Procedural Language/ Structured Query Language (PL/SQL)

Introduction to PLSQL

PL/SQL is an extension of SQL language that combines the data manipulation power of SQL with the processing power of procedural language to create super powerful SQL queries. PL/SQL ensures seamless processing of SQL statements by enhancing the security, portability, and robustness of the Database.

PL/SQL means instructing the compiler 'what to do' through SQL and 'how to do' through its procedural way. Similar to other database languages, it gives more control to the programmers by the use of loops, conditions and object-oriented concepts. The PL/SQL Full form is "Procedural Language extensions to SQL".

Disadvantages of SQL

Disadvantages of SQL:

- SQL doesn’t provide the programmers with a technique of condition checking, looping and branching.
- SQL statements are passed to DB engine one at a time which increases traffic and decreases speed.
- SQL has no facility of error checking during manipulation of data.

Advantages of PL/SQL

PL/SQL has the following advantages −

- SQL is the standard database language and PL/SQL is strongly integrated with SQL. PL/SQL supports both static and dynamic SQL. Static SQL supports DML operations and transaction control from PL/SQL block. In Dynamic SQL, SQL allows embedding DDL statements in PL/SQL blocks.
- PL/SQL allows sending an entire block of statements to the database at one time. This reduces network traffic and provides high performance for the applications.

- PL/SQL gives high productivity to programmers as it can query, transform, and update data in a database.
- PL/SQL saves time on design and debugging by strong features, such as exception handling, encapsulation, data hiding, and object-oriented data types.
- Applications written in PL/SQL are fully portable.
- PL/SQL provides high security level.
- PL/SQL provides access to predefined SQL packages.
- PL/SQL provides support for Object-Oriented Programming.
- PL/SQL provides support for developing Web Applications and Server Pages.

PL/SQL Block Structure

What is PL/SQL block?

In PL/SQL, the code is not executed in single line format, but it is always executed by grouping the code into a single element called Blocks

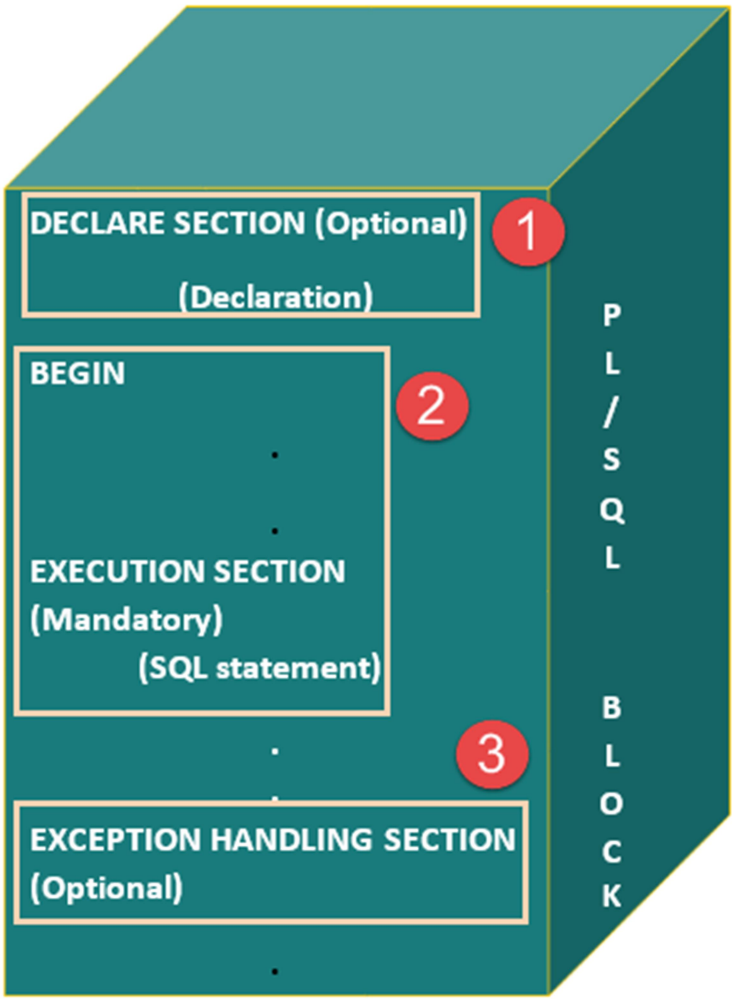
Blocks contain both PL/SQL as well as SQL instruction. All these instruction will be executed as a whole rather than executing a single instruction at a time.

Block Structure

PL/SQL blocks have a pre-defined structure in which the code is to be grouped. Below are different sections of PL/SQL blocks.

1. Declaration section
2. Execution section
3. Exception-Handling section

The below picture illustrates the different PL/SQL block and their section order.



Declaration Section

This is the first section of the PL/SQL blocks. This section is an optional part. This is the section in which the declaration of variables, cursors, exceptions, subprograms, pragma instructions and collections that are needed in the block will be declared. Below are few more characteristics of this part.

- This particular section is optional and can be skipped if no declarations are needed.
- This should be the first section in a PL/SQL block, if present.
- This section starts with the keyword 'DECLARE' for triggers and anonymous block. For other subprograms, this keyword will not be present. Instead, the part after the subprogram name definition marks the declaration section.
- This section should always be followed by execution section.

Execution Section

Execution part is the main and mandatory part which actually executes the code that is written inside it. Since the PL/SQL expects the executable statements from this block this cannot be an

empty block, i.e., it should have at least one valid executable code line in it. Below are few more characteristics of this part.

- This can contain both PL/SQL code and SQL code.
- This can contain one or many blocks inside it as a nested block.
- This section starts with the keyword 'BEGIN'.
- This section should be followed either by 'END' or Exception-Handling section (if present)

Exception-Handling Section:

The exception is unavoidable in the program which occurs at run-time and to handle this Database has provided an Exception-handling section in blocks. This section can also contain PL/SQL statements. This is an optional section of the PL/SQL blocks.

- This is the section where the exception raised in the execution block is handled.
- This section is the last part of the PL/SQL block.
- Control from this section can never return to the execution block.
- This section starts with the keyword 'EXCEPTION'.
- This section should always be followed by the keyword 'END'.

The Keyword 'END' marks the end of PL/SQL block.

PL/SQL Block Syntax

Below is the syntax of the PL/SQL block structure.

Syntax of PL/SQL Block Structure:

```
DECLARE       --optional
        <declarations>
        .
        .
        .
BEGIN         --mandatory
        <executable statements. At least one executable statement is mandatory>
        .
        .
        .
EXCEPTION     --optional
        <exception handler>
        .
        .
        .
END;          --mandatory
/
```

Types of PL/SQL block

PL/SQL blocks are of mainly two types.

1. Anonymous blocks
2. Named Blocks

Anonymous blocks:

Anonymous blocks are PL/SQL blocks which do not have any names assigned to them. They need to be created and used in the same session because they will not be stored in the server as database objects.

Since they need not store in the database, they need no compilation steps. They are written and executed directly, and compilation and execution happen in a single process.

Below are few more characteristics of Anonymous blocks.

- These blocks don't have any reference name specified for them.
- These blocks start with the keyword 'DECLARE' or 'BEGIN'.
- Since these blocks do not have any reference name, these cannot be stored for later purpose. They shall be created and executed in the same session.
- They can call the other named blocks, but call to anonymous block is not possible as it is not having any reference.
- It can have nested block in it which can be named or anonymous. It can also be nested in any blocks.
- These blocks can have all three sections of the block, in which execution section is mandatory; the other two sections are optional.

Named blocks:

Named blocks have a specific and unique name for them. They are stored as the database objects in the server. Since they are available as database objects, they can be referred to or used as long as it is present on the server. The compilation process for named blocks happens separately while creating them as a database objects.

Below are few more characteristics of Named blocks.

- These blocks can be called from other blocks.
- The block structure is same as an anonymous block, except it will never start with the keyword 'DECLARE'. Instead, it will start with the keyword 'CREATE' which instruct the compiler to create it as a database object.
- These blocks can be nested within other blocks. It can also contain nested blocks.

- Named blocks are basically of two types:

1. Procedure
2. Function

Block Data Types

Every constant, variable, and parameter has a **data type** (also called a **type**) that determines its storage format, constraints, valid range of values, and operations that can be performed on it. PL/SQL provides many predefined data types and subtypes, and lets you define your own PL/SQL subtypes.

A **subtype** is a subset of another data type, which is called its **base type**. A subtype has the same valid operations as its base type, but only a subset of its valid values. Subtypes can increase reliability, provide compatibility with ANSI/ISO types, and improve readability by indicating the intended use of constants and variables.

Categories of Predefined PL/SQL Data Types

Data Type Category	Data Description
Scalar	Single values with no internal components.
Composite	Data items that have internal components that can be accessed individually.
Reference	Pointers to other data items.
Large Object (LOB)	Pointers to large objects that are stored separately from other data items, such as text, graphic images, video clips, and sound waveforms.

MySQL Data Type	Oracle Data Type
TINYINT	NUMBER(3, 0)
SMALLINT	NUMBER(5, 0)
MEDIUMINT	NUMBER(7, 0)
INT	NUMBER(10, 0)
INTEGER	NUMBER(10, 0)
BIGINT	NUMBER(19, 0)
FLOAT	FLOAT
DOUBLE	FLOAT (24)
DOULBE PRECISION	FLOAT (24)
REAL	FLOAT (24)
DECIMAL	FLOAT (24)
NUMERIC	NUMBER
DATE	DATE
DATETIME	DATE
TIMESTAMP	NUMBER
TIME	DATE
YEAR	NUMBER
CHAR	CHAR
VARCHAR	VARCHAR2
TINYBLOB	RAW
TINYTEXT	VARCHAR2
BLOB	BLOB, RAW
TEXT	VARCHAR2, CLOB
MEDIUMBLOB	BLOB, RAW
MEDIUMTEXT	RAW, CLOB
LONGBLOB	BLOB, RAW
LONGTEXT	RAW, CLOB
ENUM	VARCHAR2, set to 100 by default
SET	VARCHAR2, set to 100 by default

Block Variable Declaration

In PL/SQL, a variable is named storage location that stores a value of a particular data type.

The value of the variable changes through the program. Before using a variable, you must declare it in the declaration section of a block.

Declaring variables

The syntax for a variable declaration is as follows:

```
variable_name datatype [NOT NULL] [:= initial_value];
```

In this syntax:

- First, specify the name of the variable. The name of the variable should be as descriptive as possible, e.g., l_total_sales, l_credit_limit, and l_sales_revenue.
- Second, choose an appropriate data type for the variable, depending on the kind of value which you want to store, for example, number, character, Boolean, and datetime.

By convention, local variable names should start with l_ and global variable names should have a prefix of g_ .

The following example declares three variables l_total_sales, l_credit_limit, and l_contact_name:

```
DECLARE
    l_total_sales NUMBER(15,2);
    l_credit_limit NUMBER (10,0);
    l_contact_name VARCHAR2(255);
BEGIN
    NULL;
END;
```

Default values

PL/SQL allows you to set a default value for a variable at the declaration time. To assign a default value to a variable, you use the assignment operator (:=) or the DEFAULT keyword.

The following example declares a variable named l_product_name with an initial value 'Laptop':

```
DECLARE
  l_product_name VARCHAR2( 100 ) := 'Laptop';
BEGIN
  NULL;
END;
```

```
It is equivalent to the following block:
```

```
DECLARE
   l_product_name VARCHAR2(100) DEFAULT 'Laptop';
BEGIN
   NULL;
END;
```

In this example, instead of using the assignment operator := , we used the DEFAULT keyword to initialize a variable.

NOT NULL constraint

If you impose the NOT NULL constraint on a value, then the variable cannot accept a NULL value. Besides, a variable declared with the NOT NULL must be initialized with a non-null value. Note that PL/SQL treats a zero-length string as a NULL value.

The following example first declares a variable named l_shipping_status with the NOT NULL constraint. Then, it assigns the variable a zero-length string.

```
DECLARE
  l_shipping_status VARCHAR2( 25 ) NOT NULL := 'Shipped';
BEGIN
  l_shipping_status := '';
END;
```

```
PL/SQL issued the following error:
ORA-06502: PL/SQL: numeric or value error
```

Because the variable l_shipping_status declared with the NOT NULL constraint, it could not accept a NULL value or zero-length string in this case.

Variable assignments

To assign a value to a variable, you use the assignment operator (:=), for example:

```
DECLARE
    l_customer_group VARCHAR2(100) := 'Silver';
BEGIN
    l_customer_group := 'Gold';
    DBMS_OUTPUT.PUT_LINE(l_customer_group);
END;
```

You can assign a value of a variable to another as shown in the following example:

```
DECLARE
    l_business_parter VARCHAR2(100) := 'Distributor';
    l_lead_for VARCHAR2(100);
BEGIN
    l_lead_for := l_business_parter;
    DBMS_OUTPUT.PUT_LINE(l_lead_for);
END;
```

Anchored declarations

Typically, you declare a variable and select a value from a table column to this variable. If the data type of the table column changes, you must adjust the program to make it work with the new type.

PL/SQL allows you to declare a variable whose data type anchor to a table column or another variable. Consider the following example:

```
DECLARE
  l_customer_name customers.name%TYPE;
  l_credit_limit customers.credit_limit%TYPE;
BEGIN
  SELECT
    name, credit_limit
  INTO
    l_customer_name, l_credit_limit
  FROM
```

```
    customers
  WHERE
    customer_id = 38;


  DBMS_OUTPUT.PUT_LINE(l_customer_name || ':' || l_credit_limit );
END;
/
```

In this example:

- First, declare two variables l_customer_name and l_credit_limit whose data type anchors to the name and credit_limit columns respectively, in the declaration section of the block.
- Second, query customer name and credit limit of the customer id 38 and assign these column values to the l_customer_name and l_credit_limit variables in the execution block.
- Third, display the customer name and credit limit.

PL/SQL returned the following output:

Kraft Heinz:500

This example illustrates how to declare variables that anchor to another variable:

```
DECLARE
    l_credit_limit   customers.credit_limit%TYPE;
    l_average_credit l_credit_limit%TYPE;
    l_max_credit     l_credit_limit%TYPE;
    l_min_credit     l_credit_limit%TYPE;
BEGIN
    -- get credit limits
    SELECT
        MIN(credit_limit),
        MAX(credit_limit),
        AVG(credit_limit)
    INTO
        l_min_credit,
```

```
        l_max_credit,
        l_average_credit
    FROM customers;


   SELECT
        credit_limit
    INTO
        l_credit_limit
    FROM
        customers
    WHERE
        customer_id = 100;

    -- show the credits
    dbms_output.put_line('Min Credit: ' || l_min_credit);
    dbms_output.put_line('Max Credit: ' || l_max_credit);
    dbms_output.put_line('Avg Credit: ' || l_average_credit);

    -- show customer credit
    dbms_output.put_line('Customer Credit: ' || l_credit_limit);
END;
```

Here is the output:

```
Min Credit: 100
Max Credit: 5000
Avg Credit: 1894.67
Customer Credit: 1894.67
```

Exception Handling

An exception is an error which disrupts the normal flow of program instructions. PL/SQL provides us the exception block which raises the exception thus helping the programmer to find out the fault and resolve it.

There are two types of exceptions defined in PL/SQL

1. User defined exception.
2. System defined exceptions.

Syntax to write an exception

WHEN exception **THEN**

statement;

```
DECLARE
declarations section;
BEGIN
executable command(s);
EXCEPTION
WHEN exception1 THEN
statement1;
WHEN exception2 THEN
statement2;
[WHEN others THEN]
/* default exception handling code */
END;
```

When other keyword should be used only at the end of the exception handling block as no exception handling part present later will get executed as the control will exit from the block after executing the WHEN OTHERS.

1. **System defined exceptions:**
   These exceptions are predefined in PL/SQL which get raised WHEN certain **database rule is violated.**
   System-defined exceptions are further divided into two categories:
   1. Named system exceptions.
   2. Unnamed system exceptions.

   - **Named system exceptions:** They have a predefined name by the system like ACCESS_INTO_NULL, DUP_VAL_ON_INDEX, LOGIN_DENIED etc. the list is quite big.

So we will discuss some of the most commonly used exceptions:

```
DECLARE
   c_id customers.id%type := 8;
   c_name customerS.Name%type;
   c_addr customers.address%type;
BEGIN
   SELECT  name, address INTO  c_name, c_addr
   FROM customers
   WHERE id = c_id;
   DBMS_OUTPUT.PUT_LINE ('Name: '||  c_name);
   DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);

EXCEPTION
   WHEN no_data_found THEN
      dbms_output.put_line('No such customer!');
   WHEN others THEN
      dbms_output.put_line('Error!');
END;
/
```

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

```
No such customer!

PL/SQL procedure successfully completed.
```

The above program displays the name and address of a customer whose ID is given. Since there is no customer with ID value 8 in our database, the program raises the run-time exception **NO_DATA_FOUND**, which is captured in the **EXCEPTION block**.

Raising Exceptions

Exceptions are raised by the database server automatically whenever there is any internal database error, but exceptions can be raised explicitly by the programmer by using the command **RAISE**. Following is the simple syntax for raising an exception −

```
DECLARE
   exception_name EXCEPTION;
BEGIN
   IF condition THEN
      RAISE exception_name;
   END IF;
EXCEPTION
   WHEN exception_name THEN
   statement;
END;
```

Pre-defined Exceptions

PL/SQL provides many pre-defined exceptions, which are executed when any database rule is violated by a program. For example, the predefined exception NO_DATA_FOUND is raised when a SELECT INTO statement returns no rows. The following table lists few of the important pre-defined exceptions −

Exception	Oracle Error	SQLCODE	Description
ACCESS_INTO_NULL	06530	-6530	It is raised when a null object is automatically assigned a value.
CASE_NOT_FOUND	06592	-6592	It is raised when none of the choices in the WHEN clause of a CASE statement is selected, and there is no ELSE clause.
COLLECTION_IS_NULL	06531	-6531	It is raised when a program attempts to apply collection methods other than EXISTS to an uninitialized nested table or varray, or the program attempts to assign values to the elements of an uninitialized nested table or varray.
DUP_VAL_ON_INDEX	00001	-1	It is raised when duplicate values are attempted to be stored in a column with unique index.
INVALID_CURSOR	01001	-1001	It is raised when attempts are made to make a cursor operation that is not allowed, such as closing an unopened cursor.
INVALID_NUMBER	01722	-1722	It is raised when the conversion of a character string into a number fails because the string does not represent a valid number.
LOGIN_DENIED	01017	-1017	It is raised when a program attempts to log on to the database with an invalid username or password.
NO_DATA_FOUND	01403	+100	It is raised when a SELECT INTO statement returns no

			rows.
NOT_LOGGED_ON	01012	-1012	It is raised when a database call is issued without being connected to the database.
PROGRAM_ERROR	06501	-6501	It is raised when PL/SQL has an internal problem.
ROWTYPE_MISMATCH	06504	-6504	It is raised when a cursor fetches value in a variable having incompatible data type.
SELF_IS_NULL	30625	-30625	It is raised when a member method is invoked, but the instance of the object type was not initialized.
STORAGE_ERROR	06500	-6500	It is raised when PL/SQL ran out of memory or memory was corrupted.
TOO_MANY_ROWS	01422	-1422	It is raised when a SELECT INTO statement returns more than one row.
VALUE_ERROR	06502	-6502	It is raised when an arithmetic, conversion, truncation, or sizeconstraint error occurs.
ZERO_DIVIDE	01476	1476	It is raised when an attempt is made to divide a number by zero.

User-defined Exceptions

PL/SQL allows you to define your own exceptions according to the need of your program. A user-defined exception must be declared and then raised explicitly, using either a RAISE statement or the procedure **DBMS_STANDARD.RAISE_APPLICATION_ERROR**.

The syntax for declaring an exception is –

```
DECLARE
   my-exception EXCEPTION;
```

The following example illustrates the concept. This program asks for a customer ID, when the user enters an invalid ID, the exception **invalid_id** is raised.

```
DECLARE
   c_id customers.id%type := &cc_id;
   c_name customerS.Name%type;
   c_addr customers.address%type;
   -- user defined exception
   ex_invalid_id  EXCEPTION;
BEGIN
   IF c_id <= 0 THEN
      RAISE ex_invalid_id;
   ELSE
      SELECT  name, address INTO  c_name, c_addr
      FROM customers
      WHERE id = c_id;
      DBMS_OUTPUT.PUT_LINE ('Name: '||  c_name);
      DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);
   END IF;

EXCEPTION
   WHEN ex_invalid_id THEN
      dbms_output.put_line('ID must be greater than zero!');
   WHEN no_data_found THEN
      dbms_output.put_line('No such customer!');
   WHEN others THEN
      dbms_output.put_line('Error!');
END;
/
```

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

```
Enter value for cc_id: -6 (let's enter a value -6)

old  2: c_id customers.id%type := &cc_id;

new  2: c_id customers.id%type := -6;

ID must be greater than zero!
```

PL/SQL procedure successfully completed.

NOTE : None of the given code with work in MYSQL as MYSQL follows a different way for Exception Handling

Link : https://dev.mysql.com/doc/refman/8.0/en/condition-handling.html

Most of the DB uses the code that is given in the Notes(Oracle,Microsoft,Sybase) with difference in syntax

Cursors

DB 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

Types of Cursors

Implicit Cursors

Implicit cursors are automatically created by DB 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 DB 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.

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;
```

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

```
DECLARE
   c_id customers.id%type;
   c_name customer.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.

MySQL supports cursors inside stored programs. The syntax is as in embedded SQL. Cursors have these properties:

- Asensitive: The server may or may not make a copy of its result table
- Read only: Not updatable
- Nonscrollable: Can be traversed only in one direction and cannot skip rows

Cursor declarations must appear before handler declarations and after variable and condition declarations.

```
CREATE PROCEDURE curdemo()
BEGIN
  DECLARE done INT DEFAULT FALSE;
  DECLARE a CHAR(16);
  DECLARE b, c INT;
  DECLARE cur1 CURSOR FOR SELECT id,data FROM test.t1;
  DECLARE cur2 CURSOR FOR SELECT i FROM test.t2;
  DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

  OPEN cur1;
  OPEN cur2;

  read_loop: LOOP
    FETCH cur1 INTO a, b;
    FETCH cur2 INTO c;
    IF done THEN
      LEAVE read_loop;
    END IF;
    IF b < c THEN
      INSERT INTO test.t3 VALUES (a,b);
    ELSE
      INSERT INTO test.t3 VALUES (a,c);
    END IF;
  END LOOP;

  CLOSE cur1;
  CLOSE cur2;
END;
```

Functions

Like functions in programming languages, SQL User Defined Functions are routines that accept parameters, perform an action such as a complex calculation, and returns the result of that action as a value. The return value can either be a single scalar value or a result set.

Functions in programming languages are subroutines used to encapsulate frequently performed logic. Any code that must perform the logic incorporated in a function can call the function rather than having to repeat all of the function logic.

SQL supports two types of functions

- *Built-in functions*

  Operate as defined in the Transact-SQL Reference and cannot be modified. The functions can be referenced only in Transact-SQL statements using the syntax defined in the Transact-SQL Reference.

User Defined Functions

Allow you to define your own Transact-SQL functions using the CREATE FUNCTION statement. User Defined Functions use zero or more input parameters, and return a single value. Some User Defined Functions return a single, scalar data value, such as an int, char, or decimal value.

Benefits of User Defined Functions

- *They allow modular programming*

  You can create the function once, store it in the database, and call it any number of times in your program. User Defined Functions can be modified independently of the program source code.

- *They allow faster execution*

  Similar to Stored Procedures, Transact-SQL User Defined Functions reduce the compilation cost of Transact-SQL code by caching the plans and reusing them for repeated executions. This means the user-defined function does not need to be reparsed and reoptimized with each use resulting in much faster execution times. CLR functions offer significant performance advantage over Transact-SQL functions for computational tasks, string manipulation, and business logic. Transact-SQL functions are better suited for data-access intensive logic.

- *They can reduce network traffic*

  An operation that filters data based on some complex constraint that cannot be expressed in a single scalar expression can be expressed as a function. The function can then invoked in the WHERE clause to reduce the number or rows sent to the client.

NO EXAMPLES HERE AS NOTES ARE SAME AS LAST SEMESTER

Procedure

A Stored Procedure is nothing more than prepared SQL code that you save so you can reuse the code over and over again. So if you think about a query that you write over and over again, instead of having to write that query each time you would save it as a Stored Procedure and then just call the Stored Procedure to execute the SQL code that you saved as part of the Stored Procedure.

In addition to running the same SQL code over and over again you also have the ability to pass parameters to the Stored Procedure, so depending on what the need is, the Stored Procedure can act accordingly based on the parameter values that were passed.

Stored Procedures can also improve performance. Many tasks are implemented as a series of SQL statements. Conditional logic applied to the results of the first SQL statements determine which subsequent SQL statements are executed. If these SQL statements and conditional logic are written into a Stored Procedure, they become part of a single execution plan on the server. The results do not need to be returned to the client to have the conditional logic applied; all of the work is done on the server.

Benefits of Stored Procedures

- *Precompiled execution*

  SQL Server compiles each Stored Procedure once and then reutilizes the execution plan. This results in tremendous performance boosts when Stored Procedures are called repeatedly.

- *Reduced client/server traffic*

  If network bandwidth is a concern in your environment then you'll be happy to learn that Stored Procedures can reduce long SQL queries to a single line that is transmitted over the wire.

- *Efficient reuse of code and programming abstraction*

  Stored Procedures can be used by multiple users and client programs. If you utilize them in a planned manner then you'll find the development cycle requires less time.

- *Enhanced security controls*

  You can grant users permission to execute a Stored Procedure independently of underlying table permissions.

NO EXAMPLES HERE AS NOTES ARE SAME AS LAST SEMESTER

Triggers

A trigger is a database object that is associated with the table, it will be activated when a defined action is executed for the table. The CREATE TRIGGER statement allows you to create a new trigger that is fired automatically whenever an event such as INSERT, DELETE, or UPDATE occurs against a table.

The following illustrates the syntax of the CREATE TRIGGER statement:

```
CREATE TRIGGER [schema_name.]trigger_name
ON table_name
AFTER  {[INSERT],[UPDATE],[DELETE]}
[NOT FOR REPLICATION]
AS
{sql_statements}
```

In this syntax:

- The schema_name is the name of the schema to which the new trigger belongs. The schema name is optional.
- The trigger_name is the user-defined name for the new trigger.
- The table_name is the table to which the trigger applies.
- The event is listed in the AFTER clause. The event could be INSERT, UPDATE, or DELETE. A single trigger can fire in response to one or more actions against the table.
- The NOT FOR REPLICATION option instructs SQL Server not to fire the trigger when data modification is made as part of a replication process.
- The sql_statements is one or more Transact-SQL used to carry out actions once an event occurs.