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          8.5
7           1098          1829           10
7 rows selected.
```

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 retrieves 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';
```

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 Salesman_Master table to sman_mast.