

UNIT : 1

Introduction to Procedural SQL



Unit Covers

- Introduction RDBMS: E.F. Codd's Rule, DBMS vs. RDBMS
- Language Fundamentals
- Advanced Data Types
- Conditional Statements
- Looping Statements
- Exceptional Handling



RDBMS

- Relational Database Management System
- For defining a fully relational database
 - Dr. E. F. Codd's 12 rules is used.
 - **Codd's twelve rules** are a set of thirteen rules (numbered zero to twelve) proposed by **Edgar F. Codd**, a pioneer of the relational model for databases.
 - **Codd's Rule Designed** to define what is required from a database management system in order for it to be considered *relational*.



Dr. E.F. Codd's 12 Rule

- Rule 0 : Foundation Rule

- For a system to qualify as a relational database management system (RDBMS), that system must use its *relational* facilities (Relation between table) to *manage* the *database*.

- Rule 1: The Information Rule

- All data should be presented to the user in table form.

- Rule 2: Guaranteed Access Rule

- All data should be accessible without ambiguity.
- This can be achieved through a combination of the table name, primary key, and column name.



- **Rule 3: Systematic Treatment of Null Values:**

- A field should be allowed to remain empty.
- This involves the support of a null value, which is distinct from an empty string or a number with a value of zero.
- Of course, this can't apply to primary keys.

- **Rule 4: Active online catalog based on the relational model:**

- The system must support an online, inline, relational catalog that is accessible to authorized users by means of their regular query language.
- Users must be able to access the database's structure (catalog) using the **same query language** that they use to access the database's data.



— Rule 5: Comprehensive Data Sublanguage

- The database must support at least one clearly defined language that includes functionality for data definition, data manipulation, data integrity, and database transaction control.
- All commercial relational databases use forms of the standard SQL (Structured Query Language) as their supported comprehensive language.
- Supported Language :
 - Data definition
 - View definition
 - Data manipulation (interactive and by program)
 - Integrity constraints
 - Authorization
 - Transaction boundaries (begin, commit, and rollback).



– Rule 6: View Updating Rule

- **View** : Data can be presented to the user in different logical combinations, called views.
- Each view should support the same full range of data manipulation that direct-access to a table has available.

– Rule 7: High-level Insert, Update, and Delete

- The system must support **set-at-a-time** *insert*, *update*, and *delete* operators.
- This means that data can be retrieved from a relational database in sets constructed of data from multiple rows and/or multiple tables.



- **Rule 8: Physical Data Independence**

- Changes to the physical level (how the data is stored, whether in arrays or linked lists etc.) must not require a change to an application based on the structure.

- **Rule 9: Logical Data Independence**

- Changes to the logical level (tables, columns, rows, and so on) must not require a change to an application based on the structure.
- Logical data independence is **more difficult to achieve** than physical data independence



– Rule 10: Integrity Independence

- Integrity constraints must be specified separately from application programs and stored in the structure/catalog.
- No component of a primary key can have a null value. (see rule 3)
If a foreign key is defined in one table, any value in it must exist as a primary key in another table.
- Key and Check constraints, trigger etc should be stored in Data Dictionary.



— Rule 11: Distribution Independence

- A user should be totally unaware of whether or not the database is distributed (whether parts of the database exist in multiple locations).
- A variety of reasons make this rule difficult to implement;

— Rule 12: Non subversion Rule

- *If a relational system has or supports a low-level (single-record-at-a-time) language, that low-level language **cannot** be used to subvert or bypass the integrity rules or constraints expressed in the higher-level (multiple-records-at-a-time) relational language.*



SQL PL

- SQL PL stands for SQL **procedural language** with extension SQL.
- The SQL Procedural Language (SQL PL) is a language that consists of statements and language elements that can be used to implement procedural logic in SQL statements.
- SQL PL provides
 - Statements for declaring variables
 - Condition handlers
 - Assigning values to variables
 - For implementing procedural logic.



Datatypes

Data type	Explanation	Examples of corresponding values
INT, INTEGER	A 32-bit integer (whole number). Values can be from approximately -2.1 billion to +2.1 billion. If unsigned, the value can reach about 4.2 billion, but negative numbers are not allowed.	123,345 -2,000,000,000
BIGINT	A 64-bit integer (whole number). Values can be from approximately -9 million trillion to +9 million trillion or from 0 to 18 million trillion if unsigned.	9,000,000,000,000,000,000 -9,000,000,000,000,000,000
FLOAT	A 32-bit floating-point number. Values can range from about -1.7e38 to 1.7e38 for signed numbers or 0 to 3.4e38 if unsigned.	0.0000000000000002 17897.890790 -345.8908770 1.7e21
DOUBLE	A 64-bit floating-point number. The value range is close to infinite (1.7e308).	1.765e203 -1.765e100
DECIMAL(<i>precision</i> , <i>scale</i>) NUMERIC(<i>precision</i> , <i>scale</i>)	A fixed-point number. Storage depends on the precision, as do the possible numbers that can be stored. NUMERICs are typically used where the number of decimals is important, such as for currency.	78979.00 -87.50 9.95
DATE	A calendar date, with no specification of time.	'1999-12-31'

Datatypes

Data type	Explanation	Examples of corresponding values
DATETIME	A date and time, with resolution to a particular second.	'1999-12-31 23:59:59'
CHAR(<i>length</i>)	A fixed-length character string. The value will be right-padded up to the length specified. A maximum of 255 bytes can be specified for the length.	'hello world'
VARCHAR(<i>length</i>)	A variable-length string up to 64K in length.	'Hello world'
BLOB, TEXT	Up to 64K of data, binary in the case of BLOB, or text in the case of TEXT.	Almost anything imaginable
LOB, LONGTEXT	Longer versions of the BLOB and TEXT types, capable of storing up to 4GB of data.	Almost anything imaginable, but a lot more than you would have imagined for BLOB or TEXT

Variable Declaration

DECLARE *variable_name* [,var_name...] datatype [DEFAULT value];

- DECLARE **l_int1** INT DEFAULT -2000000;
- DECLARE **l_int2** INT UNSIGNED DEFAULT 4000000;
- DECLARE **l_bigint1** BIGINT DEFAULT 4000000000000000000;
- DECLARE **l_float** FLOAT DEFAULT 1.8e8;
- DECLARE **l_double** DOUBLE DEFAULT 2e45;
- DECLARE **l_numeric** NUMERIC(8,2) DEFAULT 9.95;
- DECLARE **l_date** DATE DEFAULT '1999-12-31';
- DECLARE **l_datetime** DATETIME DEFAULT '1999-12-31 23:59:59';



Examples of variable declarations

- DECLARE **l_int1** INT DEFAULT -2000000;
- DECLARE **l_int2** INT UNSIGNED DEFAULT 4000000;
- DECLARE **l_bigint1** BIGINT DEFAULT 4000000000000000000;
- DECLARE **l_float** FLOAT DEFAULT 1.8e8;
- DECLARE **l_double** DOUBLE DEFAULT 2e45;
- DECLARE **l_numeric** NUMERIC(8,2) DEFAULT 9.95;
- DECLARE **l_date** DATE DEFAULT '1999-12-31';
- DECLARE **l_datetime** DATETIME DEFAULT '1999-12-31 23:59:59';



Assigning Values to Variables

SET variable_name = expression [,variable_name = expression ...]

Delimiter \$\$

```
CREATE PROCEDURE no_set_stmt( )
```

```
BEGIN
```

```
    DECLARE i INTEGER;
```

```
    SET i=1;
```

```
END $$
```

Delimiter ;



Comment

- Comments

- I. Two dashes -- followed by a space create a comment that continues until the end of the current line. We'll call these *single-line comments*. (Single line)
- II. C-style comments commence with /* and terminate with */. We'll call these *multiline comments*. (More than one line)



Operator

- Mathematical Operators

Operator	Description	Example
+	Addition	SET var1=2+2; → 4
-	Subtraction	SET var2=3-2; → 1
*	Multiplication	SET var3=3*2; → 6
/	Division	SET var4=10/3; → 3.3333
DIV	Integer division	SET var5=10 DIV 3; → 3
%	Modulus	SET var6=10%3 ; → 1

Comparison Operators

Table 3-3. Comparison operators

Operator	Description	Example	Example result
>	Is greater than	1>2	FALSE
<	Is less than	2<1	FALSE
<=	Is less than or equal to	2<=2	TRUE
>=	Is greater than or equal to	3>=2	TRUE
BETWEEN	Value is between two values	5 BETWEEN 1 AND 10	TRUE
NOT BETWEEN	Value is not between two values	5 NOT BETWEEN 1 AND 10	FALSE
IN	Value is in a list	5 IN (1,2,3,4)	FALSE
NOT IN	Value is not in a list	5 NOT IN (1,2,3,4)	TRUE
=	Is equal to	2=3	FALSE
<>, !=	Is not equal to	2<>3	FALSE
<=>	Null safe equal (returns TRUE if both arguments are NULL)	NULL<=>NULL	TRUE
LIKE	Matches a simple pattern	"Guy Harrison" LIKE "Guy%"	TRUE
REGEXP	Matches an extended regular expression	"Guy Harrison" REGEXP "[Gg]reg"	FALSE
IS NULL	Value is NULL	0 IS NULL	FALSE
IS NOT NULL	Value is not NULL	0 IS NOT NULL	TRUE

Structure of a Block

- A block consists of various types of declarations
 - variables, cursors, handlers
 - program code (e.g., assignments, conditional statements, loops)

[label:] BEGIN

variable and condition declarations

cursor declarations

handler declarations

program code

END [label];



Conditional Control

- Conditional control—or “flow of control”—statements allow you to execute code based on the value of some expression.
- Two conditional control statements: **IF** and **CASE**.
- The four forms of the IF statement are:
 - **IF-THEN**
 - **IF-THEN-ELSE**
 - **IF-THEN-ELSEIF-ELSE [Else if Ladder]**



Simple If

IF *expression* THEN
 statements

END IF;



Example

emp (id, name, salary)		
Id	Name	Salary
1	Jiya	50000
2	Ashavi	20000

- ❑ Write a procedure that gives the increment of Rs.1000 to employee having id 2 if his salary is >40000.


```
create procedure upsal()  
begin  
    declare c int;  
    select salary into c from emp where id=2;  
    if(c>40000) then  
        update emp set salary=c+1000 where  
        id=2;  
    end if;  
end;
```



IF-THEN-ELSE

IF *expression* THEN

*statements that execute if the expression is
TRUE*

ELSE

*statements that execute if the expression is
FALSE or NULL*

END IF;




EXAMPLE

Write a procedure that gives the increment of Rs.1000 to employee having id 2 if his salary is >40000 other wise give bonus of Rs.500.



Answer

```
create procedure upsal()  
begin  
    declare c int;  
    select salary into c from emp where id=2 ;  
    if(c>40000) then  
        set c=c+1000;  
    else  
        set c=c+500;  
    end if;  
    update emp set salary=c where id=2;  
end
```



IF-THEN-ELSEIF-ELSE [Else if Ladder]

IF *expression* THEN

statements that execute if the expression is TRUE

ELSEIF *expression* THEN

statements that execute if expression1 is TRUE

ELSE

statements that execute if all the preceding expressions are FALSE or NULL

END IF;



Example

```
create procedure upsal()
```

```
begin
```

```
    declare c int;
```

```
    select salary into c from emp where id=2 ;
```

```
    if(c>40000) then
```

```
        set c=c+5000;
```

```
    elseif c>30000 then
```

```
        set c=c+3000
```

```
    elseif c>20000 then
```

```
        set c=c+1000
```

```
    else
```

```
        set c=c+500;
```

```
    end if;
```

```
    update emp set salary=c where id=2;
```

```
end
```



Nested IF

```
IF Boolean-expression THEN  
    IF Boolean-expression THEN  
        statements  
    END IF;  
ELSE  
    IF Boolean-expression THEN  
        statements  
    END IF;  
END IF;
```



□ Write a procedure that check whether the student is valid or not.

username="BSCIT006"

password="MSCIT".

If user is valid SIS will print "valid user"
otherwise it will print "Invalid user".



CASE Statement

- The CASE statement is an alternative conditional execution or flow control statement.
- CASE statements are often more readable and efficient when multiple conditions need to be evaluated,
 - when the conditions all compare the output from a single expression.
- **Types of Case**
 - Simple CASE statement
 - Searched CASE statement



- *simple* CASE statement—compares the output of an expression with multiple conditions:

CASE *expression*

 WHEN *value* THEN *statements*

 [WHEN *value* THEN *statements* ...]

 [ELSE *statements*]

END CASE;



Terms in case statement

- **CASE expression**
 - If the value of *selector-expression* matches the first *match-expression*, the statements in the corresponding THEN clause are executed.
 - If there are no matches, the statements in the corresponding ELSE clause are executed.
 - If there are no matches and there is no ELSE clause, an exception is thrown.



Terms in case statement

- **WHEN Value**
 - If *expression* matches a *value* , the statements in the corresponding THEN clause are executed.
- **Statements**
 - Specifies one or more SQL or PL/SQL statements, each terminated with a semicolon.
- **ELSE**
 - A keyword that introduces the default case of the CASE statement.



Emp(id,name,salary,department)

Create procedure updateUsingCase()

begin

declare c varchar(20) ;

declare bonus int;

select dept into c from emp where id=2 ;

case c

when 'marketing' then

set bonus=1000;

when 'Sales' then

set bonus=500;

else

set bonus=0;

end case;

update emp set salary=salary+bonus where id=2;

end;



“Searched” CASE statement

- The *searched CASE statement* is functionally equivalent to an **IF-ELSEIF-ELSE-END** IF block.

CASE

 WHEN *condition* THEN
 statements

 [WHEN *condition* THEN
 statements...]

 [ELSE
 statements]

END CASE;




```
Create procedure upSalCaseSearch()
begin
  declare c int;
  declare bonus int;
  select salary into c from emp where id=2;
  case
    when c>=30000 then
      set bonus=c+2000;
    when c>=20000 then
      set bonus=c+1000;
    else
      set bonus=c+500;
  end case;
  update emp set salary=bonus where id=2;
end
```



LOOP

- We can have three different types of Loops in MySQL Stored Procedure, which are as below:
 - REPEAT
 - WHILE
 - LOOP, LEAVE & ITERATE



WHILE Statement

- Pretest Loop
- loop will check the condition first before executing the statement.

Syntax :

WHILE expression DO

Statements;

END WHILE



EXAMPLE

```
DECLARE //
```

```
DECLARE num INT;
```

```
DECLARE my_string VARCHAR(255);
```

```
SET num = 1;
```

```
SET str = '';
```

```
WHILE num <= 10 DO
```

```
    SET my_string = CONCAT(my_string,num,',');
```

```
    SET num = num + 1;
```

```
END WHILE//
```

```
DECLARE ;
```



REPEAT

- This loop also known as **POST-TEST** loop as this loop will execute the statement first then check for the condition.
- This loop will keep executing untill the conditions gets false.
- Syntax for this loop:

REPEAT

Statements;

UNTIL expression

END REPEAT;



EXAMPLE

```
DECLARE num INT;
```

```
DECLARE my_string VARCHAR(255);
```

```
REPEAT
```

```
    SET my_string = CONCAT(my_string, num, ',');
```

```
    SET num = num + 1;
```

```
UNTIL num > 5
```

```
END REPEAT;
```



ITERATE Statement

- ITERATE can appear only within LOOP, REPEAT, and WHILE statements.
- ITERATE means “start the loop again.”
- It is a same as continue statement .
- This is used where the record does not match certain criteria and logic need to get the next record and start processing.



EXAMPLE

- The iterate statement must have a label supplied as an argument.
- Any FOR or WHILE loop that has ITERATE statement, must have a corresponding label associated with it.



LEAVE Statement

- The leave statement is a compliments the iterate statement in FOR and WHILE loops.
- This statement is used to break the loop.
- If user want to get out of a loop then LEAVE statement is used.
- Loop is accomplished with a **LEAVE statement**. Within a stored function, **RETURN** can also be used, which exits the function entirely.




LEAVE Statement

- The LEAVE statement must have a label supplied as an argument.
- Any LOOP, FOR or WHILE loop that has LEAVE statement, must have a corresponding label associated with it.




```
CREATE PROCEDURE doiterate(p1 INT)
BEGIN
    label1: LOOP
        SET p1 = p1 + 1;
        IF p1 < 10 THEN
            ITERATE label1;
        END IF;
        LEAVE label1;
    END LOOP label1;
    SET @x = p1;
END;
```



LOOP

- LOOP is very simple control structure that continues to execute the statement within the loop until a return or leave statement is encountered.



EXAMPLE

BEGIN

declare sum integer default 0;

l:

LOOP

set sum= sum + 1;

IF sum > 10 THEN

leave l;

END IF;

END LOOP ;

END

/



Return Statement

- ❑ The return statement is used to exit out of a function and return a value.
- ❑ The return statement is not supported in SQL PL command block or trigger.
- ❑ If a return code is required then the signal command should be used.



Handler Syntax

- A *condition handler* defines the actions that the stored program is to take when a specified event—such as a warning or an error—occurs.

```
DECLARE handler_action HANDLER  
    FOR condition_value  
    ... Statement
```

- *handler_action*: CONTINUE | EXIT | UNDO
- *condition_value*:
 - *mysql_error_code*
 - SQLSTATE [VALUE] *sqlstate_value*
 - *condition_name*
 - SQLWARNING
 - NOT FOUND
 - SQLEXCEPTION



- The handler declaration has three main clauses;
 - Handler type (CONTINUE, EXIT)
 - Handler condition (SQLSTATE, MySQL error code, named condition)
 - Handler actions

➤ EXIT

- When an EXIT handler fires, the currently executing block is terminated.
- If this block is the main block for the stored program, the procedure terminates, and control is returned to the procedure or external program that invoked the procedure.
- If the block is enclosed within an outer block inside of the same stored program, control is returned to that outer block.

➤ CONTINUE

- With a CONTINUE handler, execution continues with the statement following the one that caused the error to occur. In either case, any statements defined within the handler (the handler actions ...

•



- The **DECLARE ... HANDLER** statement specifies a handler that deals with one or more conditions.
- If one of these conditions occurs, the specified statement executes.
- *statement* can be a simple statement such as
 - SET *var_name* = *value*, or
 - compound statement written using BEGIN and END




```
create procedure exception()  
begin  
    declare i,r int;  
    DECLARE continue HANDLER FOR SQLSTATE '42000' set r=1;  
    set r=0;  
    select salary into i from emp1;  
    if r=0 then  
        select i;  
    else  
        select 'Error Occurs';  
    end if;  
end;  
..
```

