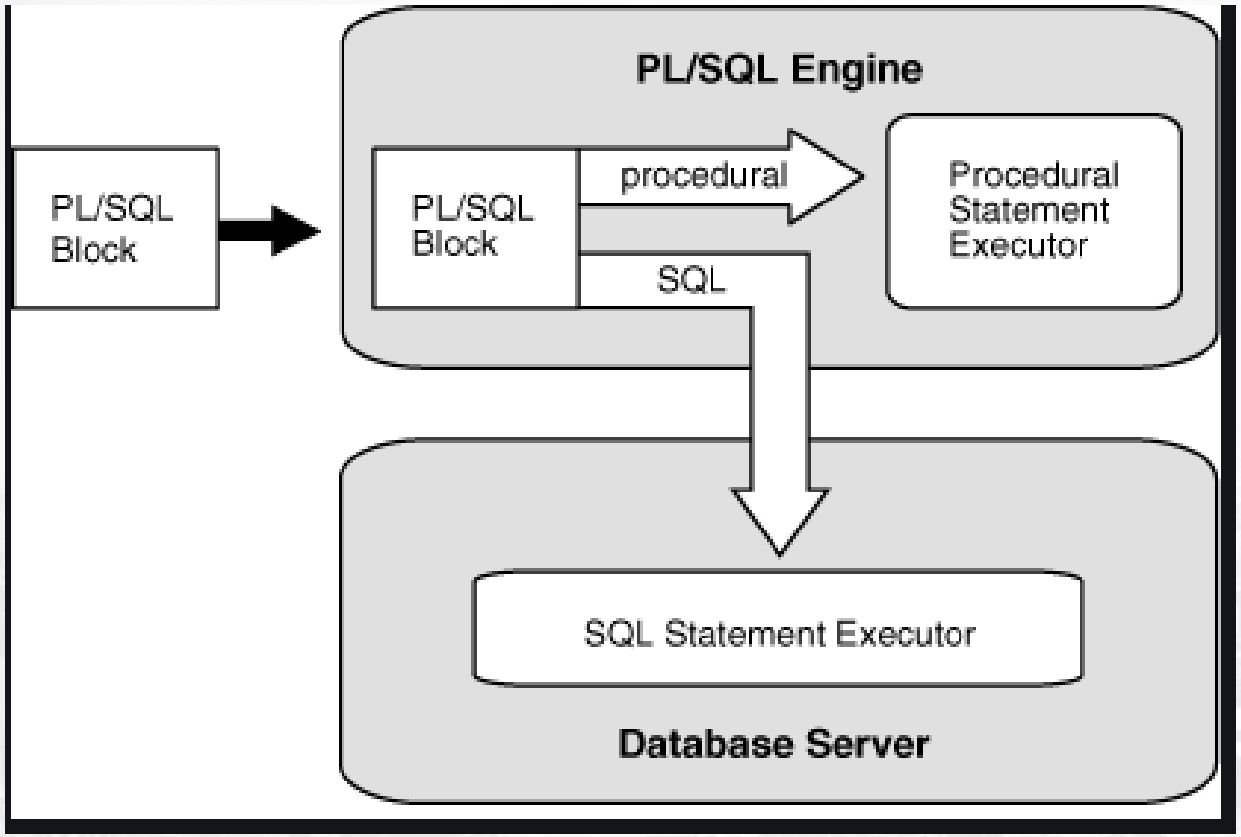
Extensions to SQL (PL/SQL)

What is PL/SQL

PL/SQL:

- Stands for Procedural Language extension to SQL
- Is Oracle Corporation's standard data access language for relational databases
- Seamlessly integrates procedural constructs with SQL

PL/SQL Execution



Advantages

- PL/SQL is a completely portable, high-performance transaction-processing language.
- PL/SQL provides a built-in, interpreted and OS independent programming environment.
- It supports structured programming through functions and procedures.
- Direct call can also be made from external programming language calls to database.

Basic Structure of PL/SQL

- PL/SQL stands for Procedural Language/SQL.
- PL/SQL extends SQL by adding constructs found in procedural languages
- The basic unit in PL/SQL is a block. All PL/SQL programs are made up of blocks, which can be nested within each other.
- Each block performs a logical action in the program.
- The only SQL statements allowed in a PL/SQL program are SELECT, INSERT, UPDATE, DELETE and several other data manipulation statements plus some transaction control.
- Data definition statements like CREATE, DROP, or ALTER are not allowed.
- The executable section also contains constructs such as assignments, branches, loops, procedure calls, and triggers,
- PL/SQL is not case sensitive. C style comments (/* ... */) may be used.

Basic Structure of PL/SQL

A block has the following structure:

```
DECLARE
   /* Declarative section: variables, types, and local subprograms. */
BEGIN

   /* Executable section: procedural and SQL statements go here. */

   /* This is the only section of the block that is required. */

EXCEPTION

   /* Exception handling section: error handling statements go here. */
END;
```

To execute a PL/SQL program,

- A line with a single dot (".", and then
- A line with run;

Variables and Types

- Information is transmitted between a PL/SQL program and the database through variables. Every variable has a specific type associated with it. That type can be
  - One of the types used by SQL for database columns
  - A generic type used in PL/SQL such as NUMBER
  - Declared to be the same as the type of some database column

Variables and Types(Contd..)

Variables of type NUMBER can hold either an integer or a real number. The most commonly used character string type is VARCHAR(n), where n is the maximum length of the string in bytes

```
DECLARE
   price  NUMBER;
   product VARCHAR(20);
```

In cases, where a PL/SQL variable will be used to manipulate data stored in a existing relation use %TYPE.

```
DECLARE
   myProduct Product.name%TYPE;
```

A variable may also have a type that is a record with several fields.

```
DECLARE
   ProductTuple Product%ROWTYPE;
```

Variables and Types(Contd..)

The initial value of any variable, regardless of its type, is NULL. We can assign values to variables, using the ":=" operator. The assignment can occur either immediately after the type of the variable is declared, or anywhere in the executable portion of the program. For example:

```
DECLARE
   a NUMBER := 3;
BEGIN
   a := a + 1;
END;
```

PL/SQL Functions and Procedures

- SQL:1999 supports functions and procedures
  - Functions/procedures can be written in SQL itself, or in an external programming language (e.g., C, Java).
  - Some database systems support **table-valued functions**, which can return a relation as a result.
- SQL:1999 also supports a rich set of imperative constructs, including
  - Loops, if-then-else, assignment
- Many databases have proprietary procedural extensions to SQL that differ from SQL:1999.
- Procedures and functions are stored in `mysql.routines` and `mysql.parameters` tables, which are part of the data dictionary.

Simple Programs in PL/SQL

The simplest form of program has some declarations followed by an executable section consisting of one or more of the SQL statements

```
CREATE TABLE T1(
   e INTEGER,
   f INTEGER
);
DELETE FROM T1;
INSERT INTO T1 VALUES(1, 3);
INSERT INTO T1 VALUES(2, 4);

/* Above is plain SQL; below is the PL/SQL program. */
DECLARE
   a NUMBER;
   b NUMBER;
BEGIN
   SELECT e,f INTO a,b FROM T1 WHERE e>1;
   INSERT INTO T1 VALUES(b,a);
END;
```

Control Flow in PL/SQL

PL/SQL allows you to branch and create loops in a fairly familiar way. An IF statement looks like:

IF <condition> THEN <statement_list> ELSE <statement_list> END IF;

The ELSE part is optional. If you want a multiway branch, use:

IF <condition_1> THEN ...

ELSIF <condition_2> THEN ...
... ...
ELSIF <condition_n> THEN ...
ELSE ...

END IF;

Control Flow in PL/SQL

Loops are created with the following:

LOOP

<loop_body> / A list of statements. */*

END LOOP;

At least one of the statements in <loop_body> should be an EXIT statement of the form

EXIT WHEN <condition>;

The loop breaks if <condition> is true.

Control Flow in PL/SQL

Loops are created with the following:

```
DECLARE
   i NUMBER := 1;

BEGIN
          LOOP
           INSERT INTO T1 VALUES(i, i);
           i := i+1;
           EXIT WHEN i>100;
   END LOOP;
END;
```

PL/SQL Control Flow

```
DECLARE
  b_profitable BOOLEAN;
  n_sales      NUMBER;
  n_costs      NUMBER;
BEGIN
  b_profitable := false;
  IF n_sales > n_costs THEN
    b_profitable := true;
  END IF;
END;
```

PL/SQL Control Flow

```
DECLARE
  n_sales NUMBER := 300000;
  n_commission NUMBER( 10, 2 ) := 0;
BEGIN
  IF n_sales > 200000 THEN
    n_commission := n_sales * 0.1;
  ELSE
    n_commission := n_sales * 0.05;
  END IF;
END;
```

PL/SQL Control Flow

```
DECLARE
  n_sales NUMBER := 300000;
  n_commission NUMBER( 10, 2 ) := 0;
BEGIN
  IF n_sales > 200000 THEN
    n_commission := n_sales * 0.1;
  ELSIF n_sales <= 200000 AND n_sales > 100000 THEN
    n_commission := n_sales * 0.05;
  ELSIF n_sales <= 100000 AND n_sales > 50000 THEN
    n_commission := n_sales * 0.03;
  ELSE
    n_commission := n_sales * 0.02;
  END IF;
END;
```

Stored Function

PL/SQL Functions

- Functions are declared using the following syntax:

```
Create function <function-name> (param_1, …, param_k)
        returns <return_type>
        [not] deterministic        allow optimization if same output for the
                same input (use RAND not deterministic )
Begin
    -- execution code
end;
```

For a FUNCTION, parameters are always regarded as IN parameters.

For a Procedure , parameter as IN, OUT, or INOUT is valid.

Deterministic and Non- deterministic Functions

- A deterministic function always returns the same result for the same input parameters whereas a non-deterministic function returns different results for the same input parameters.
- If you don't use DETERMINISTIC or NOT DETERMINISTIC, MySQL uses the NOT DETERMINISTIC option by default.
- **rand()** is nondeterministic function. That means we do not know what it will return ahead of time.
- Some **deterministic** functions
- ISNULL, ISNUMERIC, DATEDIFF, POWER, CEILING, FLOOR, DATEADD, DAY, MONTH, YEAR, SQUARE, SQRT etc.
- Some **non deterministic** functions
- RAND(), RANK(), SYSDATE()

PL/SQL Functions – Example 1

- Define a function that, given the name of a department, returns the count of the number of instructors in that department.

```
create function dept_count (dept_name varchar(20))
    returns integer
begin
    declare d_count     integer;
        select count ( * ) into d_count
        from instructor
        where instructor.dept_name = dept_name
    return d_count;
end
```

Example 1 (Cont)..

- The function dept_count can be used to find the department names and budget of all departments with more than 12 instructors.

```
select dept_name, budget
from department
where dept_count (dept_name ) > 12
```

Example 2

- A function that returns the level of a customer based on credit limit. We use the IF statement to determine the credit limit.

```
1  DELIMITER $$
2
3  CREATE FUNCTION CustomerLevel(p_creditLimit double) RETURNS VARCHAR(10)
4      DETERMINISTIC
5  BEGIN
6      DECLARE lvl varchar(10);
7
8      IF p_creditLimit > 50000 THEN
9   SET lvl = 'PLATINUM';
10     ELSEIF (p_creditLimit <= 50000 AND p_creditLimit >= 10000) THEN
11         SET lvl = 'GOLD';
12     ELSEIF p_creditLimit < 10000 THEN
13         SET lvl = 'SILVER';
14     END IF;
15
16   RETURN (lvl);
17 END
```

Example 2 (Cont..)

- **Calling function:**
- we can call the CustomerLevel() in a SELECT statement as follows:

```
1  SELECT
2      customerName,
3      CustomerLevel(creditLimit)
4  FROM
5      customers
6  ORDER BY
7      customerName;
```

Output:

	customerName	CustomerLevel(creditLimit)
▶	Alpha Cognac	PLATINUM
	American Souvenirs Inc	SILVER
	Amica Models & Co.	PLATINUM
	ANG Resellers	SILVER
	Anna's Decorations, Ltd	PLATINUM
	Anton Designs, Ltd.	SILVER

Example 3

```
mysql> select * from employee;
+----+-------+-----------+--------+---------------+------+
| id | name  | superid   | salary | bdate         | dno  |
+----+-------+-----------+--------+---------------+------+
|  1 | john  |         3 | 100000 | 1960-01-01    |    1 |
|  2 | mary  |         3 |  50000 | 1964-12-01    |    3 |
|  3 | bob   |      NULL |  80000 | 1974-02-07    |    3 |
|  4 | tom   |         1 |  50000 | 1970-01-17    |    2 |
|  5 | bill  |      NULL |   NULL | 1985-01-20    |    1 |
+----+-------+-----------+--------+---------------+------+
5 rows in set (0.00 sec)

mysql> delimiter !
mysql> create function giveRaise (oldval double, amount double
    -> returns double
    -> deterministic
    -> begin
    ->         declare newval double;
    ->         set newval = oldval * (1 + amount);
    ->         return newval;
    -> end !
Query OK, 0 rows affected (0.00 sec)

mysql> delimiter ;
```

Example 3 (cont..)

```
mysql> select name, salary, giveRaise(salary, 0.1) as newsal
    -> from employee;
+------+--------+--------+
| name | salary | newsal |
+------+--------+--------+
| john | 100000 | 110000 |
| mary |  50000 |  55000 |
| bob  |  80000 |  88000 |
| tom  |  50000 |  55000 |
| bill |   NULL |   NULL |
+------+--------+--------+
5 rows in set (0.00 sec)
```

```
mysql> DELIMITER $$
mysql>
mysql> CREATE FUNCTION isEligible(
    -> age INTEGER
    -> )
    -> RETURNS VARCHAR(20)
    -> DETERMINISTIC
    -> BEGIN
    ->     DECLARE customerLevel VARCHAR(20);
    ->
    ->       IF age > 18 THEN
    -> RETURN ("yes");
    -> ELSE
    -> RETURN ("No");
    -> END IF;
    ->
    -> END$$
Query OK, 0 rows affected (0.00 sec)

mysql> DELIMITER ;
```

```
mysql> select isEligible(20);
+----------------+
| isEligible(20) |
+----------------+
| yes            |
+----------------+
1 row in set (0.00 sec)
```

```
mysql> select isEligible(10);
+----------------+
| isEligible(10) |
+----------------+
| No             |
+----------------+
1 row in set (0.00 sec)
```

Stored Procedures

Stored Function Vs Stored Procedure

Function	Stored Procedure
Always returns a single value; either scalar or a table.	Can return zero, single or multiple values.
Functions are compiled and executed at run time.	Stored procedures are stored in parsed and compiled state in the database.
Only Select statements. DML statements like update & insert are not allowed.	Can perform any operation on database objects including select and DML statements.
Allows only input parameters. Does not allow output parameters.	Allows both input and output parameters
Does not allow the use of Try...Catch blocks for exception handling.	Allows use of Try...Catch blocks for exception handling.
Cannot have transactions within a function.	Can have transactions within a stored procedure.
Cannot call a stored procedure from a function.	Can call a function from a stored procedure.
Temporary tables cannot be used within a function. Only table variables can be used.	Both table variables and temporary tables can be used.
Functions can be called from a Select statement.	Stored procedures cannot be called from a Select/Where or Having statements. Execute statement has to be used to execute a stored procedure.
Functions can be used in JOIN clauses.	Stored procedures cannot be used in JOIN clauses

Stored Procedures in MySQL

- A stored procedure contains a sequence of SQL commands stored in the database catalog so that it can be invoked later by a program
- Stored procedures are declared using the following syntax:

```
Create Procedure <proc-name>
        (param_spec1, param_spec2,param_specn )
begin
    -- execution code
end;
```

where each param_spec is of the form:

[in | out | inout] <param_name> <param_type>

- in mode: allows you to pass values into the procedure,
- out mode: allows you to pass value back from procedure to the calling program, initial value in the procedure is taken as null .

  Inout mode : allows you to pass value back from procedure to the calling program, initial value in the procedure is taken from the caller value

Example 1 – No parameters

- The GetAllProducts() stored procedure selects all products from the products table.

```
mysql> use classicmodels;
Database changed
mysql> DELIMITER //
mysql> CREATE PROCEDURE GetAllProducts()
    -> BEGIN
    -> SELECT * FROM products;
    -> END//
Query OK, 0 rows affected (0.00 sec)

mysql> DELIMITER ;
mysql>
```

Example 1 (Cont..)

- **Calling Procedure:**

```
CALL GetAllProducts();
```

- **Output:**

productCode	productName	productLine	productScale
S10_1678	1969 Harley Davidson Ultimate Chopper	Motorcycles	1:10
S10_1949	1952 Alpine Renault 1300	Classic Cars	1:10
S10_2016	1996 Moto Guzzi 1100i	Motorcycles	1:10
S10_4698	2003 Harley-Davidson Eagle Drag Bike	Motorcycles	1:10
S10_4757	1972 Alfa Romeo GTA	Classic Cars	1:10

Example 2 (with IN parameter)

```
mysql> select * from employee;
+----+------+---------+--------+------------+------+
| id | name | superid | salary | bdate      | dno  |
+----+------+---------+--------+------------+------+
|  1 | john |       3 | 100000 | 1960-01-01 |    1 |
|  2 | mary |       3 |  50000 | 1964-12-01 |    3 |
|  3 | bob  |    NULL |  80000 | 1974-02-07 |    3 |
|  4 | tom  |       1 |  50000 | 1978-01-17 |    2 |
|  5 | bill |    NULL |   NULL | 1985-01-20 |    1 |
+----+------+---------+--------+------------+------+
```

```
mysql> select * from department;
+---------+-------------+
| dnumber | dname       |
+---------+-------------+
|       1 | Payroll     |
|       2 | TechSupport |
|       3 | Research    |
+---------+-------------+
```

- Suppose we want to keep track of the total salaries of employees working for each department

```
mysql> create table deptsal as
    -> select dnumber, 0 as totalsalary from department;
Query OK, 3 rows affected (0.00 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> select * from deptsal;
+---------+-------------+
| dnumber | totalsalary |
+---------+-------------+
|       1 |           0 |
|       2 |           0 |
|       3 |           0 |
```

We need to write a procedure to update the salaries in the deptsal table

Example 2 (Cont..)

```
mysql> delimiter //
mysql> create procedure updateSalary (IN param1 int)
    -> begin
    ->     update deptsal
    ->     set totalsalary = (select sum(salary) from employee where dno = param1)
    ->     where dnumber = param1;
    -> end; //
Query OK, 0 rows affected (0.01 sec)
```

1. Define a procedure called updateSalary which takes as input a department number.
2. The body of the procedure is an SQL command to update the totalsalary column of the deptsal table.

Example 2 (Cont..)

Step 3: Call the procedure to update the totalsalary for each department

```
mysql> call updateSalary(1);
Query OK, 0 rows affected (0.00 sec)

mysql> call updateSalary(2);
Query OK, 1 row affected (0.00 sec)

mysql> call updateSalary(3);
Query OK, 1 row affected (0.00 sec)
```

Example 2 (Cont..)

Step 4: Show the updated total salary in the deptsal table

```
mysql> select * from deptsal;
+---------+-------------+
| dnumber | totalsalary |
+---------+-------------+
|       1 |      100000 |
|       2 |       50000 |
|       3 |      130000 |
+---------+-------------+
3 rows in set (0.00 sec)
```

Example 3 (with OUT Parameter)

- The following example shows a simple stored procedure that uses an OUT parameter.
- Within the procedure MySQL MAX() function retrieves maximum salary from MAX_SALARY of jobs table.

```
mysql> CREATE PROCEDURE my_proc_OUT (OUT highest_salary INT)
-> BEGIN
-> SELECT MAX(MAX_SALARY) INTO highest_salary FROM JOBS;
-> END$$
Query OK, 0 rows affected (0.00 sec)
```

(Cont..)

- **Procedure Call:**

mysql> CALL my_proc_OUT(@M)$$

Query OK, 1 row affected (0.03 sec)

- To see the result type the following command

mysql< SELECT @M$$

- **Output:**

```
+-----+
| @M    |
+-----
| 40000|
+-----+
1 row in set (0.00 sec)
```

Example 4 (with INOUT Parameter)

- The following example shows a simple stored procedure that uses an INOUT parameter.
- 'count' is the INOUT parameter, which can store and return values and 'increment' is the IN parameter, which accepts the values from user.

```
mysql> DELIMITER // ;
mysql> Create PROCEDURE counter(INOUT count INT, IN increment INT)
    -> BEGIN
    -> SET count = count + increment;
    -> END //
Query OK, 0 rows affected (0.03 sec)
```

Example 4 (Cont..)

Function Call:

```
mysql> DELIMITER ;
mysql> SET @counter = 0;
Query OK, 0 rows affected (0.00 sec)

mysql> CALL counter(@Counter, 1);
Query OK, 0 rows affected (0.00 sec)

mysql> Select @Counter;
+----------+
| @Counter |
+----------+
| 1        |
+----------+
1 row in set (0.00 sec)
```

Stored Procedures (Cont..)

- Use show procedure status to display the list of stored procedures you have created

```
mysql> show procedure status;
+------+--------------+-----------+--------+---------------------+---------------------+----------
-----+---------+----------------------+----------------------+--------------------+
| Db   | Name         | Type      | Definer | Modified            | Created             | Security_
type | Comment | character_set_client | collation_connection | Database Collation |
+------+--------------+-----------+--------+---------------------+---------------------+----------
-----+---------+----------------------+----------------------+--------------------+
| ptan | updateSalary0 | PROCEDURE | ptan@%  | 2010-03-16 12:21:55 | 2010-03-16 12:21:55 | DEFINER
     |         | latin1               | latin1_swedish_ci    | latin1_swedish_ci  |
+------+--------------+-----------+--------+---------------------+---------------------+----------
-----+---------+----------------------+----------------------+--------------------+
1 row in set (0.02 sec)
```

- Use drop procedure to remove a stored procedure

```
mysql> drop procedure updateSalary;
Query OK, 0 rows affected (0.00 sec)
```

Language Constructs for Procedures & Functions

- SQL supports constructs that gives it almost all the power of a general-purpose programming language.
  - Warning: Most database systems implement their own variant of the standard syntax below.
- Compound statement: **begin … end**,
  - May contain multiple SQL statements between **begin** and **end**.
  - Local variables can be declared within a compound statements

Language Constructs

- **CASE Statement**

```
CASE  case_expression
  WHEN when_expression_1 THEN
  commands
  WHEN when_expression_2 THEN
  commands
  ...
  ELSE commands
END CASE;
```

- **While** and **repeat** statements:

while *boolean expression*
do *sequence of statements* ; **end while**

repeat
sequence of statements ;
until *boolean expression*
end repeat

Language Constructs (Cont.)

- **Loop, Leave and Iterate statements…**
  - Permits iteration over all results of a query.

```
loop_label:     LOOP
IF      x > 10 THEN
        LEAVE
loop_label; END  IF;
SET     x = x + 1;
IF      (x mod 2)
THEN ITERATE
loop_label; ELSE
                SET     str = CONCAT(str,x,',');
END     IF;
        END LOOP;
```