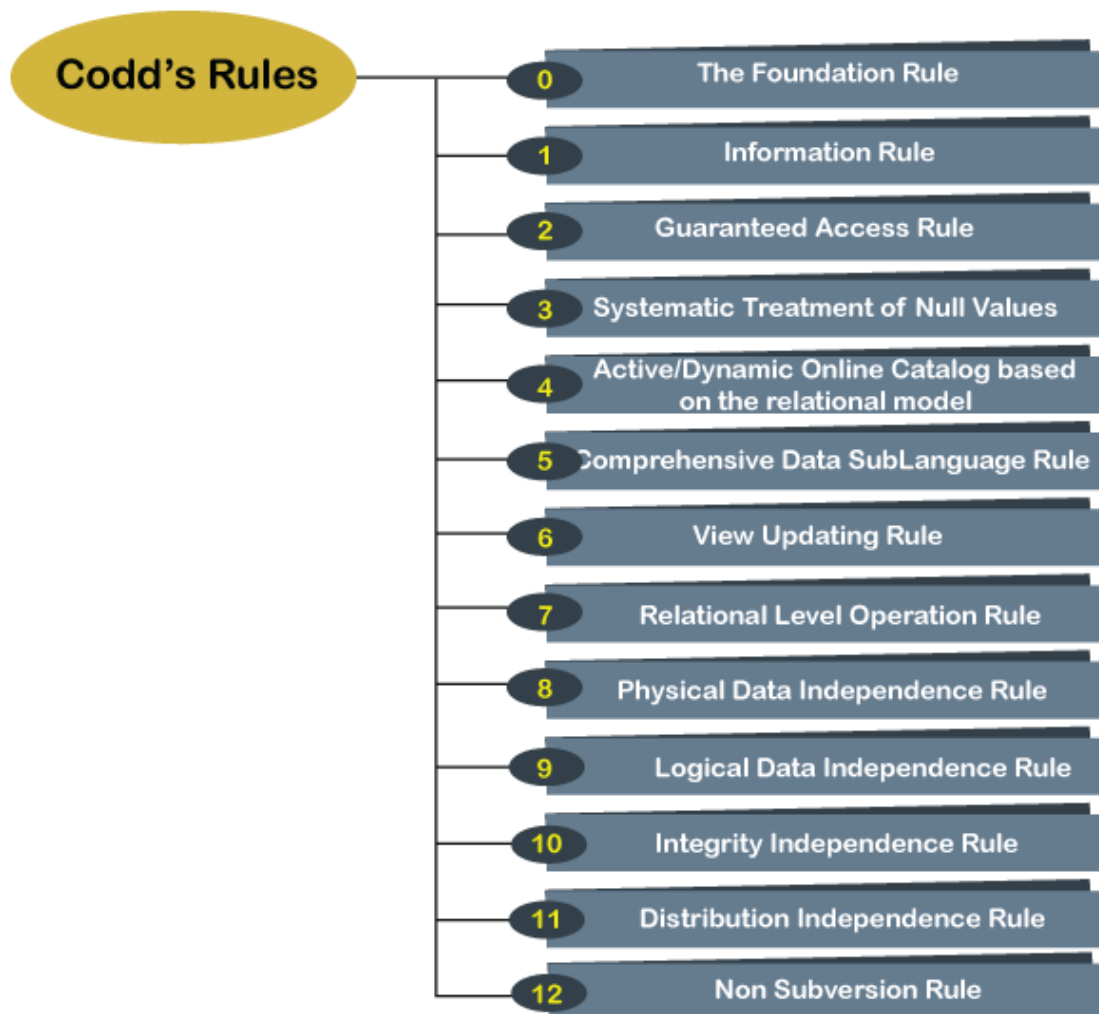


UNIT 1

1.1 12 Codd's Rules

Every database has tables, and constraints cannot be referred to as a rational database system. And if any database has only relational data model, it cannot be a **Relational Database System (RDBMS)**

. So, some rules define a database to be the correct RDBMS. These rules were developed by **Dr. Edgar F. Codd (E.F. Codd)** in **1985**, who has vast research knowledge on the Relational Model of database Systems. Codd presents his 13 rules for a database to test the concept of DBMS against his relational model, and if a database follows the rule, it is called a **true relational database (RDBMS)**. These 13 rules are popular in RDBMS, known as **Codd's 12 rules**.



Rule 0: The Foundation Rule

The database must be in relational form. So that the system can handle the database through its relational capabilities.

Rule 1: Information Rule

A database contains various information, and this information must be stored in each cell of a table in the form of rows and columns.

Rule 2: Guaranteed Access Rule

Every single or precise data (atomic value) may be accessed logically from a relational database using the combination of primary key value, table name, and column name.

Rule 3: Systematic Treatment of Null Values

This rule defines the systematic treatment of Null values in database records. The null value has various meanings in the database, like missing the data, no value in a cell, inappropriate information, unknown data and the primary key should not be null.

Rule 4: Active/Dynamic Online Catalog based on the relational model

It represents the entire logical structure of the descriptive database that must be stored online and is known as a database dictionary. It authorizes users to access the database and implement a similar query language to access the database.

Rule 5: Comprehensive Data SubLanguage Rule

The relational database supports various languages, and if we want to access the database, the language must be the explicit, linear or well-defined syntax, character strings and supports the comprehensive: data definition, view definition, data manipulation, integrity constraints, and limit transaction management operations. If the database allows access to the data without any language, it is considered a violation of the database.

Rule 6: View Updating Rule

All views table can be theoretically updated and must be practically updated by the database systems.

Rule 7: Relational Level Operation (High-Level Insert, Update and delete) Rule

A database system should follow high-level relational operations such as insert, update, and delete in each level or a single row. It also supports union, intersection and minus operation in the database system.

Rule 8: Physical Data Independence Rule

All stored data in a database or an application must be physically independent to access the database. Each data should not depend on other data or an application. If data is updated or the physical structure of the database is changed, it will not show any effect on external applications that are accessing the data from the database.

Rule 9: Logical Data Independence Rule

It is similar to physical data independence. It means, if any changes occurred to the logical level (table structures), it should not affect the user's view (application). For example, suppose a table either split into two tables, or two table joins to create a single table, these changes should not be impacted on the user view application.

Rule 10: Integrity Independence Rule

A database must maintain integrity independence when inserting data into table's cells using the SQL query language. All entered values should not be changed or rely on any external factor or application to maintain integrity. It is also helpful in making the database-independent for each front-end application.

Rule 11: Distribution Independence Rule

The distribution independence rule represents a database that must work properly, even if it is stored in different locations and used by different end-users. Suppose a user accesses the database through an application; in that case, they should not be aware that another user uses particular data, and the data they always get is only located on one site. The end users can access the database, and these access data should be independent for every user to perform the SQL queries.

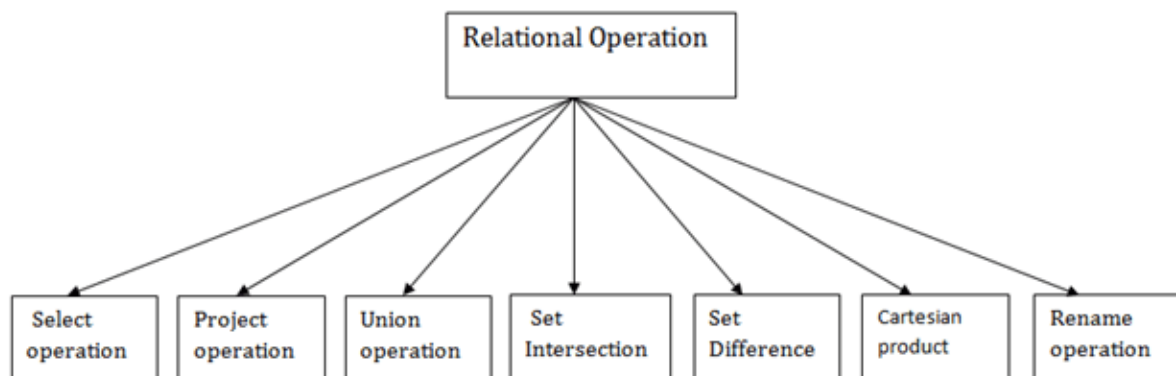
Rule 12: Non Subversion Rule

language to store and manipulate the data in the database. If a system has a low-level or separate language other than SQL to access the database system, it should not subvert or bypass integrity to transform data.

1.2 Relational Algebra

Relational algebra is a procedural query language. It gives a step by step process to obtain the result of the query. It uses operators to perform queries.

Types of Relational operation



1. Select Operation:

- The select operation selects tuples that satisfy a given predicate.
- It is denoted by sigma (σ).

1. Notation: $\sigma p(r)$

Where:

σ is used for selection prediction

r is used for relation

p is used as a propositional logic formula which may use connectors like: AND OR and NOT. These relational can use as relational operators like $=, \neq, \geq, <, >, \leq$.

For example: LOAN Relation

BRANCH_NAME	LOAN_NO	AMOUNT
Downtown	L-17	1000

Redwood	L-23	2000
Perryride	L-15	1500
Downtown	L-14	1500
Mianus	L-13	500
Roundhill	L-11	900
Perryride	L-16	1300

Input:

1. σ BRANCH_NAME="perryride" (LOAN)
2. select * from LOAN where BRANCH_NAME="perryride"

Output:

BRANCH_NAME	LOAN_NO	AMOUNT
Perryride	L-15	1500
Perryride	L-16	1300

2. Project Operation:

- This operation shows the list of those attributes that we wish to appear in the result. Rest of the attributes are eliminated from the table.
- It is denoted by π .

1. Notation: $\pi A_1, A_2, A_n (r)$

Where

A1, A2, A3 is used as an attribute name of relation **r**.

Example: CUSTOMER RELATION

NAME	STREET	CITY
Jones	Main	Harrison
Smith	North	Rye
Hays	Main	Harrison
Curry	North	Rye
Johnson	Alma	Brooklyn
Brooks	Senator	Brooklyn

Input:

1. π NAME, CITY (CUSTOMER)

Output:

NAME	CITY
Jones	Harrison
Smith	Rye
Hays	Harrison
Curry	Rye
Johnson	Brooklyn

Brooks	Brooklyn
--------	----------

3. Union Operation:

- Suppose there are two tuples R and S. The union operation contains all the tuples that are either in R or S or both in R & S.
- It eliminates the duplicate tuples. It is denoted by \cup .

1. Notation: $R \cup S$

A union operation must hold the following condition:

- R and S must have the attribute of the same number.
- Duplicate tuples are eliminated automatically.

Example:

DEPOSITOR RELATION

CUSTOMER_NAME	ACCOUNT_NO
Johnson	A-101
Smith	A-121
Mayes	A-321
Turner	A-176
Johnson	A-273
Jones	A-472
Lindsay	A-284

BORROW RELATION

CUSTOMER_NAME	LOAN_NO
Jones	L-17
Smith	L-23
Hayes	L-15
Jackson	L-14
Curry	L-93
Smith	L-11
Williams	L-17

Input:

1. \prod CUSTOMER_NAME (BORROW) \cup \prod CUSTOMER_NAME (DEPOSITOR)

Output:

CUSTOMER_NAME
Johnson
Smith
Hayes
Turner
Jones

Lindsay
Jackson
Curry
Williams
Mayes

4. Set Intersection:

- Suppose there are two tuples R and S. The set intersection operation contains all tuples that are in both R & S.
- It is denoted by intersection \cap .

1. Notation: $R \cap S$

Example: Using the above DEPOSITOR table and BORROW table

Input:

1. $\prod \text{CUSTOMER_NAME (BORROW)} \cap \prod \text{CUSTOMER_NAME (DEPOSITOR)}$

Output:

CUSTOMER_NAME
Smith
Jones

5. Set Difference:

- Suppose there are two tuples R and S. The set intersection operation contains all tuples that are in R but not in S.

- It is denoted by intersection minus (-).

1. Notation: $R - S$

Example: Using the above DEPOSITOR table and BORROW table

Input:

1. $\Pi \text{ CUSTOMER_NAME (BORROW) - } \Pi \text{ CUSTOMER_NAME (DEPOSITOR)}$

Output:

CUSTOMER_NAME
Jackson
Hayes
Willians
Curry

6. Cartesian product

- The Cartesian product is used to combine each row in one table with each row in the other table. It is also known as a cross product.
- It is denoted by X.

1. Notation: $E \times D$

Example:

EMPLOYEE

EMP_ID	EMP_NAME	EMP_DEPT
1	Smith	A

2	Harry	C
3	John	B

DEPARTMENT

DEPT_NO	DEPT_NAME
A	Marketing
B	Sales
C	Legal

Input:

1. EMPLOYEE X DEPARTMENT

Output:

EMP_ID	EMP_NAME	EMP_DEPT	DEPT_NO	DEPT_NAME
1	Smith	A	A	Marketing
1	Smith	A	B	Sales
1	Smith	A	C	Legal
2	Harry	C	A	Marketing
2	Harry	C	B	Sales
2	Harry	C	C	Legal

3	John	B	A	Marketing
3	John	B	B	Sales
3	John	B	C	Legal

7. Rename Operation:

The rename operation is used to rename the output relation. It is denoted by **rho** (ρ).

Example: We can use the rename operator to rename STUDENT relation to STUDENT1.

1. $\rho(\text{STUDENT1}, \text{STUDENT})$

1.3 TCL Commands in SQL

- In SQL, TCL stands for **Transaction control language**.
- A single unit of work in a database is formed after the consecutive execution of commands is known as a transaction.
- There are certain commands present in SQL known as TCL commands that help the user manage the transactions that take place in a database.
- **COMMIT**, **ROLLBACK** and **SAVEPOINT** are the most commonly used TCL commands in SQL.

1. COMMIT

COMMIT command in SQL is used to save all the transaction-related changes permanently to the disk. Whenever DDL commands such as INSERT, UPDATE and DELETE are used, the changes made by these commands are permanent only after closing the current session. So before closing the session, one can easily roll back the changes made by the DDL commands. Hence, if we want the changes to be saved permanently to the disk without closing the session, we will use the commit command.

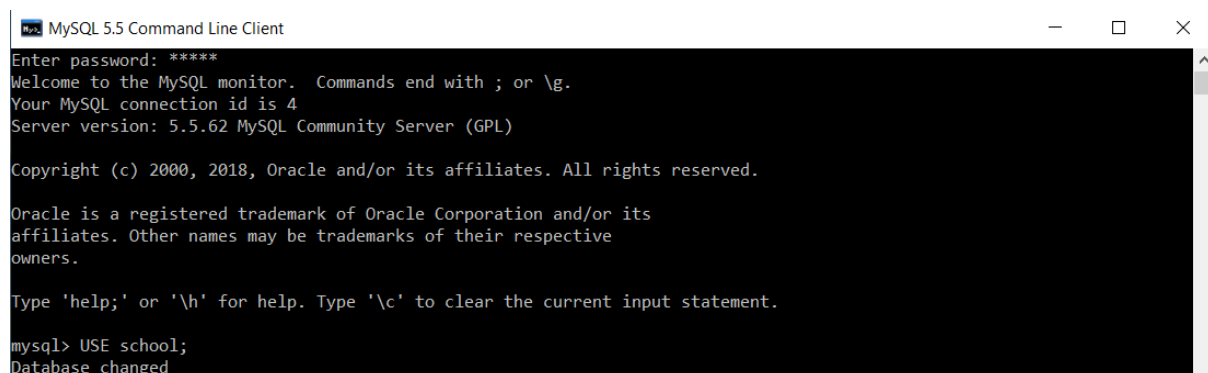
Syntax:

1. **COMMIT;**

Example:

We will select an existing database, i.e., school.

1. `mysql> USE school;`



```
MySQL 5.5 Command Line Client
Enter password: *****
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 4
Server version: 5.5.62 MySQL Community Server (GPL)

Copyright (c) 2000, 2018, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> USE school;
Database changed
```

To create a table named `t_school`, we will execute the following query:

2. SAVEPOINT

We can divide the database operations into parts. For example, we can consider all the insert related queries that we will execute consecutively as one part of the transaction and the delete command as the other part of the transaction. Using the SAVEPOINT command in SQL, we can save these different parts of the same transaction using different names. **For example**, we can save all the insert related queries with the savepoint named INS. To save all the insert related queries in one savepoint, we have to execute the SAVEPOINT query followed by the savepoint name after finishing the insert command execution.

Syntax:

1. SAVEPOINT savepoint_name;

3. ROLLBACK

While carrying a transaction, we must create savepoints to save different parts of the transaction. According to the user's changing requirements, he/she can roll back the transaction to different savepoints. *Consider a scenario*: We have initiated a transaction followed by the table creation and record insertion into the table. After inserting records, we have created a savepoint INS. Then we executed a delete query, but later we thought that mistakenly we had removed the useful record. Therefore in such situations, we have an option of rolling back our transaction. In this case, we have to roll back our transaction using the *ROLLBACK* command to the savepoint INS, which we have created before executing the DELETE query.

Syntax:

1. ROLLBACK TO savepoint_name;

1.4 Data Control Language

DCL commands are used to grant and take back authority from any database user.

Here are some commands that come under DCL:

- Grant
- Revoke

a. Grant: It is used to give user access privileges to a database.

Example

1. GRANT SELECT, UPDATE ON MY_TABLE TO SOME_USER, ANOTHER_USER;
R;

b. Revoke: It is used to take back permissions from the user.

Example

1. REVOKE SELECT, UPDATE ON MY_TABLE FROM USER1, USER2;

UNIT 2

2.1 SQL Data Types

Data types are used to represent the nature of the data that can be stored in the database table. For example, in a particular column of a table, if we want to store a string type of data then we will have to declare a string data type of this column.

Data types mainly classified into three categories for every database.

- String Data types
- Numeric Data types
- Date and time Data types

Data Types in MySQL, SQL Server and Oracle Databases

Oracle Data Types

Oracle String data types

CHAR(size)	It is used to store character data within the predefined length. It can be stored up to 2000 bytes.
NCHAR(size)	It is used to store national character data within the predefined length. It can be stored up to 2000 bytes.
VARCHAR2(size)	It is used to store variable string data within the predefined length. It can be stored up to 4000 byte.
VARCHAR(SIZE)	It is the same as VARCHAR2(size). You can also use VARCHAR(size), but it is suggested to use VARCHAR2(size)
NVARCHAR2(size)	It is used to store Unicode string data within the predefined length. We have to must specify the size of NVARCHAR2 data type. It can be stored up to 4000 bytes.

Oracle Numeric Data Types

NUMBER(p, s)	It contains precision p and scale s. The precision p can range from 1 to 38, and the scale s can range from -84 to 127.
FLOAT(p)	It is a subtype of the NUMBER data type. The precision p can range from 1 to 126.
BINARY_FLOAT	It is used for binary precision(32-bit). It requires 5 bytes, including length byte.
BINARY_DOUBLE	It is used for double binary precision (64-bit). It requires 9 bytes, including length byte.

Oracle Date and Time Data Types

DATE	It is used to store a valid date-time format with a fixed length. Its range varies from January 1, 4712 BC to December 31, 9999 AD.
-------------	---

TIMESTAMP	It is used to store the valid date in YYYY-MM-DD with time hh:mm:ss format.
------------------	---

Oracle Large Object Data Types (LOB Types)

BLOB	It is used to specify unstructured binary data. Its range goes up to $2^{32}-1$ bytes or 4 GB.
BFILE	It is used to store binary data in an external file. Its range goes up to $2^{32}-1$ bytes or 4 GB.
CLOB	It is used for single-byte character data. Its range goes up to $2^{32}-1$ bytes or 4 GB.
NCLOB	It is used to specify single byte or fixed length multibyte national character set (NCHAR) data. Its range is up to $2^{32}-1$ bytes or 4 GB.
RAW(size)	It is used to specify variable length raw binary data. Its range is up to 2000 bytes per row. Its maximum size must be specified.
LONG RAW	It is used to specify variable length raw binary data. Its range up to $2^{31}-1$ bytes or 2 GB, per row.

2.2 ROWID Pseudocolumn

For each row in the database, the **ROWID** pseudocolumn returns the address of the row. Oracle Database rowid values contain information necessary to locate a row:

- The data object number of the object
- The data block in the datafile in which the row resides
- The position of the row in the data block (first row is 0)
- The datafile in which the row resides (first file is 1). The file number is relative to the tablespace.

Usually, a rowid value uniquely identifies a row in the database. However, rows in different tables that are stored together in the same cluster can have the same rowid.

Values of the **ROWID** pseudocolumn have the datatype **ROWID** or **UROWID**. Please refer to "**ROWID Datatype**" and "**UROWID Datatype**" for more information.

Rowid values have several important uses:

- They are the fastest way to access a single row.
- They can show you how the rows in a table are stored.
- They are unique identifiers for rows in a table.

You should not use **ROWID** as the primary key of a table. If you delete and reinsert a row with the Import and Export utilities, for example, then its rowid may change. If you delete a row, then Oracle may reassign its rowid to a new row inserted later. Although you can use the **ROWID** pseudocolumn in the **SELECT** and **WHERE** clause of a query, these pseudocolumn values are not actually stored in the database. You cannot insert, update, or delete a value of the **ROWID** pseudocolumn.

Example This statement selects the address of all rows that contain data for employees in department 20:

```
SELECT ROWID, last_name
FROM employees
WHERE department_id = 20;
```

Selecting from the DUAL Table

DUAL is a table automatically created by Oracle Database along with the data dictionary. **DUAL** is in the schema of the user **SYS** but is accessible by the name **DUAL** to all users. It has one column, **DUMMY**, defined to be **VARCHAR2(1)**, and contains one row with a value **X**. Selecting from the **DUAL** table is useful for computing a constant expression with the **SELECT** statement. Because **DUAL** has only one row, the constant is returned only once. Alternatively, you can select a constant, pseudocolumn, or expression from any table, but the value will be returned as many times as there are rows in the table.

2.3 DATE FUNCTION

SQL **SYSDATE()** Function

Example

Return the current date and time:

```
SELECT SYSDATE();
```

Definition and Usage

The SYSDATE() function returns the current date and time.

Note: The date and time is returned as "YYYY-MM-DD HH:MM:SS" (string) or as YYYYMMDDHHMMSS (numeric).

Syntax

SYSDATE()

More Examples

Example

Return the current date and time + 1:

```
SELECT SYSDATE() + 1;
```

SYSTIMESTAMP Function In Oracle

SYSTIMESTAMP is one of the vital Date/Time functions of Oracle. It is used to get the current system timestamp on the local database. The SYSTIMESTAMP function is supported in the various versions of the Oracle/PLSQL, including, Oracle 12c, Oracle 11g, Oracle 10g and Oracle 9i.

Syntax:

SYSTIMESTAMP

Example:

Select SYSTIMESTAMP

from dual;

Explanation:

The var is a variable that will contain the system's timestamp at the moment of execution.

TO_CHAR Function

This Oracle tutorial explains how to use the Oracle/PLSQL **TO_CHAR** function with syntax and examples.

Syntax

The syntax for the TO_CHAR function in Oracle/PLSQL is:

```
TO_CHAR( value [, format_mask] [, nls_language] )
```

Parameters or Arguments

value

A number or date that will be converted to a string.

format_mask

Optional. This is the format that will be used to convert *value* to a string.

nls_language

Optional. This is the nls language used to convert *value* to a string.

Returns

The TO_CHAR function returns a string value.

Example

Let's look at some Oracle TO_CHAR function examples and explore how to use the TO_CHAR function in Oracle/PLSQL.

Examples with Numbers

For example:

The following are number examples for the TO_CHAR function.

```
TO_CHAR(1210.73, '9999.9')
```

Result: ' 1210.7'

```
TO_CHAR(-1210.73, '9999.9')
```

Result: '-1210.7'

```
TO_CHAR(1210.73, '9,999.99')
```

Result: ' 1,210.73'

```
TO_CHAR(1210.73, '$9,999.00')
```

Result: ' \$1,210.73'

```
TO_CHAR(21, '000099')
```

Result: ' 000021'

Examples with Dates

The following is a list of valid parameters when the TO_CHAR function is used to convert a date to a string. These parameters can be used in many combinations.

Parameter	Explanation
YEAR	Year, spelled out
YYYY	4-digit year
YYY YY Y	Last 3, 2, or 1 digit(s) of year.
IYY IY I	Last 3, 2, or 1 digit(s) of ISO year.
IYYY	4-digit year based on the ISO standard
Q	Quarter of year (1, 2, 3, 4; JAN-MAR = 1).
MM	Month (01-12; JAN = 01).
MON	Abbreviated name of month.
MONTH	Name of month, padded with blanks to length of 9 characters.
RM	Roman numeral month (I-XII; JAN = I).

Parameter	Explanation
WW	Week of year (1-53) where week 1 starts on the first day of the year and continues to the seventh day of the year.
W	Week of month (1-5) where week 1 starts on the first day of the month and ends on the seventh.
IW	Week of year (1-52 or 1-53) based on the ISO standard.
D	Day of week (1-7).
DAY	Name of day.
DD	Day of month (1-31).
DDD	Day of year (1-366).
DY	Abbreviated name of day.
J	Julian day; the number of days since January 1, 4712 BC.
HH	Hour of day (1-12).
HH12	Hour of day (1-12).
HH24	Hour of day (0-23).
MI	Minute (0-59).
SS	Second (0-59).
SSSSS	Seconds past midnight (0-86399).
FF	Fractional seconds.

The following are date examples for the TO_CHAR function.

TO_CHAR(sysdate, 'yyyy/mm/dd')

Result: '2003/07/09'

TO_CHAR(sysdate, 'Month DD, YYYY')

Result: 'July 09, 2003'

TO_CHAR(sysdate, 'FMMonth DD, YYYY')

Result: 'July 9, 2003'

TO_CHAR(sysdate, 'MON DDth, YYYY')

Result: 'JUL 09TH, 2003'

TO_CHAR(sysdate, 'FMMON DDth, YYYY')

Result: 'JUL 9TH, 2003'

```
TO_CHAR(sysdate, 'FMMon ddth, YYYY')  
Result: 'Jul 9th, 2003'
```

You will notice that in some TO_CHAR function examples, the *format_mask* parameter begins with "FM". This means that zeros and blanks are suppressed. This can be seen in the examples below.

```
TO_CHAR(sysdate, 'FMMonth DD, YYYY')  
Result: 'July 9, 2003'
```

```
TO_CHAR(sysdate, 'FMMON DDth, YYYY')  
Result: 'JUL 9TH, 2003'
```

```
TO_CHAR(sysdate, 'FMMon ddth, YYYY')  
Result: 'Jul 9th, 2003'
```

The zeros have been suppressed so that the day component shows as "9" as opposed to "09".

TRUNC (date)

The TRUNC (date) function returns *date* with the time portion of the day truncated to the unit specified by the format model *fmt*. The value returned is always of datatype DATE, even if you specify a different datetime datatype for *date*. If you omit *fmt*, then *date* is truncated to the nearest day.

Examples

The following example truncates a date:

```
SELECT TRUNC(TO_DATE('27-OCT-92','DD-MON-YY'), 'YEAR')  
"New Year" FROM DUAL;  
New Year  
-----  
01-JAN-92
```

ROUND() Function

Example

Round the number to 2 decimal places:

```
SELECT ROUND(235.415, 2) AS RoundValue FROM DUAL;
```

NEXT_DAY

NEXT_DAY returns the date of the first weekday named by *char* that is later than the date *date*. The return type is always DATE, regardless of the datatype of *date*. The argument *char* must be a day of the week in the date language of your session, either the full name or the abbreviation. The minimum number of letters required is the number of letters in the abbreviated version. Any characters immediately following the valid abbreviation are ignored. The return value has the same hours, minutes, and seconds component as the argument *date*.

Examples

This example returns the date of the next Tuesday after February 2, 2001:

```
SELECT NEXT_DAY('02-FEB-2001','TUESDAY') "NEXT DAY"
FROM DUAL;
NEXT DAY
-----
06-FEB-2001
```

LAST_DAY

LAST_DAY returns the date of the last day of the month that contains *date*. The return type is always DATE, regardless of the datatype of *date*.

Examples

The following statement determines how many days are left in the current month.

```
SELECT SYSDATE,
       LAST_DAY(SYSDATE) "Last",
       LAST_DAY(SYSDATE) - SYSDATE "Days Left"
FROM DUAL;
SYSDATE  Last    Days Left
-----
30-MAY-01 31-MAY-01      1
```

The following example adds 5 months to the hire date of each employee to give an evaluation date:

```
SELECT last_name, hire_date, TO_CHAR(
       ADD_MONTHS(LAST_DAY(hire_date), 5)) "Eval Date"
FROM employees;
```

MONTHS_BETWEEN

MONTHS_BETWEEN returns number of months between dates *date1* and *date2*. If *date1* is later than *date2*, then the result is positive. If *date1* is earlier than *date2*, then the result is negative. If *date1* and *date2* are either the same days of the month or both last days of months, then the result is always an integer.

Examples

The following example calculates the months between two dates:

```
SELECT MONTHS_BETWEEN
       (TO_DATE('02-02-1995','MM-DD-YYYY'),
        TO_DATE('01-01-1995','MM-DD-YYYY')) "Months"
FROM DUAL;
Months
-----
1.03225806
```

ADD_MONTHS function

ADD_MONTHS() function returns a date with a given number of months added (date plus integer months). A month is defined by the session parameter NLS_CALENDAR.

Syntax:

```
ADD_MONTHS(date, integer)
```

Parameters:

Name	Description
date	A datetime value or any value that can be implicitly converted to DATE.
integer	An integer or any value that can be implicitly converted to an integer.

Example

```
SELECT ADD_MONTHS(SYSDATE, 7) FROM DUAL;
```

Return value type :

The return type is always DATE, regardless of the datatype of date.

Note: If the date is the last day of the month or if the resulting month has fewer days than the day component of date, then the result is the last day of the resulting month. Otherwise, the result has the same day component as a date.

Oracle ADD_MONTHS() function
<p>Syntax :</p> <p>ADD_MONTHS(date, integer)</p>
<p>Example :</p> <p>ADD_MONTHS('05-JAN-14', 7)</p> <p>01 + 7 = 08 = AUG 05-JAN-14 + 7 = 05-AUG-14</p>
<p>Output : 05-AUG-14</p>

© w3resource.com

Example: Oracle ADD_MONTHS() function

The following statement returns the hire date, month before and after the hire_date in the sample table employees :

Sample table: employees

```
SQL> SELECT hire_date, TO_CHAR(ADD_MONTHS(hire_date, -1), 'DD-MON-YYYY')
"Previous month",
TO_CHAR(ADD_MONTHS(hire_date, 1), 'DD-MON-YYYY') "Next month"
FROM employees
WHERE first_name = 'Lex';
```

Sample Output:

HIRE_DATE	Previous month	Next month
13-JAN-01	13-DEC-2000	13-FEB-2001

2.4 Concepts of Index (Create, drop)

Indexes are **special lookup tables** that the database search engine can use to speed up data retrieval. Simply put, an index is a pointer to data in a table. An index in a database is very similar to an index in the back of a book.

For example, if you want to reference all pages in a book that discusses a certain topic, you first refer to the index, which lists all the topics alphabetically and are then referred to one or more specific page numbers.

An index helps to speed up **SELECT** queries and **WHERE** clauses, but it slows down data input, with the **UPDATE** and the **INSERT** statements. Indexes can be created or dropped with no effect on the data.

Creating an index involves the **CREATE INDEX** statement, which allows you to name the index, to specify the table and which column or columns to index, and to indicate whether the index is in an ascending or descending order.

Indexes can also be unique, like the **UNIQUE** constraint, in that the index prevents duplicate entries in the column or combination of columns on which there is an index.

The CREATE INDEX Command

The basic syntax of a **CREATE INDEX** is as follows.

```
CREATE INDEX index_name ON table_name;
```

Single-Column Indexes

A single-column index is created based on only one table column. The basic syntax is as follows.

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

Unique Indexes

Unique indexes are used not only for performance, but also for data integrity. A unique index does not allow any duplicate values to be inserted into the table. The basic syntax is as follows.

```
CREATE UNIQUE INDEX index_name  
on table_name (column_name);
```

Composite Indexes

A composite index is an index on two or more columns of a table. Its basic syntax is as follows.

```
CREATE INDEX index_name  
on table_name (column1, column2);
```

Whether to create a single-column index or a composite index, take into consideration the column(s) that you may use very frequently in a query's WHERE clause as filter conditions.

Should there be only one column used, a single-column index should be the choice. Should there be two or more columns that are frequently used in the WHERE clause as filters, the composite index would be the best choice.

Implicit Indexes

Implicit indexes are indexes that are automatically created by the database server when an object is created. Indexes are automatically created for primary key constraints and unique constraints.

The DROP INDEX Command

An index can be dropped using SQL **DROP** command. Care should be taken when dropping an index because the performance may either slow down or improve.

The basic syntax is as follows –

```
DROP INDEX index_name;
```

You can check the INDEX Constraint chapter to see some actual examples on Indexes.

When should indexes be avoided?

Although indexes are intended to enhance a database's performance, there are times when they should be avoided.

The following guidelines indicate when the use of an index should be reconsidered.

- Indexes should not be used on small tables.
- Tables that have frequent, large batch updates or insert operations.
- Indexes should not be used on columns that contain a high number of NULL values.
- Columns that are frequently manipulated should not be indexed.

2.5 SQL JOIN

A JOIN clause is used to combine rows from two or more tables, based on a related column between them.

"Orders" table:

OrderID	CustomerID	OrderDate
10308	2	1996-09-18
10309	37	1996-09-19
10310	77	1996-09-20

"Customers" table:

CustomerID	CustomerName	ContactName	Country
1	Alfreds Futterkiste	Maria Anders	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mexico

Notice that the "CustomerID" column in the "Orders" table refers to the "CustomerID" in the "Customers" table. The relationship between the two tables above is the "CustomerID" column.

Then, we can create the following SQL statement (that contains an INNER JOIN), that selects records that have matching values in both tables:

Example

```
SELECT Orders.OrderID, Customer.CustomerName, Orders.OrderDate
FROM Orders
INNER JOIN Customer ON Orders.CustomerID=Customer.CustomerID;
```

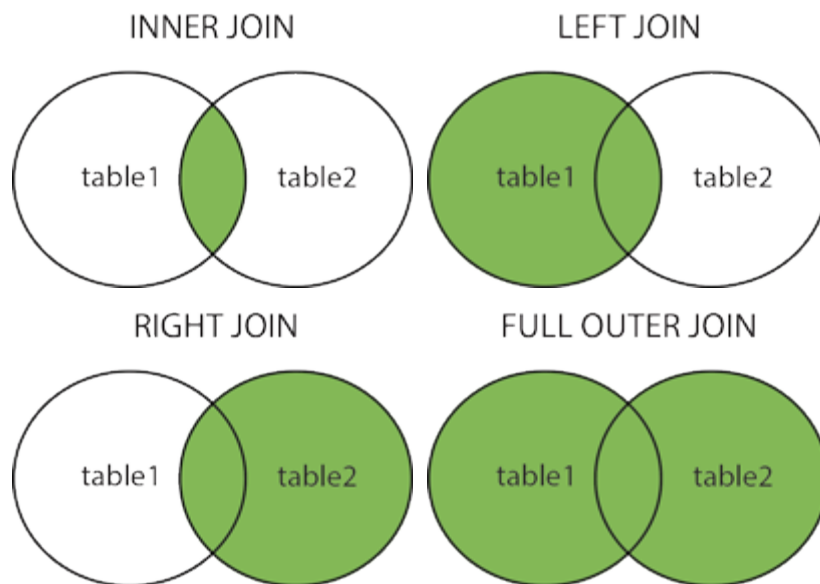
and it will produce something like this:

OrderID	CustomerName	OrderDate
10308	Ana Trujillo Emparedados y helados	9/18/1996
10365	Antonio Moreno Taquería	11/27/1996
10383	Around the Horn	12/16/1996
10355	Around the Horn	11/15/1996
10278	Berglunds snabbköp	8/12/1996

Different Types of SQL JOINS

Here are the different types of the JOINS in SQL:

- (INNER) JOIN: Returns records that have matching values in both tables
- LEFT (OUTER) JOIN: Returns all records from the left table, and the matched records from the right table
- RIGHT (OUTER) JOIN: Returns all records from the right table, and the matched records from the left table
- FULL (OUTER) JOIN: Returns all records when there is a match in either left or right table



Test Yourself With Exercises

Exercise:

Insert the missing parts in the JOIN clause to join the two tables Orders and Customers, using the CustomerID field in both tables as the relationship between the two tables.

SQL INNER JOIN Keyword

The INNER JOIN keyword selects records that have matching values in both tables.

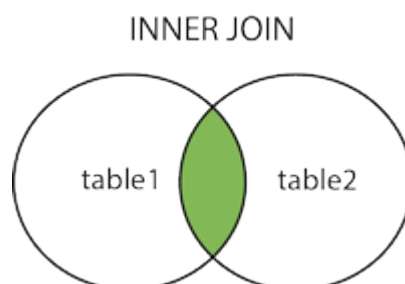
INNER JOIN Syntax

SELECT *column_name(s)*

FROM *table1*

INNER JOIN *table2*

ON *table1.column_name = table2.column_name;*



Demo Database

"Orders" table:

OrderID	CustomerID	EmployeeID	OrderDate	ShipperID
10308	2	7	1996-09-18	3
10309	37	3	1996-09-19	1
10310	77	8	1996-09-20	2

"Customers" table:

Customer ID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico

SQL INNER JOIN Example

The following SQL statement selects all orders with customer information:

Example

```
SELECT Orders.OrderID, Customer.CustomerName
```

```
FROM Orders
```

```
INNER JOIN Customer ON Orders.CustomerID = Customer.CustomerID;
```

Note: The INNER JOIN keyword selects all rows from both tables as long as there is a match between the columns. If there are records in the "Orders" table that do not have matches in "Customers", these orders will not be shown!

JOIN Three Tables

The following SQL statement selects all orders with customer and shipper information:

Example

```
SELECT Orders.OrderID, Customer.CustomerName, Shipper.Shipper_Name
```

```
FROM ((Orders
```

INNER JOIN Customer ON Orders.CustomerID = Customer.CustomerID)

INNER JOIN Shipper ON Orders.Shipper_ID = Shipper.Shipper_ID);

SQL LEFT JOIN Keyword

The LEFT JOIN keyword returns all records from the left table (table1), and the matching records from the right table (table2). The result is 0 records from the right side, if there is no match.

LEFT JOIN Syntax

SELECT *column_name(s)*

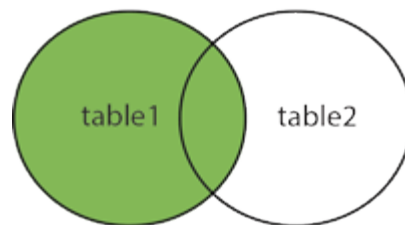
FROM *table1*

LEFT JOIN *table2*

ON *table1.column_name = table2.column_name;*

Note: In some databases LEFT JOIN is called LEFT OUTER JOIN.

LEFT JOIN



Demo Database

"Customers" table:

Customer ID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico

"Orders" table:

OrderID	CustomerID	EmployeeID	OrderDate	ShipperID
10308	2	7	1996-09-18	3

10309	37	3	1996-09-19	1
10310	77	8	1996-09-20	2

SQL LEFT JOIN Example

The following SQL statement will select all customers, and any orders they might have:

Example

```
SELECT Customer.CustomerName, Orders.OrderID
FROM Customer
LEFT JOIN Orders ON Customer.CustomerID = Orders.CustomerID
ORDER BY Customer.CustomerName;
```

```
SELECT Customer.CustomerName, Orders.OrderID
FROM Orders
LEFT JOIN Customer ON Customer.CustomerID = Orders.CustomerID
ORDER BY Customer.CustomerName;
```

SQL RIGHT JOIN Keyword

The RIGHT JOIN keyword returns all records from the right table (table2), and the matching records from the left table (table1). The result is 0 records from the left side, if there is no match.

RIGHT JOIN Syntax

```
SELECT column_name(s)
```

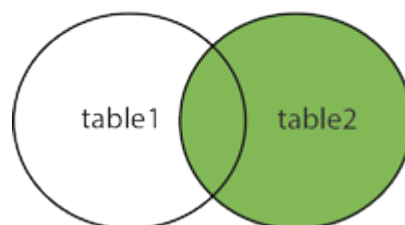
```
FROM table1
```

```
RIGHT JOIN table2
```

```
ON table1.column_name = table2.column_name;
```

Note: In some databases RIGHT JOIN is called RIGHT OUTER JOIN.

RIGHT JOIN



Demo Database

"Orders" table:

OrderID	CustomerID	EmployeeID	OrderDate	ShipperID
10308	2	7	1996-09-18	3

10309	37	3	1996-09-19	1
10310	77	8	1996-09-20	2

"Employees" table:

EmployeeID	LastName	FirstName	BirthDate	Photo
1	Davolio	Nancy	12/8/1968	EmplD1.pic
2	Fuller	Andrew	2/19/1952	EmplD2.pic
3	Leverling	Janet	8/30/1963	EmplD3.pic

SQL RIGHT JOIN Example

The following SQL statement will return all employees, and any orders they might have placed:

Example

```
SELECT Orders.OrderID, Employees.LastName, Employees.FirstName
FROM Orders
RIGHT JOIN Employees ON Orders.EmployeeID = Employees.EmployeeID
ORDER BY Orders.OrderID;
```

SQL FULL OUTER JOIN Keyword

The FULL OUTER JOIN keyword returns all records when there is a match in left (table1) or right (table2) table records.

Tip: FULL OUTER JOIN and FULL JOIN are the same.

FULL OUTER JOIN Syntax

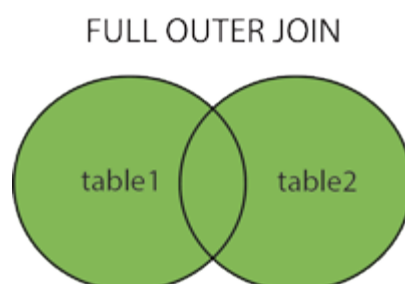
SELECT *column_name(s)*

FROM *table1*

FULL OUTER JOIN *table2*

ON *table1.column_name = table2.column_name*

WHERE *condition*;



Note: FULL OUTER JOIN can potentially return very large result-sets!

Demo Database

"Customers" table:

Customer ID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico

"Orders" table:

OrderID	CustomerID	EmployeeID	OrderDate	ShipperID
10308	2	7	1996-09-18	3
10309	37	3	1996-09-19	1
10310	77	8	1996-09-20	2

SQL FULL OUTER JOIN Example

The following SQL statement selects all customers, and all orders:

```
SELECT Customer.CustomerName, Orders.OrderID
```

```
FROM Customer
```

```
FULL OUTER JOIN Orders ON Customer.CustomerID=Orders.CustomerID
```

```
ORDER BY Customer.CustomerName;
```

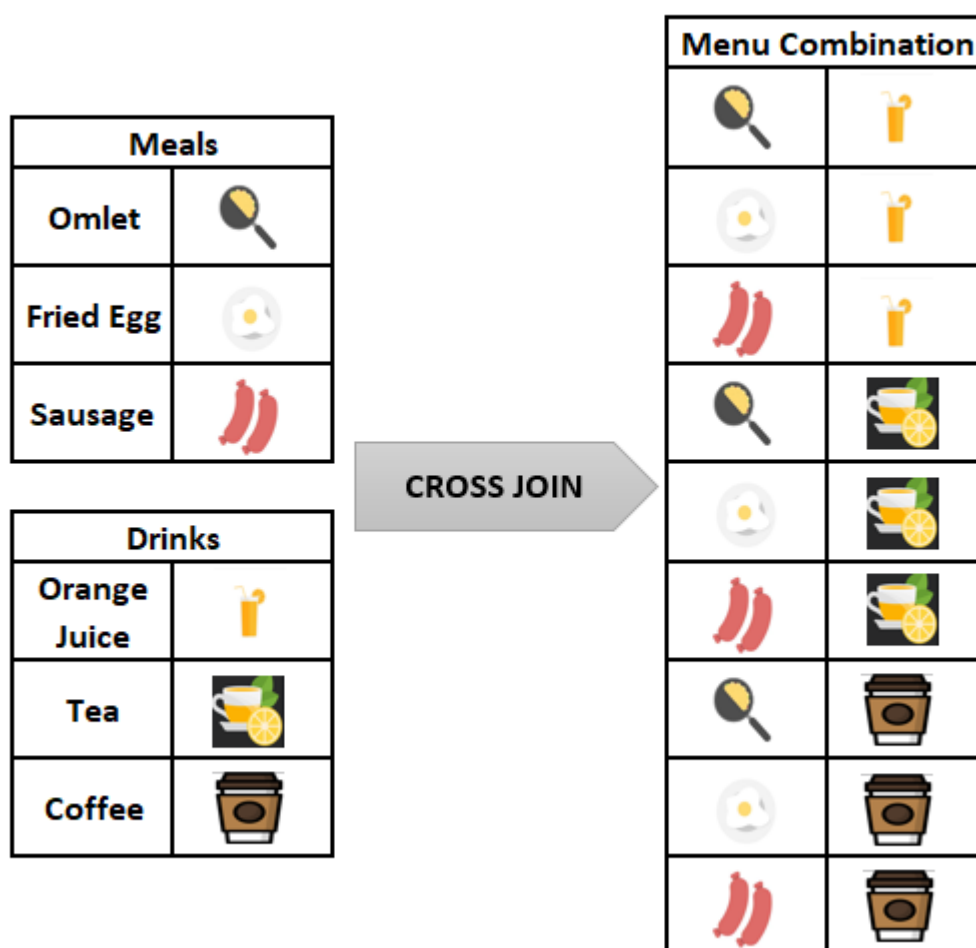
A selection from the result set may look like this:

CustomerName	OrderID
<i>Null</i>	10309
<i>Null</i>	10310
Alfreds Futterkiste	<i>Null</i>
Ana Trujillo Emparedados y helados	10308
Antonio Moreno Taquería	<i>Null</i>

Cross Join

The CROSS JOIN is used to generate a paired combination of each row of the first table with each row of the second table. This join type is also known as cartesian join. Suppose that we are sitting in a coffee shop and we decide to order breakfast. Shortly, we will look at the menu and we will start thinking of which meal and drink combination could be more tastier. Our brain will receive this signal and begin to generate all meal and drink combinations.

The following image illustrates all menu combinations that can be generated by our brain. The SQL CROSS JOIN works similarly to this mechanism, as it creates all paired combinations of the rows of the tables that will be joined.



The main idea of the CROSS JOIN is that it returns the Cartesian product of the joined tables. In the following tip, we will briefly explain the Cartesian product;

Syntax

The syntax of the CROSS JOIN in SQL will look like the below syntax:

```
1 SELECT ColumnName_1,  
2    ColumnName_2,  
3    ColumnName_N
```

```
4FROM [Table_1]
5  CROSS JOIN [Table_2]
```

Or we can use the following syntax instead of the previous one. This syntax does not include the CROSS JOIN keyword; only we will place the tables that will be joined after the FROM clause and separated with a comma.

```
1SELECT ColumnName_1,
2  ColumnName_2,
3  ColumnName_N
4FROM [Table_1],[Table_2]
```

The resultset does not change for either of these syntaxes. In addition, we must notice one point about the CROSS JOIN. Unlike the INNER JOIN, LEFT JOIN and FULL OUTER JOIN, the CROSS JOIN does not require a joining condition.

SQL CROSS JOIN example:

In this example, we will consider the breakfast menu example again, which we mentioned in the earlier part of the article. Firstly, we will create the two-sample tables which contain the drink and meal names. After then, we will populate them with some sample data.

Through the following query, we will perform these two-steps:

```
CREATE TABLE Meals(MealName VARCHAR(15));
CREATE TABLE Drinks(DrinkName VARCHAR(15));
INSERT INTO Drinks
VALUES('Orange Juice');
INSERT INTO Drinks
VALUES ('Tea');
INSERT INTO Drinks
VALUES ('Coffee');
INSERT INTO Meals
VALUES('Omlet');
INSERT INTO Meals
VALUES ('Fried Egg');
INSERT INTO Meals
VALUES ('Sausage');
SELECT *
FROM Meals;
SELECT *
FROM Drinks;
```

100 %	
Results Messages	
	MealName
1	Omlet
2	Fried Egg
3	Sausage

	DrinkName
1	Orange Juice
2	Tea
3	Coffee

The following query will join the Meals and Drinks table with the CROSS JOIN keyword and we will obtain all of the paired combinations of the meal and drink names.

```
SELECT * FROM Meals
CROSS JOIN Drinks
```

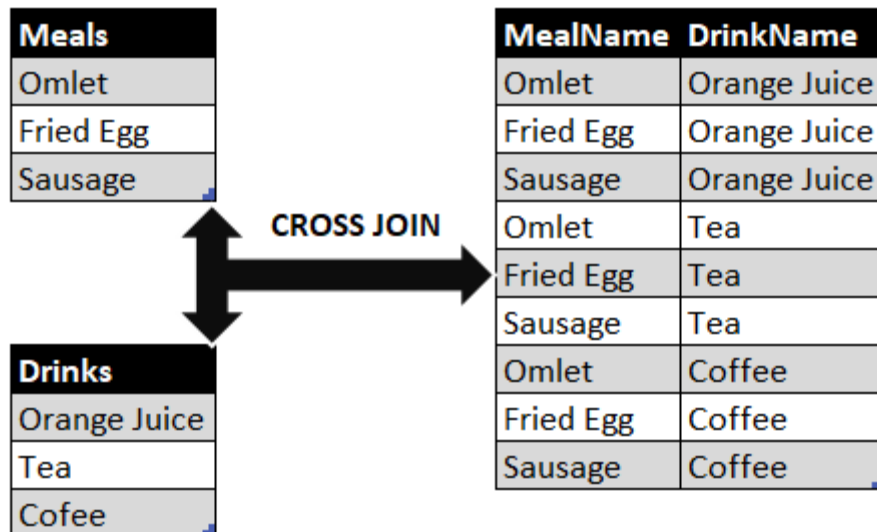
100 %

Results

Messages

	MealName	DrinkName
1	Omlet	Orange Juice
2	Fried Egg	Orange Juice
3	Sausage	Orange Juice
4	Omlet	Tea
5	Fried Egg	Tea
6	Sausage	Tea
7	Omlet	Cofee
8	Fried Egg	Cofee
9	Sausage	Cofee

The below image illustrates the working principle of the CROSS JOIN.



At the same time, we can use the following query in order to obtain the same result set with an alternative syntax without CROSS JOIN.

```
SELECT * FROM Meals
,Drinks
```

100 %

Results Messages

	MealName	DrinkName
1	Omlet	Orange Juice
2	Fried Egg	Orange Juice
3	Sausage	Orange Juice
4	Omlet	Tea
5	Fried Egg	Tea
6	Sausage	Tea
7	Omlet	Coffee
8	Fried Egg	Coffee
9	Sausage	Coffee

3.1 Introduction to PL/SQL (Definition & Block Structure)

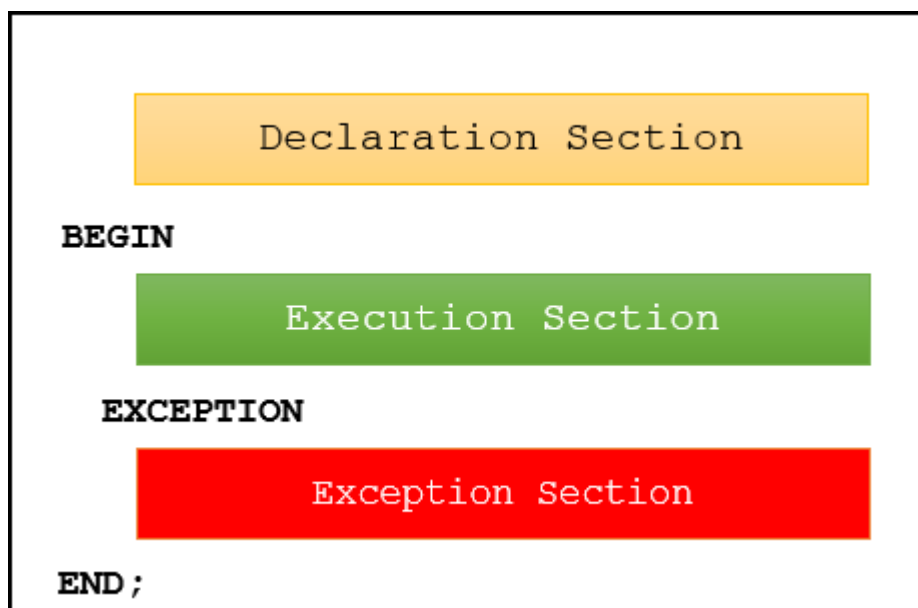
PL/SQL anonymous block

PL/SQL is a block-structured language whose code is organized into blocks. A PL/SQL block consists of three sections: declaration, executable, and exception-handling sections. In a block, the executable section is mandatory while the declaration and exception-handling sections are optional.

A PL/SQL block has a name. [Functions](#) or [Procedures](#) is an example of a named block. A named block is stored into the Oracle Database server and can be reused later.

A block without a name is an anonymous block. An anonymous block is not saved in the Oracle Database server, so it is just for **one-time** use. However, PL/SQL anonymous blocks can be useful for **testing purposes**.

The following picture illustrates the structure of a PL/SQL block:



1) Declaration section

A PL/SQL block has a declaration section where you [declare variables](#), allocate memory for [cursors](#), and define data types.

2) Executable section

A PL/SQL block has an executable section. An executable section starts with the keyword `BEGIN` and ends with the keyword `END`. The executable section must have at least one executable statement, even if it is the [NULL statement](#) which does nothing.

3) Exception-handling section

A PL/SQL block has an exception-handling section that starts with the keyword [EXCEPTION](#). The exception-handling section is where you catch and handle exceptions raised by the code in the execution section.

Note a block itself is an executable statement, therefore you can nest a block within other blocks.

PL/SQL anonymous block example

The following example shows a simple PL/SQL anonymous block with one executable section.

```
BEGIN
    DBMS_OUTPUT.put_line ('Hello World!');
END;
```

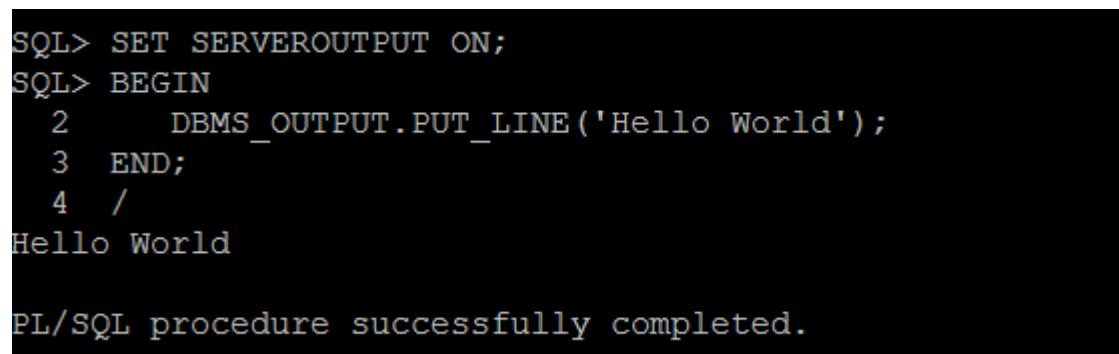
Code language: SQL (Structured Query Language) (sql)

The executable section calls the `DBMS_OUTPUT.PUT_LINE` procedure to display the "Hello World" message on the screen.

Execute a PL/SQL anonymous block using SQL*Plus

Once you have the code of an anonymous block, you can execute it using SQL*Plus, which is a command-line interface for executing SQL statement and PL/SQL blocks provided by Oracle Database.

The following picture illustrates how to execute a PL/SQL block using SQL*Plus:



```
SQL> SET SERVEROUTPUT ON;
SQL> BEGIN
  2     DBMS_OUTPUT.PUT_LINE('Hello World');
  3 END;
  4 /
Hello World

PL/SQL procedure successfully completed.
```

First, connect to the Oracle Database server using a username and password.

Second, turn on the server output using the `SET SERVEROUTPUT ON` command so that the `DBMS_OUTPUT.PUT_LINE` procedure will display text on the screen.

Third, type the code of the block and enter a forward slash (/) to instruct SQL to execute the block. Once you type the forward-slash (/), SQL will execute the block and display the Hello World message on the screen as shown in the illustrations.

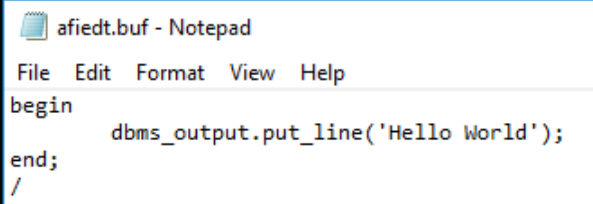
Note that you must execute SET SERVEROUTPUT ON command in every session that you connect to the Oracle Database in order to show the message using the DBMS_OUTPUT.PUT_LINE procedure.

To execute the block that you have entered again, you use / command instead of typing everything from the scratch:

```
SQL> /  
Hello World  
  
PL/SQL procedure successfully completed.
```

If you want to edit the code block, use the edit command. SQL will write the code block to a file and open it in a text editor as shown in the following picture:

```
SQL> edit  
Wrote file afiedt.buf
```



The screenshot shows a Notepad window titled 'afiedt.buf - Notepad'. The menu bar includes File, Edit, Format, View, and Help. The text area contains the following SQL code:

```
begin  
    dbms_output.put_line('Hello World');  
end;  
/
```

You can change the contents of the file like the following:

```
begin  
    dbms_output.put_line('Hello There');  
end;  
/
```

Code language: SQL (Structured Query Language) (sql)

And save and close the file. The contents of the file will be written to the buffer and recompiled.

After that, you can execute the code block again, it will use the new code:


```
SQL> /  
Hello There  
  
PL/SQL procedure successfully completed.
```

Execute a PL/SQL anonymous block using SQL Developer

First, connect to the Oracle Database server using Oracle SQL Developer.

Second, create a new SQL file named `anonymous-block.sql` resided in the `C:\plsql` directory that will store the PL/SQL code.

3.2 Variables, Constants and Data Type

What is an Identifier?

You are identified by your name, hence your name is your identifier. Similarly in programming we use identifiers to name our data which can be any form of data.

PL/SQL Variables

A variable is a reserved memory area for storing the data of a particular datatype. It is an **identifier** which identifies memory locations where data is stored. This memory is reserved at the time of declaration of a variable which is done in the **DECLARE** section of any PL/SQL block.

Syntax for declaration of a variable:

```
Variable_name datatype(size);
```

Let's take a simple example of how we can define a variable in PL/SQL,

```
roll_no NUMBER(2);
```

```
a int;
```

And if we want to assign some value to the variable at the time of declaration itself, the syntax would be,

```
Variable_name datatype(size) NOT NULL:=value;
```

Let's take a simple example for this too,

```
eid NUMBER(2) NOT NULL := 5;
```

In the PL/SQL code above, we have defined a variable with name `eid` which is of datatype `NUMBER` and can hold a number of length `2` bytes, which means it can hold a number upto `99`(because 100 onwards we have 3 digits) and the default value for this variable is `5`.

The keyword **NOT NULL** indicates that `eid` cannot be a blank field.

Here, `:=` is an **assignment operator** used to assign a value to a variable.

Rules for declaring a Variable in PL/SQL

Following are some important rules to keep in mind while defining and using a variable in PL/SQL:

1. A variable name is user-defined. It should begin with a character and can be followed by maximum of 29 characters.
2. Keywords (i.e, reserved words) of PL/SQL cannot be used as variable name.
3. Multiple variables can be declared in a single line provided they must be separated from each other by at least one space and comma.

For eg: `a,b,c int;`

4. Variable names containing two words should not contain space between them. It must be covered by underscore instead.

For eg: `Roll_no`

5. A variable name is case sensitive, which means `a_var` is not same as `A_var`

Example!

```
set serveroutput on;

DECLARE

    a NUMBER(2);

    b NUMBER(2) := 5;

BEGIN

    a := b;

    dbms_output.put_line(a);

END;
```

5

In the example above we have declared two variables **a** and **b** and we have assigned value to the variable **b**, then in the **BEGIN** block, we assign the value of variable **b** to the variable **a** and then print it's value on console.

PL/SQL Constants

Constants are those values which when declared remain fixed throughout the PL/SQL block. For declaring constants, a **constant** keyword is used.

Syntax for declaring constants:

```
Constant_Name  constant  Datatype(size) := value;

school_name constant VARCHAR2(20) := "DPS";
```

In the above code example, constant name is user defined followed by a keyword **constant** and then we have declared its value, which once declared cannot be changed.

Example!

Below we have a simple program to demonstrate the use of constants in PL/SQL,

```
set serveroutput on;

DECLARE

    school_name constant varchar2(20) := 'DPS';

BEGIN

    dbms_output.put_line('I study in ' ||
school_name);

END;
```

I study in DPS

PL/SQL Procedure successfully completed.

PL/SQL Literals

A literal is a value that is expressed by itself and are generally constant. For example, if your name is Alex, then for you **Alex** is a literal(the value which is constant). In other words, the value that is declared as a constant in a program is said to be literal. A literal can be numeric, string or a date.

Type of Literal	Explanation	Example
Numeric	It can be a positive or a negative number.	2,-5,10,-50
String	It is a collection of characters and enclosed in single quotes when used.	'Hello world'
Date	It is a date in DD-MON-YYYY format and always enclosed in single quotes when used in any program.	'25-nov-1995'

Here we have a simple program to demonstrate the use of literals in PL/SQL,

```

set serveroutput on;

DECLARE

    str varchar2(20) := 'Welcome to Udhna';

BEGIN

    dbms_output.put_line(str);

END;
```

3.3 Assigning Values to Variables

A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in PL/SQL has a specific data type, which determines the size and the layout of the variable's memory; the range of values that can be stored within that memory and the set of operations that can be applied to the variable.

The name of a PL/SQL variable consists of a letter optionally followed by more letters, numerals, dollar signs, underscores, and number signs and should not exceed 30 characters. By default, variable names are not case-sensitive. You cannot use a reserved PL/SQL keyword as a variable name.

PL/SQL programming language allows to define various types of variables, such as date time data types, records, collections, etc.

Variable Declaration in PL/SQL

PL/SQL variables must be declared in the declaration section or in a package as a global variable. When you declare a variable, PL/SQL allocates memory for the variable's value and the storage location is identified by the variable name.

The syntax for declaring a variable is –

```
variable_name [CONSTANT] datatype [NOT NULL] [:= | DEFAULT  
initial_value]
```

Where, *variable_name* is a valid identifier in PL/SQL, *datatype* must be a valid PL/SQL data type or any user defined data type. Some valid variable declarations along with their definition are shown below –

```
sales number(10, 2);  
pi CONSTANT double precision := 3.1415;  
name varchar2(25);  
address varchar2(100);
```

When you provide a size, scale or precision limit with the data type, it is called a **constrained declaration**. Constrained declarations require less memory than unconstrained declarations. For example –

```
sales number(10, 2);  
name varchar2(25);  
address varchar2(15);
```

Initializing Variables in PL/SQL

Whenever you declare a variable, PL/SQL assigns it a default value of NULL. If you want to initialize a variable with a value other than the NULL value, you can do so during the declaration, using either of the following –

- The **DEFAULT** keyword
- The **assignment** operator

For example –

```
counter binary_integer := 0;
```



```

        num2 number := 185;
BEGIN
    dbms_output.put_line('Inner Variable num1: ' || num1);
    dbms_output.put_line('Inner Variable num2: ' || num2);
END;
END;
/

```

When the above code is executed, it produces the following result –

```

Outer Variable num1: 95
Outer Variable num2: 85
Inner Variable num1: 195
Inner Variable num2: 185

```

PL/SQL procedure successfully completed.

Assigning SQL Query Results to PL/SQL Variables

You can use the **SELECT INTO** statement of SQL to assign values to PL/SQL variables. For each item in the **SELECT list**, there must be a corresponding, type-compatible variable in the **INTO list**. The following example illustrates the concept. Let us create a table named CUSTOMERS –

```

CREATE TABLE CUSTOMERS (
    ID      INT NOT NULL,
    NAME    VARCHAR (20) NOT NULL,
    AGE     INT NOT NULL,
    ADDRESS CHAR (25),
    SALARY  DECIMAL (18, 2),
    PRIMARY KEY (ID)
);

```

Table Created

Let us now insert some values in the table –

```

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (1, 'Ramesh', 32, 'Ahmedabad', 2000.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (2, 'Khilan', 25, 'Delhi', 1500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (3, 'kaushik', 23, 'Kota', 2000.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (4, 'Chaitali', 25, 'Mumbai', 6500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (5, 'Hardik', 27, 'Bhopal', 8500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

```

```
VALUES (6, 'Komal', 22, 'MP', 4500.00 );
```

The following program assigns values from the above table to PL/SQL variables using the **SELECT INTO** clause of SQL –

```
DECLARE
    c_id customers.id%type := 1;
    c_name customers.name%type;
    c_addr customers.address%type;
    c_sal customers.salary%type;
BEGIN
    SELECT name, address, salary INTO c_name, c_addr, c_sal
    FROM customers
    WHERE id = c_id;
    dbms_output.put_line
    ('Customer ' || c_name || ' from ' || c_addr || ' earns ' ||
    c_sal);
END;
/
```

When the above code is executed, it produces the following result –

```
Customer Ramesh from Ahmedabad earns 2000
```

```
PL/SQL procedure completed successfully
```

3.4 User Defined Record

A **record** is a data structure that can hold data items of different kinds. Records consist of different fields, similar to a row of a database table.

For example, you want to keep track of your books in a library. You might want to track the following attributes about each book, such as Title, Author, Subject, Book ID. A record containing a field for each of these items allows treating a BOOK as a logical unit and allows you to organize and represent its information in a better way.

PL/SQL can handle the following types of records –

- Table-based
- Cursor-based records
- User-defined records

Table-Based Records

The `%ROWTYPE` attribute enables a programmer to create **table-based** and **cursorbased** records.

The following example illustrates the concept of **table-based** records. We will be using the CUSTOMERS table we had created –

```
DECLARE
    customer_rec customers%rowtype;
BEGIN
    SELECT * INTO customer_rec
    FROM customers
    WHERE id = 5;
    dbms_output.put_line('Customer ID: ' || customer_rec.id);
    dbms_output.put_line('Customer Name: ' || customer_rec.name);
    dbms_output.put_line('Customer Address: ' ||
customer_rec.address);
    dbms_output.put_line('Customer Salary: ' ||
customer_rec.salary);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result –

```
Customer ID: 5
Customer Name: Hardik
Customer Address: Bhopal
Customer Salary: 9000
```

PL/SQL procedure successfully completed.

Cursor-Based Records

The following example illustrates the concept of **cursor-based** records. We will be using the CUSTOMERS table we had created and used in the previous chapters –

```

DECLARE
    CURSOR customer_cur is
        SELECT id, name, address
        FROM customers;
    customer_rec customer_cur%rowtype;
BEGIN
    OPEN customer_cur;
    LOOP
        FETCH customer_cur into customer_rec;
        EXIT WHEN customer_cur%notfound;
        DBMS_OUTPUT.put_line(customer_rec.id || ' ' ||
customer_rec.name);
    END LOOP;
END;
/

```

When the above code is executed at the SQL prompt, it produces the following result –

```

1 Ramesh
2 Khilan
3 kaushik
4 Chaitali
5 Hardik
6 Komal

```

PL/SQL procedure successfully completed.

User-Defined Records

PL/SQL provides a user-defined record type that allows you to define the different record structures. These records consist of different fields. Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book –

- Title
- Author
- Subject
- Book ID

Defining a Record

The record type is defined as –

```

TYPE
type_name IS RECORD
( field_name1 datatype1 [NOT NULL] [:= DEFAULT EXPRESSION],
  field_name2 datatype2 [NOT NULL] [:= DEFAULT EXPRESSION],
  ...
  field_nameN datatypeN [NOT NULL] [:= DEFAULT EXPRESSION]);
record-name type_name;

```

The Book record is declared in the following way –

```

DECLARE
TYPE books IS RECORD
(title varchar(50),
 author varchar(50),
 subject varchar(100),
 book_id number);
book1 books;
book2 books;

```

Accessing Fields

To access any field of a record, we use the dot (.) operator. The member access operator is coded as a period between the record variable name and the field that we wish to access. Following is an example to explain the usage of record –

```

DECLARE
    type books is record
        (title varchar(50),
         author varchar(50),
         subject varchar(100),
         book_id number);
    book1 books;
    book2 books;
BEGIN
    -- Book 1 specification
    book1.title := 'C Programming';
    book1.author := 'Nuha Ali ';
    book1.subject := 'C Programming Tutorial';
    book1.book_id := 6495407;
    -- Book 2 specification
    book2.title := 'Telecom Billing';
    book2.author := 'Zara Ali';
    book2.subject := 'Telecom Billing Tutorial';
    book2.book_id := 6495700;

    -- Print book 1 record
    dbms_output.put_line('Book 1 title : ' || book1.title);
    dbms_output.put_line('Book 1 author : ' || book1.author);
    dbms_output.put_line('Book 1 subject : ' || book1.subject);
    dbms_output.put_line('Book 1 book_id : ' || book1.book_id);

    -- Print book 2 record
    dbms_output.put_line('Book 2 title : ' || book2.title);
    dbms_output.put_line('Book 2 author : ' || book2.author);
    dbms_output.put_line('Book 2 subject : ' || book2.subject);
    dbms_output.put_line('Book 2 book_id : ' || book2.book_id);
END;
/

```

When the above code is executed at the SQL prompt, it produces the following result –

```

Book 1 title : C Programming
Book 1 author : Nuha Ali

```

```
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700
```

PL/SQL procedure successfully completed.

Records as Subprogram Parameters

You can pass a record as a subprogram parameter just as you pass any other variable. You can also access the record fields in the same way as you accessed in the above example –

```
DECLARE
    type books is record
        (title varchar(50),
         author varchar(50),
         subject varchar(100),
         book_id number);
    book1 books;
    book2 books;
PROCEDURE printbook (book books) IS
BEGIN
    dbms_output.put_line ('Book title : ' || book.title);
    dbms_output.put_line('Book author : ' || book.author);
    dbms_output.put_line('Book subject : ' || book.subject);
    dbms_output.put_line('Book book_id : ' || book.book_id);
END;

BEGIN
    -- Book 1 specification
    book1.title := 'C Programming';
    book1.author := 'Nuha Ali';
    book1.subject := 'C Programming Tutorial';
    book1.book_id := 6495407;

    -- Book 2 specification
    book2.title := 'Telecom Billing';
    book2.author := 'Zara Ali';
    book2.subject := 'Telecom Billing Tutorial';
    book2.book_id := 6495700;

    -- Use procedure to print book info
    printbook(book1);
    printbook(book2);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result –

```
Book title : C Programming
Book author : Nuha Ali
```

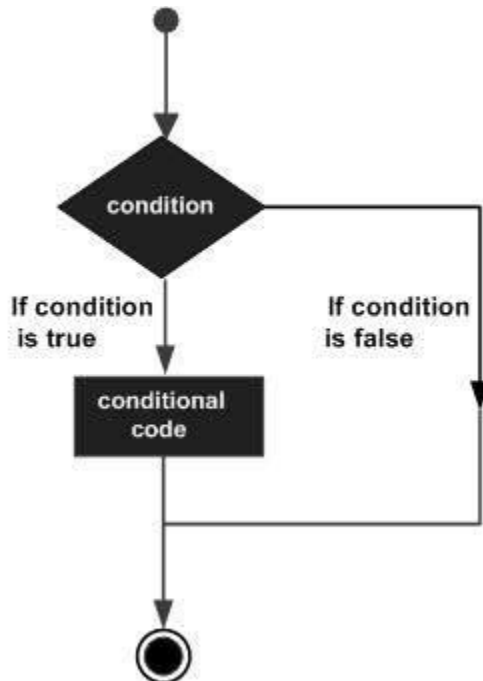
```
Book subject : C Programming Tutorial
Book book_id : 6495407
Book title : Telecom Billing
Book author : Zara Ali
Book subject : Telecom Billing Tutorial
Book book_id : 6495700

PL/SQL procedure successfully completed.
```

3.5 Conditional Statements

Decision-making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general form of a typical conditional (i.e., decision making) structure found in most of the programming languages –



PL/SQL programming language provides following types of decision-making statements. Click the following links to check their detail.

S.No	Statement & Description
1	<p>IF - THEN statement</p> <p>The IF statement associates a condition with a sequence of statements enclosed by the keywords THEN and END IF. If the condition is true, the statements get executed and if the condition is false or NULL then the IF statement does nothing.</p>
2	<p>IF-THEN-ELSE statement</p> <p>IF statement adds the keyword ELSE followed by an alternative sequence of statement. If the condition is false or NULL, then only the alternative sequence of statements get executed. It ensures that either of the sequence of statements is executed.</p>
3	<p>IF-THEN-ELSIF statement</p>

	It allows you to choose between several alternatives.
4	<p>Case statement</p> <p>Like the IF statement, the CASE statement selects one sequence of statements to execute.</p> <p>However, to select the sequence, the CASE statement uses a selector rather than multiple Boolean expressions. A selector is an expression whose value is used to select one of several alternatives.</p>
5	<p>Searched CASE statement</p> <p>The searched CASE statement has no selector, and it's WHEN clauses contain search conditions that yield Boolean values.</p>
6	<p>nested IF-THEN-ELSE</p> <p>You can use one IF-THEN or IF-THEN-ELSIF statement inside another IF-THEN or IF-THEN-ELSIF statement(s).</p>

PL/SQL - IF-THEN Statement

It is the simplest form of the **IF** control statement, frequently used in decision-making and changing the control flow of the program execution.

The **IF statement** associates a condition with a sequence of statements enclosed by the keywords **THEN** and **END IF**. If the condition is **TRUE**, the statements get executed, and if the condition is **FALSE** or **NULL**, then the **IF** statement does nothing.

Syntax

Syntax for **IF-THEN** statement is –

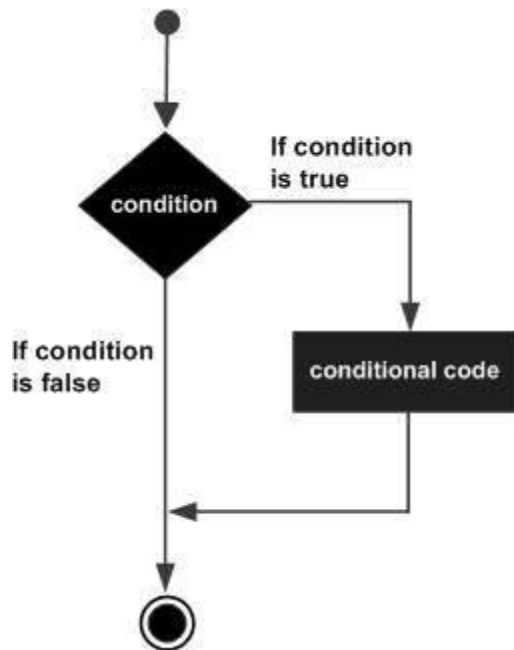
```
IF condition THEN
    S;
END IF;
```

Where *condition* is a Boolean or relational condition and S is a simple or compound statement. Following is an example of the IF-THEN statement –

```
IF (a <= 20) THEN
    c := c+1;
END IF;
```

If the Boolean expression condition evaluates to true, then the block of code inside the **if statement** will be executed. If the Boolean expression evaluates to false, then the first set of code after the end of the **if statement** (after the closing end if) will be executed.

Flow Diagram



Example 1

Let us try an example that will help you understand the concept –

```
DECLARE
    a number(2) := 10;
BEGIN
    a:= 10;
    -- check the boolean condition using if statement
    IF( a < 20 ) THEN
        -- if condition is true then print the following
        dbms_output.put_line('a is less than 20 ');
    END IF;
    dbms_output.put_line('value of a is : ' || a);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result –

```
a is less than 20
value of a is : 10
```

PL/SQL procedure successfully completed.

Example 2

Consider we have a table and few records in the table as we had created in [PL/SQL Variable Types](#)

```

DECLARE
    c_id customers.id%type := 1;
    c_sal customers.salary%type;
BEGIN
    SELECT salary
    INTO c_sal
    FROM customers
    WHERE id = c_id;
    IF (c_sal <= 2000) THEN
        UPDATE customers
        SET salary = salary + 1000
        WHERE id = c_id;
        dbms_output.put_line ('Salary updated');
    END IF;
END;
/

```

When the above code is executed at the SQL prompt, it produces the following result
-

```

Salary updated

PL/SQL procedure successfully completed.

```

The IF...ELSE statement is a control-flow statement that allows you to execute or skip a [statement block](#) based on a specified condition.

The IF statement

The following illustrates the syntax of the IF statement:

```

IF boolean_expression
BEGIN
    { statement_block }
END

```

Code language: SQL (Structured Query Language) (sql)

In this syntax, if the Boolean_expression evaluates to TRUE then the statement_block in the [BEGIN...END](#) block is executed. Otherwise, the statement_block is skipped and the control of the program is passed to the statement after the END keyword.

Note that if the Boolean expression contains a SELECT statement, you must enclose the SELECT statement in parentheses.

The following example first gets the sales amount from the sales.order_items table in the [sample database](#) and then prints out a message if the sales amount is greater than 1 million.

```

BEGIN

```

```
DECLARE @sales INT;
```

```
SELECT  
    @sales = SUM(list_price * quantity)  
FROM  
    sales.order_items i  
    INNER JOIN sales.orders o ON o.order_id = i.order_id  
WHERE  
    YEAR(order_date) = 2018;
```

```
SELECT @sales;
```

```
IF @sales > 1000000  
BEGIN  
    PRINT 'Great! The sales amount in 2018 is greater than  
1,000,000';  
END  
END
```

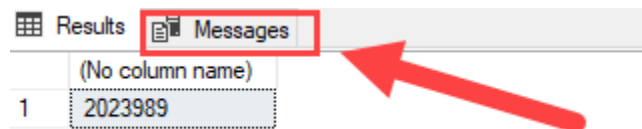
Code language: SQL (Structured Query Language) (sql)

The output of the code block is:

```
Great! The sales amount in 2018 is greater than 1,000,000
```

Code language: SQL (Structured Query Language) (sql)

Note that you have to click the **Messages** tab to see the above output message:



The IF ELSE statement

When the condition in the IF clause evaluates to FALSE and you want to execute another statement block, you can use the ELSE clause.

The following illustrates the IF ELSE statement:

```
IF Boolean_expression  
BEGIN  
    -- Statement block executes when the Boolean expression is  
TRUE  
END  
ELSE  
BEGIN  
    -- Statement block executes when the Boolean expression is  
FALSE  
END
```

Code language: SQL (Structured Query Language) (sql)

Each IF statement has a condition. If the condition evaluates to TRUE then the statement block in the IF clause is executed. If the condition is FALSE, then the code block in the ELSE clause is executed.

See the following example:

```
BEGIN
    DECLARE @sales INT;

    SELECT
        @sales = SUM(list_price * quantity)
    FROM
        sales.order_items i
    INNER JOIN sales.orders o ON o.order_id = i.order_id
    WHERE
        YEAR(order_date) = 2017;

    SELECT @sales;

    IF @sales > 10000000
    BEGIN
        PRINT 'Great! The sales amount in 2018 is greater than
10,000,000';
    END
    ELSE
    BEGIN
        PRINT 'Sales amount in 2017 did not reach 10,000,000';
    END
END
```

Code language: SQL (Structured Query Language) (sql)

In this example:

First, the following statement sets the total sales in 2017 to the @sales variable:

```
SELECT
    @sales = SUM(list_price * quantity)
FROM
    sales.order_items i
INNER JOIN sales.orders o ON o.order_id = i.order_id
WHERE
    YEAR(order_date) = 2017;
```

Code language: SQL (Structured Query Language) (sql)

Second, this statement returns the sales to the output:

```
SELECT @sales;
```

Code language: SQL (Structured Query Language) (sql)

Finally, the IF clause checks if the sales amount in 2017 is greater than 10 million. Because the sales amount is less than that, the statement block in the ELSE clause executes.

```
IF @sales > 10000000
BEGIN
    PRINT 'Great! The sales amount in 2018 is greater than
10,000,000';
END
ELSE
BEGIN
    PRINT 'Sales amount in 2017 did not reach 10,000,000';
END
```

Code language: SQL (Structured Query Language) (sql)

The following shows the output:

```
Sales amount did not reach 10,000,000
```

Code language: SQL (Structured Query Language) (sql)

Nested IF...ELSE

SQL Server allows you to nest an IF...ELSE statement within inside another IF...ELSE statement, see the following example:

```
BEGIN
    DECLARE @x INT = 10,
            @y INT = 20;

    IF (@x > 0)
    BEGIN
        IF (@x < @y)
            PRINT 'x > 0 and x < y';
        ELSE
            PRINT 'x > 0 and x >= y';
        END
    END
END
```

Code language: SQL (Structured Query Language) (sql)

In this example:

First, declare two variables @x and @y and set their values to 10 and 20 respectively:

```
DECLARE @x INT = 10,
        @y INT = 20;
```

Code language: SQL (Structured Query Language) (sql)

Second, the output IF statement check if @x is greater than zero. Because @x is set to 10, the condition (@x > 0) is true. Therefore, the nested IF statement executes.

Finally, the nested IF statement check if @x is less than @y (@x < @y). Because @y is set to 20, the condition (@x < @y) evaluates to true. The PRINT 'x > 0 and x < y'; statement in the IF branch executes.

Here is the output:

```
x > 0 and x < y
```

It is a good practice to not nest an IF statement inside another statement because it makes the code difficult to read and hard to maintain.

SQL CASE Statement

[< Previous](#)[Next >](#)

The SQL CASE Statement

The **CASE** statement goes through conditions and returns a value when the first condition is met (like an if-then-else statement). So, once a condition is true, it will stop reading and return the result. If no conditions are true, it returns the value in the **ELSE** clause.

If there is no **ELSE** part and no conditions are true, it returns NULL.

CASE Syntax

```
CASE
  WHEN condition1 THEN result1
  WHEN condition2 THEN result2
  WHEN conditionN THEN resultN
  ELSE result
END;
```

Demo Database

Below is a selection from the "OrderDetails" table in the Northwind sample database:

OrderDetailID	OrderID	ProductID	Quantity
1	10248	11	1
2	10248	42	1
3	10248	72	5
4	10249	14	9
5	10249	51	4

ADVERTISEMENT

SQL CASE Examples

The following SQL goes through conditions and returns a value when the first condition is met:

Example

```
SELECT OrderID, Quantity,
CASE
    WHEN Quantity > 30 THEN 'The quantity is greater than 30'
    WHEN Quantity = 30 THEN 'The quantity is 30'
    ELSE 'The quantity is under 30'
END AS QuantityText
FROM OrderDetails;
```

[Try it Yourself »](#)

The following SQL will order the customers by City. However, if City is NULL, then order by Country:

Example

```
SELECT CustomerName, City, Country
FROM Customers
ORDER BY
(CASE
    WHEN City IS NULL THEN Country
    ELSE City
END);
```

[Try it Yourself »](#)

iterative Statements in PL/SQL

Iterative control Statements are used when we want to repeat the execution of one or more statements for specified number of times.

There are three types of loops in PL/SQL:

- Simple Loop
- While Loop
- For Loop

1) Simple Loop

A Simple Loop is used when a set of statements is to be executed at least once before the loop terminates. An EXIT condition must be specified in the loop, otherwise the loop will get into an infinite number of iterations. When the EXIT condition is satisfied the process exits from the loop.

General Syntax to write a Simple Loop is

```
:  
LOOP  
    statements;  
    EXIT;  
    {or EXIT WHEN condition;}  
END LOOP;
```

These are the important steps to be followed while using Simple Loop.

- 1) Initialise a variable before the loop body.
- 2) Increment the variable in the loop.
- 3) Use a EXIT WHEN statement to exit from the Loop. If you use a EXIT statement without WHEN condition, the statements in the loop is executed only once.

2) While Loop

A WHILE LOOP is used when a set of statements has to be executed as long as a condition is true. The condition is evaluated at the beginning of each iteration. The iteration continues until the condition becomes false.

The General Syntax to write a WHILE LOOP is:

```
WHILE <condition>  
  LOOP statements;  
END LOOP;
```

Important steps to follow when executing a while loop:

- 1) Initialise a variable before the loop body.
- 2) Increment the variable in the loop.
- 3) EXIT WHEN statement and EXIT statements can be used in while loops but it's not done oftenly.

3) FOR Loop

A FOR LOOP is used to execute a set of statements for a predetermined number of times. Iteration occurs between the start and end integer values given. The counter is always incremented by 1. The loop exits when the counter reaches the value of the end integer.

The General Syntax to write a FOR LOOP is:

```
FOR counter IN val1..val2  
  LOOP statements;  
END LOOP;
```

- val1 - Start integer value.
- val2 - End integer value.

Important steps to follow when executing a while loop:

- 1) The counter variable is implicitly declared in the declaration section, so it's not necessary to declare it explicitly.
- 2) The counter variable is incremented by 1 and does not need to be incremented explicitly.
- 3) EXIT WHEN statement and EXIT statements can be used in FOR loops but it's not done oftenly.

PL/SQL Exit Loop (Basic Loop)

PL/SQL exit loop is used when a set of statements is to be executed at least once before the termination of the loop. There must be an EXIT condition specified in the loop,

otherwise the loop will get into an infinite number of iterations. After the occurrence of EXIT condition, the process exits the loop.

Syntax of basic loop:

1. LOOP
2. **Sequence of** statements;
3. **END** LOOP;

Syntax of exit loop:

1. LOOP
2. statements;
3. EXIT;
4. {or EXIT **WHEN** condition;}
5. **END** LOOP;

Example of PL/SQL EXIT Loop

Let's take a simple example to explain it well:

1. **DECLARE**
2. i NUMBER := 1;

3. **BEGIN**
4. LOOP
5. EXIT **WHEN** i>10;
6. DBMS_OUTPUT.PUT_LINE(i);
7. i := i+1;
8. **END** LOOP;
9. **END**;

After the execution of the above code, you will get the following result:

```
1
2
3
4
5
6
7
8
9
10
```

Note: You must follow these steps while using PL/SQL Exit Loop.

- Initialize a variable before the loop body
- Increment the variable in the loop.
- You should use EXIT WHEN statement to exit from the Loop. Otherwise the EXIT statement without WHEN condition, the statements in the Loop is executed only once.

PL/SQL EXIT Loop Example 2

1. **DECLARE**
2. VAR1 NUMBER;
3. VAR2 NUMBER;
4. **BEGIN**
5. VAR1:=100;
6. VAR2:=1;
7. LOOP

8. DBMS_OUTPUT.PUT_LINE (VAR1*VAR2);
9. IF (VAR2=10) **THEN**
10. EXIT;
11. **END IF**;
12. VAR2:=VAR2+1;
13. **END LOOP**;
14. **END**;

Output:

```
100
200
300
400
500
600
700
800
900
1000
```

PL/SQL While Loop

PL/SQL while loop is used when a set of statements has to be executed as long as a condition is true, the While loop is used. The condition is decided at the beginning of each iteration and continues until the condition becomes false.

Syntax of while loop:

1. WHILE <condition>
2. LOOP statements;
3. **END LOOP**;

Example of PL/SQL While Loop

Let's see a simple example of PL/SQL WHILE loop.

1. **DECLARE**
2. i **INTEGER** := 1;
3. **BEGIN**
4. WHILE i <= 10 LOOP
5. DBMS_OUTPUT.PUT_LINE(i);
6. i := i+1;
7. **END LOOP**;

8. **END;**

After the execution of the above code, you will get the following result:

```
1
2
3
4
5
6
7
8
9
10
```

Note: You must follow these steps while using PL/SQL WHILE Loop.

- Initialize a variable before the loop body.
- Increment the variable in the loop.
- You can use EXIT WHEN statements and EXIT statements in While loop but it is not done often.

PL/SQL WHILE Loop Example 2

1. **DECLARE**
2. VAR1 NUMBER;
3. VAR2 NUMBER;
4. **BEGIN**
5. VAR1:=200;
6. VAR2:=1;
7. WHILE (VAR2 <= 10)
8. LOOP
9. DBMS_OUTPUT.PUT_LINE (VAR1*VAR2);
10. VAR2:=VAR2+1;
11. **END LOOP;**
12. **END;**

Output:

```
200
400
600
800
1000
1200
```

```
1400
1600
1800
2000
```

PL/SQL FOR Loop

PL/SQL for loop is used when you want to execute a set of statements for a predetermined number of times. The loop is iterated between the start and end integer values. The counter is always incremented by 1 and once the counter reaches the value of end integer, the loop ends.

Syntax of for loop:

1. **FOR** counter IN initial_value .. final_value LOOP
2. LOOP statements;
3. **END** LOOP;
 - initial_value : Start integer value
 - final_value : End integer value

PL/SQL For Loop Example 1

Let's see a simple example of PL/SQL FOR loop.

1. **BEGIN**
2. **FOR** k IN 1..10 LOOP
3. *-- note that k was not declared*
4. DBMS_OUTPUT.PUT_LINE(k);
5. **END** LOOP;
6. **END**;

After the execution of the above code, you will get the following result:

```
1
2
3
4
5
6
7
8
9
10
```

Note: You must follow these steps while using PL/SQL WHILE Loop.

- You don't need to declare the counter variable explicitly because it is declared implicitly in the declaration section.
- The counter variable is incremented by 1 and does not need to be incremented explicitly.
- You can use EXIT WHEN statements and EXIT statements in FOR Loops but it is not done often.

PL/SQL For Loop Example 2

1. **DECLARE**
2. VAR1 NUMBER;
3. **BEGIN**
4. VAR1:=10;
5. **FOR** VAR2 IN 1..10
6. LOOP
7. DBMS_OUTPUT.PUT_LINE (VAR1*VAR2);
8. **END** LOOP;
9. **END**;

Output:

```
10
20
30
40
50
60
70
80
90
100
```

PL/SQL For Loop REVERSE Example 3

Let's see an example of PL/SQL for loop where we are using REVERSE keyword.

1. **DECLARE**
2. VAR1 NUMBER;
3. **BEGIN**
4. VAR1:=10;
5. **FOR** VAR2 IN REVERSE 1..10
6. LOOP

7. DBMS_OUTPUT.PUT_LINE (VAR1*VAR2);
8. **END** LOOP;
9. **END**;

Output:

```
100
90
80
70
60
50
40
30
20
10
```

PL/SQL Continue Statement

The continue statement is used to exit the loop from the remainder of its body either conditionally or unconditionally and forces the next iteration of the loop to take place, skipping any codes in between.

The continue statement is not a keyword in Oracle 10g. It is a new feature incorporated in Oracle 11g.

For example: If a continue statement exits a cursor FOR LOOP prematurely then it exits an inner loop and transfer control to the next iteration of an outer loop, the cursor closes (in this context, CONTINUE works like GOTO).

Syntax:

1. **continue**;

Example of PL/SQL continue statement

Let's take an example of PL/SQL continue statement.

1. **DECLARE**
2. x NUMBER := 0;
3. **BEGIN**
4. LOOP -- After CONTINUE statement, control resumes here
5. DBMS_OUTPUT.PUT_LINE ('Inside loop: x = ' || TO_CHAR(x));
6. x := x + 1;

```
7.   IF x < 3 THEN
8.     CONTINUE;
9.   END IF;
10.  DBMS_OUTPUT.PUT_LINE
11.    ('Inside loop, after CONTINUE: x = ' || TO_CHAR(x));
12.  EXIT WHEN x = 5;
13.  END LOOP;
14.
15.  DBMS_OUTPUT.PUT_LINE (' After loop: x = ' || TO_CHAR(x));
16. END;
17./
```

After the execution of above code, you will get the following result:

```
Inside loop:  x = 0
Inside loop:  x = 1
Inside loop:  x = 2
Inside loop, after CONTINUE:  x = 3
Inside loop:  x = 3
Inside loop, after CONTINUE:  x = 4
Inside loop:  x = 4
Inside loop, after CONTINUE:  x = 5
After loop:  x = 5
```

Oracle creates a memory area, known as the context area, for processing an SQL statement, which contains all the information needed for processing the statement; for example, the number of rows processed, etc.

A **cursor** is a pointer to this context area. PL/SQL controls the context area through a cursor. A cursor holds the rows (one or more) returned by a SQL statement. The set of rows the cursor holds is referred to as the **active set**.

You can name a cursor so that it could be referred to in a program to fetch and process the rows returned by the SQL statement, one at a time. There are two types of cursors –

- Implicit cursors
- Explicit cursors

Implicit Cursors

Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement. Programmers cannot control the implicit cursors and the information in it.

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

In PL/SQL, you can refer to the most recent implicit cursor as the **SQL cursor**, which always has attributes such as **%FOUND**, **%ISOPEN**, **%NOTFOUND**, and **%ROWCOUNT**. The SQL cursor has additional attributes, **%BULK_ROWCOUNT** and **%BULK_EXCEPTIONS**, designed for use with the **FORALL** statement. The following table provides the description of the most used attributes –

S.No	Attribute & Description
1	%FOUND Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE.
2	%NOTFOUND The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE.
3	%ISOPEN

	Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.
4	%ROWCOUNT Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement.

Any SQL cursor attribute will be accessed as **sql%attribute_name** as shown below in the example.

Example

We will be using the CUSTOMERS table we had created and used in the previous chapters.

```
Select * from customers;
```

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00

The following program will update the table and increase the salary of each customer by 500 and use the **SQL%ROWCOUNT** attribute to determine the number of rows affected –

```
DECLARE
    total_rows number(2);
BEGIN
    UPDATE customers
    SET salary = salary + 500;
    IF sql%notfound THEN
        dbms_output.put_line('no customers selected');
    ELSIF sql%found THEN
        total_rows := sql%rowcount;
        dbms_output.put_line( total_rows || ' customers selected
');
    END IF;
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result –

```
6 customers selected
```

PL/SQL procedure successfully completed.

If you check the records in customers table, you will find that the rows have been updated –

```
Select * from customers;
```

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2500.00
2	Khilan	25	Delhi	2000.00
3	kaushik	23	Kota	2500.00
4	Chaitali	25	Mumbai	7000.00
5	Hardik	27	Bhopal	9000.00
6	Komal	22	MP	5000.00

Explicit Cursors

Explicit cursors are programmer-defined cursors for gaining more control over the **context area**. An explicit cursor should be defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row.

The syntax for creating an explicit cursor is –

```
CURSOR cursor_name IS select_statement;
```

Working with an explicit cursor includes the following steps –

- Declaring the cursor for initializing the memory
- Opening the cursor for allocating the memory
- Fetching the cursor for retrieving the data
- Closing the cursor to release the allocated memory

Declaring the Cursor

Declaring the cursor defines the cursor with a name and the associated SELECT statement. For example –

```
CURSOR c_customers IS  
  SELECT id, name, address FROM customers;
```

Opening the Cursor

Opening the cursor allocates the memory for the cursor and makes it ready for fetching the rows returned by the SQL statement into it. For example, we will open the above defined cursor as follows –

```
OPEN c_customers;
```

Fetching the Cursor

Fetching the cursor involves accessing one row at a time. For example, we will fetch rows from the above-opened cursor as follows –

```
FETCH c_customers INTO c_id, c_name, c_addr;
```

Closing the Cursor

Closing the cursor means releasing the allocated memory. For example, we will close the above-opened cursor as follows –

```
CLOSE c_customers;
```

Example

Following is a complete example to illustrate the concepts of explicit cursors &minua;

```
DECLARE
  c_id customers.id%type;
  c_name customers.name%type;
  c_addr customers.address%type;
  CURSOR c_customers IS
    SELECT id, name, address FROM customers;
BEGIN
  OPEN c_customers;
  LOOP
    FETCH c_customers INTO c_id, c_name, c_addr;
    EXIT WHEN c_customers%notfound;
    dbms_output.put_line(c_id || ' ' || c_name || ' ' ||
c_addr);
  END LOOP;
  CLOSE c_customers;
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result –

```
1 Ramesh Ahmedabad
2 Khilan Delhi
3 kaushik Kota
4 Chaitali Mumbai
5 Hardik Bhopal
6 Komal MP
```

PL/SQL procedure successfully completed.