name that is null, while the second statement inserts a record with an empty string as a branch name.

Now, retrieve all rows with a branch name that is an empty string value as follows:

```
SELECT * FROM BRANCH_MSTR WHERE NAME = '';
```

When this statement is executed, it is expected to retrieve the row that was inserted above. But instead, this statement will not retrieve any records at all.

Now, try retrieving all rows where the branch name contains a null value:

```
SELECT * FROM BRANCH_MSTR WHERE NAME IS NULL;
```

When this statement is executed, both rows are retrieved. This is because Oracle has now changed its rules so that empty strings behave as null values.

It is also important to note that the null value is unique. Usual operands such as =, <, > and so on cannot be used on a null value. Instead, the IS NULL and IS NOT NULL conditions have to be used.

NOT NULL Constraint Defined At The Column Level

In addition to Primary key and Foreign Key, Oracle has **NOT NULL** as column constraint. The **NOT NULL** column constraint ensures that a table column cannot be left empty.

When a column is defined as **not null**, then that column becomes a **mandatory** column. It implies that a value must be entered into the column if the record is to be accepted for storage in the table.

Syntax:

```
<ColumnName> <Datatype>(<Size>) NOT NULL
```

Example 12:
Drop the table CUST_MSTR, if already exists and then create it again making Date of Birth field not null. Refer to the details of table in chapter 6

```
CREATE TABLE "DBA_BANKSYS"."CUST_MSTR"(
  "CUST_NO" VARCHAR2(10), "FNAME" VARCHAR2(25),
  "MNAME" VARCHAR2(25), "LNAME" VARCHAR2(25),
  "DOB_INC" DATE NOT NULL, "OCCUP" VARCHAR2(25),
  "PHOTOGRAPH" VARCHAR2(25), "SIGNATURE"  VARCHAR2(25),
  "PANCOPY" VARCHAR2(1), "FORM60" VARCHAR2(1));
```

Output:

```
Table created.
```

Note

The **NOT NULL** constraint can only be applied at column level.

Execute the following INSERT INTO statements to verify whether mandatory field constraints are applied:

```
INSERT INTO CUST_MSTR (CUST_NO, FNAME, MNAME, LNAME, DOB_INC, OCCUP, PHOTOGRAPH,
                       SIGNATURE, PANCOPY, FORM60)
  VALUES('O14', null, null, null, null, 'Retail Business', null, null, 'N', 'Y');
```

Output:

```
INSERT INTO CUST_MSTR (CUST_NO, FNAME, MNAME, LNAME, DOB_INC, OCCUP,
PHOTOGRAPH, SIGNATURE, PANCOPY,
*
ERROR at line 1:
ORA-01400: cannot insert NULL into ("DBA_BANKSYS"."CUST_MSTR"."DOB_INC")
```

The above **error** message confirms that the mandatory field constraints are applied successfully.

Caution

The **NOT NULL** constraint can only be applied at column level. Although **NOT NULL** can be applied as a **CHECK** constraint, Oracle Corp recommends that this should **not be done.**

The CHECK Constraint

Business Rule validations can be applied to a table column by using **CHECK** constraint. CHECK constraints must be specified as a logical expression that evaluates either to **TRUE** or **FALSE**.

Note

A **CHECK** constraint takes substantially longer to execute as compared to NOT NULL, PRIMARY KEY, FOREIGN KEY or UNIQUE. Thus CHECK constraints must be avoided if the constraint can be defined using the Not Null, Primary key or Foreign key constraint.

CHECK constraint defined at the column level:

Syntax:

<ColumnName> <Datatype>(<Size>) CHECK (<Logical Expression>)

Example 13:

Drop the table CUST_MSTR, if already exists. Create a table **CUST_MSTR** with the following check constraints:

- Data values being inserted into the column **CUST_NO** must start with the capital letter **C**
- Data values being inserted into the column **FNAME, MNAME and LNAME** should be in **upper case** only

```
CREATE TABLE CUST_MSTR("CUST_NO" VARCHAR2(10) CHECK(CUST_NO LIKE 'C%'),
    "FNAME" VARCHAR2(25) CHECK (FNAME = UPPER(FNAME)),
    "MNAME" VARCHAR2(25) CHECK (MNAME = UPPER(MNAME)),
    "LNAME" VARCHAR2(25) CHECK (LNAME = UPPER(LNAME)), "DOB_INC" DATE,
    "OCCUP" VARCHAR2(25), "PHOTOGRAPH" VARCHAR2(25), "SIGNATURE"  VARCHAR2(25),
    "PANCOPY" VARCHAR2(1), "FORM60" VARCHAR2(1));
```

Output:

```
Table created.
```

CHECK Constraint Defined At The Table Level:

Syntax:

CHECK (<Logical Expression>)

Example 14:

Drop the table CUST_MSTR, if already exists. Create a table **CUST_MSTR** with the following check constraints:

- Data values being inserted into the column **CUST_NO** must start with the capital letter **C**
- Data values being inserted into the column **FNAME, MNAME and LNAME** should be in **upper case** only

```
CREATE TABLE CUST_MSTR("CUST_NO" VARCHAR2(10), "FNAME" VARCHAR2(25),
    "MNAME" VARCHAR2(25), "LNAME" VARCHAR2(25), "DOB_INC" DATE,
    "OCCUP" VARCHAR2(25), "PHOTOGRAPH" VARCHAR2(25), "SIGNATURE"  VARCHAR2(25),
    "PANCOPY" VARCHAR2(1), "FORM60" VARCHAR2(1),
    CHECK (CUST_NO LIKE 'C%'), CHECK (FNAME = UPPER(FNAME)),
    CHECK (MNAME = UPPER(MNAME)), CHECK (LNAME = UPPER(LNAME)));
```

Output:

```
Table created.
```

Execute the following INSERT INTO statements to verify whether check constraints are applied:

```
INSERT INTO CUST_MSTR (CUST_NO, FNAME, MNAME, LNAME, DOB_INC, OCCUP, PHOTOGRAPH,
                       SIGNATURE, PANCOPY, FORM60)
  VALUES('O14', 'SHARANAM', 'CHAITANYA', 'SHAH', '03-Jan-1981', 'Business', null, null, 'N', 'Y');
```

Output:

```
INSERT INTO CUST_MSTR (CUST_NO, FNAME, MNAME, LNAME, DOB_INC, OCCUP,
PHOTOGRAPH, SIGNATURE, PANCOPY,
*
ERROR at line 1:
ORA-02290: check constraint (DBA_BANKSYS.SYS_C003055) violated
```

```
INSERT INTO CUST_MSTR (CUST_NO, FNAME, MNAME, LNAME, DOB_INC, OCCUP, PHOTOGRAPH,
                       SIGNATURE, PANCOPY, FORM60)
  VALUES('C14', 'sharanam', 'CHAITANYA', 'SHAH', '03-Jan-1981', 'Business', null, null, 'N', 'Y');
```

Output:

```
INSERT INTO CUST_MSTR (CUST_NO, FNAME, MNAME, LNAME, DOB_INC, OCCUP,
PHOTOGRAPH, SIGNATURE, PANCOPY,
*
ERROR at line 1:
ORA-02290: check constraint (DBA_BANKSYS.SYS_C003056) violated
```

```
INSERT INTO CUST_MSTR (CUST_NO, FNAME, MNAME, LNAME, DOB_INC, OCCUP, PHOTOGRAPH,
                       SIGNATURE, PANCOPY, FORM60)
  VALUES('C14', 'SHARANAM', 'Chaitanya', 'SHAH', '03-Jan-1981', 'Business', null, null, 'N', 'Y');
```

Output:

```
INSERT INTO CUST_MSTR (CUST_NO, FNAME, MNAME, LNAME, DOB_INC, OCCUP,
PHOTOGRAPH, SIGNATURE, PANCOPY,
*
ERROR at line 1:
ORA-02290: check constraint (DBA_BANKSYS.SYS_C003057) violated
```

```
INSERT INTO CUST_MSTR (CUST_NO, FNAME, MNAME, LNAME, DOB_INC, OCCUP, PHOTOGRAPH,
                       SIGNATURE, PANCOPY, FORM60)
  VALUES('C14', 'SHARANAM', 'CHAITANYA', 'sHAh', '03-Jan-1981', 'Business', null, null, 'N', 'Y');
```

Output:

```
INSERT INTO CUST_MSTR (CUST_NO, FNAME, MNAME, LNAME, DOB_INC, OCCUP,
PHOTOGRAPH, SIGNATURE, PANCOPY,
*
ERROR at line 1:
ORA-02290: check constraint (DBA_BANKSYS.SYS_C003058) violated
```

The above **error** messages confirm that the check constraints are applied successfully.

When using **CHECK** constraints, consider the ANSI / ISO standard, which states that a CHECK constraint is violated only if the condition evaluates to **False**. A check constraint is not violated if the condition evaluates to **True**.

Note

If the expression in a check constraint does not return a **true / false**, the value is **Indeterminate** or **Unknown.** Unknown values do not violate a check constraint condition. For example, consider the following CHECK constraint for SellPrice column in the Product_Master table:

CHECK (SellPrice > 0)

At first glance, this rule may be interpreted as "do not allow a row in the **Product_Master** table unless **the Sellprice** is greater than 0". However, note that if a row is inserted with a **null SellPrice**, the row **does not violate** the CHECK constraint because the entire check condition is evaluated as **unknown**.

In this particular case, prevent such violations by placing the **not null** integrity constraint along with the check constraint on **SellPrice** column of the table **Product_Master**.

Restrictions On CHECK Constraints

A **CHECK** integrity constraint requires that a condition be **true** or **unknown** for the row to be processed. If an SQL statement causes the condition to evaluate to **false**, an appropriate error message is displayed and processing stops.

A **CHECK** constraint has the following limitations:

- The condition must be a **Boolean** expression that can be evaluated using the values in the row being inserted or updated.
- The condition cannot contain **subqueries** or **sequences**.
- The condition cannot include the SYSDATE, UID, USER or USERENV SQL functions.

DEFINING DIFFERENT CONSTRAINTS ON A TABLE

Example 15:

Drop the table FD_MSTR, if already exists. Create FD_MSTR table where

- The FD_SER_NO is a primary key to this table
- The BRANCH_NO is the foreign key referencing the table BRANCH_MSTR
- The CORP_CUST_NO is the foreign key referencing the table CUST_MSTR
- The VERI_EMP_NO is a foreign key referencing the table EMP_MSTR
- The CORP_CNST_TYPE will hold wither of the following values:
  ØS, 1C, 2C, 3C, 4C, 5C, 6C, 7C indicating different types of companies

```
CREATE TABLE "DBA_BANKSYS"."FD_MSTR"("FD_SER_NO" VARCHAR2(10),
  "SF_NO" VARCHAR2(10), "BRANCH_NO" VARCHAR2(10), "INTRO_CUST_NO" VARCHAR2(10),
  "INTRO_ACCT_NO" VARCHAR2(10), "INTRO_SIGN" VARCHAR2(1),
  "ACCT_NO" VARCHAR2(10), "TITLE" VARCHAR2(30), "CORP_CUST_NO" VARCHAR2(10),
  "CORP_CNST_TYPE" VARCHAR(4), "VERI_EMP_NO" VARCHAR2(10),
  "VERI_SIGN" VARCHAR2(1), "MANAGER_SIGN" VARCHAR2(1),
  CONSTRAINT PK PRIMARY KEY (FD_SER_NO, CORP_CUST_NO),
  CONSTRAINT FK_BR FOREIGN KEY (BRANCH_NO) REFERENCES BRANCH_MSTR,
  CONSTRAINT FK_CU FOREIGN KEY (CORP_CUST_NO) REFERENCES CUST_MSTR,
  CONSTRAINT FK_EM FOREIGN KEY (VERI_EMP_NO) REFERENCES EMP_MSTR,
  CONSTRAINT CHK CHECK (CORP_CNST_TYPE IN ('ØS', '1C', '2C', '3C', '4C', '5C', '6C', '7C')));
```

Output:

```
Table created.
```

THE USER_CONSTRAINTS TABLE

A table can be created with multiple constraints attached to its columns. If a user wishes to see the table structure along with its constraints, Oracle provides the **DESCRIBE <TableName>** command.

This command displays only the column names, data type, size and the NOT NULL constraint. The information about the other constraints that may be attached to the table columns such as the PRIMARY KEY, FOREIGN KEY, and so on, is not available using the DESCRIBE verb.

Oracle stores such information in a table called **USER_CONSTRAINTS**. Querying **USER_CONSTRAINTS** provides information bound to the names of all the constraints on the table. **USER_CONSTRAINTS** comprises of multiple columns, some of which are described below:

USER_CONSTRAINTS Table:

Column Name	Description
OWNER	The owner of the constraint.
CONSTRAINT_NAME	The name of the constraint
TABLE_NAME	The name of the table associated with the constraint
CONSTRAINT_TYPE	The type of constraint: **P**: Primary Key Constraint **R**: Foreign Key Constraint **U**: Unique Constraint **C**: Check Constraint
SEARCH_CONDITION	The search condition used (for CHECK Constraints)
R_OWNER	The owner of the table referenced by the FOREIGN KEY constraints
R_CONSTRAINT_NAME	The name of the constraint referenced by a FOREIGN KEY constraint.

Example 16:
View the constraints of the table CUST_MSTR

```
SELECT OWNER, CONSTRAINT_NAME, CONSTRAINT_TYPE FROM USER_CONSTRAINTS
    WHERE TABLE_NAME = 'CUST_MSTR';
```

Output:

```
OWNER          CONSTRAINT_NAME  CONSTRAINT_TYPE
-------------- ---------------  ---------------
DBA_BANKSYS    SYS_C003027      C
DBA_BANKSYS    SYS_C003028      C
DBA_BANKSYS    SYS_C003029      C
DBA_BANKSYS    SYS_C003030      C
```

DEFINING INTEGRITY CONSTRAINTS VIA THE ALTER TABLE COMMAND

Integrity constraints can be defined using the **constraint** clause, in the **ALTER TABLE** command.

Note

Oracle **will not allow** constraints defined using the **ALTER TABLE**, to be applied to the table if data previously placed in the table **violates such constraints**.

If a Primary key constraint was being applied to a table in retrospect and the column has duplicate values in it, the Primary key constraint **will not** be set to that column.

The following examples show the definitions of several integrity constraints:

Example 17:
Alter the table EMP_MSTR by adding a primary key on the column EMP_NO.

ALTER TABLE EMP_MSTR **ADD PRIMARY KEY** (EMP_NO);

Output:
```
Table altered.
```

Example 18:
Add FOREIGN KEY constraint on the column **VERI_EMP_NO** belonging to the table **FD_MSTR**, which references the table **EMP_MSTR**. Modify column **MANAGER_SIGN** to include the **NOT NULL** constraint

ALTER TABLE FD_MSTR **ADD CONSTRAINT** F_EmpKey **FOREIGN KEY**(VERI_EMP_NO)
REFERENCES EMP_MSTR **MODIFY**(MANAGER_SIGN **NOT NULL**);

Output:
```
Table altered.
```

DROPPING INTEGRITY CONSTRAINTS VIA THE ALTER TABLE COMMAND

Integrity constraint can be dropped if the rule that it enforces is no longer **true** or if the constraint is no longer **needed**. Drop the constraint using the **ALTER TABLE** command with the **DROP** clause. The following examples illustrate the dropping of integrity constraints:

Example 19:
Drop the PRIMARY KEY constraint from EMP_MSTR.

ALTER TABLE EMP_MSTR **DROP PRIMARY KEY;**

Output:
```
Table altered.
```

Example 20:
Drop FOREIGN KEY constraint on column VERI_EMP_NO in table FD_MSTR

ALTER TABLE FD_MSTR **DROP CONSTRAINT F_EmpKey;**

Output:
```
Table altered.
```

Note

Dropping UNIQUE and PRIMARY KEY constraints **also drops all** associated indexes.

DEFAULT VALUE CONCEPTS

At the time of table creation a **default value** can be assigned to a column. When a record is loaded into the table, and the column is left empty, the Oracle engine will automatically load this column with the default value specified. The data type of the default value should match the data type of the column. The **DEFAULT** clause can be used to specify a default value for a column.

Syntax:

```
<ColumnName> <Datatype>(<Size>) DEFAULT <Value>;
```

Example 21:
Create ACCT_MSTR table where the column CURBAL is the number and by default it should be zero. The other column STATUS is a varchar2 and by default it should have character A. (Refer to table definitions in the chapter 6)

```
CREATE TABLE "DBA_BANKSYS"."ACCT_MSTR"("ACCT_NO" VARCHAR2(10),
"SF_NO" VARCHAR2(10), "LF_NO" VARCHAR2(10), "BRANCH_NO" VARCHAR2(10),
"INTRO_CUST_NO" VARCHAR2(10), "INTRO_ACCT_NO" VARCHAR2(10),
"INTRO_SIGN" VARCHAR2(1), "TYPE" VARCHAR2(2), "OPR_MODE" VARCHAR2(2),
"CUR_ACCT_TYPE" VARCHAR2(4), "TITLE" VARCHAR2(30),
"CORP_CUST_NO" VARCHAR2(10), "APLNDT" DATE, "OPNDT" DATE,
"VERI_EMP_NO" VARCHAR2(10), "VERI_SIGN" VARCHAR2(1),
"MANAGER_SIGN" VARCHAR2(1), "CURBAL" NUMBER(8, 2) DEFAULT 0,
"STATUS" VARCHAR2(1) DEFAULT 'A');
```

Output:

```
Table created.
```

Note

- The data type of the default value should match the data type of the column
- Character and date values will be specified in single quotes
- If a column level constraint is defined on the column with a default value, the default value clause must precede the constraint definition

Thus the syntax will be:

```
<ColumnName> <Datatype>(<Size>) DEFAULT <Value> <constraint definition>
```

SELF REVIEW QUESTIONS

FILL IN THE BLANKS

1. Business rules, which are enforced on data being stored in a table, are called _____.
2. If the column was created as _____ Oracle will place a NULL value in the column in the absence of a user-defined value.
3. When a column is defined as not null, then that column becomes a _____ column.
4. The _____ constraint can only be applied at column level.
5. A _____ value can be inserted into the columns of any data type.
6. A single column primary key is called a _____ key.
7. The data held across the primary key column must be _____.
8. _____ keys represent relationships between tables.
9. The table in which the foreign key is defined is called a Foreign table or _____ table.

10. The default behavior of the foreign key can be changed by using the _____ option.
11. _____ constraints must be specified as a logical expression that evaluates either to TRUE or FALSE.
12. In a CHECK constraint the condition must be a _____ expression that can be evaluated using the values in the row being inserted or updated.
13. _____ constraints can be defined using the constraint clause, in the ALTER TABLE command.
14. Dropping UNIQUE and PRIMARY KEY constraints also drops all associated _____.

TRUE OR FALSE

15. Business rules that have to be applied to data are completely System dependent.
16. Constraints super control the data being entered into a table for temporary storage.
17. A NULL value is equivalent to a value of zero.
18. Setting a NULL value is appropriate when the actual value is unknown.
19. A table cannot contain multiple unique keys.
20. Oracle ignores any UNIQUE, FOREIGN KEY, CHECK constraints on a NULL value.
21. A primary key column in a table is an optional column.
22. Standard business rules do not allow multiple entries for the same product.
23. The master table can be referenced in the foreign key definition by using the clause REFERENCES tablename.columnname when defining the foreign key.
24. A CHECK constraint consists of subqueries and sequences.
25. The USER_CONSTRAINTS command displays only the column names, data type, size and the NOT NULL constraint.
26. Drop the constraint using the DROP TABLE command with the DELETE clause.
27. At the time of table creation a default value can be assigned to a column.
28. If a column level constraint is defined on the column with a default value, the default value clause must precede the constraint definition.

HANDS ON EXERCISES

1. Create the tables described below:

Table Name: CLIENT_MASTER
Description: Used to store client information.

Column Name	Data Type	Size	Default	Attributes
CLIENTNO	Varchar2	6		Primary Key / first letter must start with 'C'
NAME	Varchar2	20		Not Null
ADDRESS1	Varchar2	30		

Details for **CLIENT_MASTER** table continued.

Column Name	Data Type	Size	Default	Attributes
ADDRESS2	Varchar2	30		
CITY	Varchar2	15		
PINCODE	Number	8		
STATE	Varchar2	15		
BALDUE	Number	10,2		

Table Name: **PRODUCT_MASTER**
Description: Used to store product information.

Column Name	Data Type	Size	Default	Attributes
PRODUCTNO	Varchar2	6		Primary Key / first letter must start with 'P'
DESCRIPTION	Varchar2	15		Not Null
PROFITPERCENT	Number	4,2		Not Null
UNITMEASURE	Varchar2	10		Not Null
QTYONHAND	Number	8		Not Null
REORDERLVL	Number	8		Not Null
SELLPRICE	Number	8,2		Not Null, Cannot be 0
COSTPRICE	Number	8,2		Not Null, Cannot be 0

Table Name: **SALESMAN_MASTER**
Description: Used to store salesman information working for the company.

Column Name	Data Type	Size	Default	Attributes
SALESMANNO	Varchar2	6		Primary Key / first letter must start with 'S'
SALESMANNAME	Varchar2	20		Not Null
ADDRESS1	Varchar2	30		Not Null
ADDRESS2	Varchar2	30		
CITY	Varchar2	20		
PINCODE	Number	8		
STATE	Varchar2	20		
SALAMT	Number	8,2		Not Null, Cannot be 0
TGTTOGET	Number	6,2		Not Null, Cannot be 0
YTDSALES	Number	6,2		Not Null
REMARKS	Varchar2	60		

Table Name: **SALES_ORDER**
Description: Used to store client's orders.

Column Name	Data Type	Size	Default	Attributes
ORDERNO	Varchar2	6		Primary Key / first letter must start with 'O'
CLIENTNO	Varchar2	6		Foreign Key references ClientNo of Client_Master table
ORDERDATE	Date			Not Null
DELYADDR	Varchar2	25		
SALESMANNO	Varchar2	6		Foreign Key references SalesmanNo of Salesman_Master table
DELYTYPE	Char	1	F	Delivery: part (P) / full (F)
BILLYN	Char	1		
DELYDATE	Date			Cannot be less than Order_Date
ORDERSTATUS	Varchar2	10		Values ('In Process', 'Fulfilled', 'BackOrder', 'Cancelled')

Table Name: SALES_ORDER_DETAILS
Description: Used to store client's orders with details of each product ordered.

Column Name	Data Type	Size	Default	Attributes
ORDERNO	Varchar2	6		Foreign Key references OrderNo of Sales_Order table
PRODUCTNO	Varchar2	6		Foreign Key references ProductNo of Product_Master table
QTYORDERED	Number	8		
QTYDISP	Number	8		
PRODUCTRATE	Number	10,2		

2. Insert the following data into their respective tables:

a) Re-insert the data generated for tables CLIENT_MASTER, PRODUCT_MASTER, SALESMAN_MASTER. Refer to hands-on exercised for **Chapter 07:Interactive SQL-Part I.**

b) Data for Sales_Order table:

OrderNo	ClientNo	OrderDate	SalesmanNo	DelyType	BillYN	DelyDate	OrderStatus
O19001	C00001	12-June-04	S00001	F	N	20-July-02	In Process
O19002	C00002	25-June-04	S00002	P	N	27-June-02	Cancelled
O46865	C00003	18-Feb-04	S00003	F	Y	20-Feb-02	Fulfilled
O19003	C00001	03-Apr-04	S00001	F	Y	07-Apr-02	Fulfilled
O46866	C00004	20-May-04	S00002	P	N	22-May-02	Cancelled
O19008	C00005	24-May-04	S00004	F	N	26-July-02	In Process

c) Data for Sales_Order_Details table:

OrderNo	ProductNo	QtyOrdered	QtyDisp	ProductRate
O19001	P00001	4	4	525
O19001	P07965	2	1	8400
O19001	P07885	2	1	5250
O19002	P00001	10	0	525
O46865	P07868	3	3	3150
O46865	P07885	3	1	5250
O46865	P00001	10	10	525
O46865	P0345	4	4	1050
O19003	P03453	2	2	1050
O19003	P06734	1	1	12000
O46866	P07965	1	0	8400
O46866	P07975	1	0	1050
O19008	P00001	10	5	525
O19008	P07975	5	3	1050

9. INTERACTIVE S

COMPUTATIONS DO

None of the techniques use
done with it.

Computations may includ
Employee_Master table alo
(Salary * 12) is an example

Arithmetic and logical oper

Arithmetic Operators

Oracle allows arithmetic o
Manipulation operations s

- `+` Addition
- `-` Subtraction
- `/` Division

Example 1:
List the fixed deposits he
the fixed deposits are can

Synopsis:

Tables:	FD_DT
Columns:	FD_N
Technique:	Funct Other

Solution:

```
SELECT FD_NO, TYP
       ROUND(AMT
  FROM FD_DTLS
```

Output:

```
FD_NO   TYPE   R
ROUND(AMT+(AMT*
F6      S
F7      S
```

Explanation:
Here, **ROUND(AMT**
column in the table **FD**
AMT, OPNDT and

By default, the Oracl
displaying column ou