Database Management Systems

Vijaya Saradhi

IIT Guwahati

Mon, 09th Mar 2020

Announcements

Announcements

- Wishing you all a very happy and safe Holi
- Answer script distribution will be on
- Monday, 09-Mar-2020, 18:00 to 19:00 hours CSE seminar hall
- Tuesday, 10-Mar-2020, 18:30 to 19:30 CSE seminar hall
- Wednesday, 11-Mar-2020, 18:00 to 19:00 hours CSE seminar hall
- There will be no class on 13-Mar-2020
- There will be no class on 16-Mar-2020
- Lab will be held as per timetable on 12-Mar-2020
- We will have makeup class on mutually agreeable date and time
- This we will discuss on 19-Mar-2020

Introduction

MySQL Stored Programs

- Stored programs is a generic term used for stored procedure, stored functions and triggers
- Without stored programs database system cannot claim full compliance with variety of standards including ANSI/ISO standars
- These standards describe how a DBMS should execute stored programs.
- Judicial use of stored programs lead to greater database security and integriety
- Improve overall application performance
- Improve maintainability

What is Stored Program?

What is it anyway?

- A computer program
- A series of instructions associated with a name
- The source code and any compiled version of the stored program are held within database server's system tables
- Program is executed within the memory address of database server

What is Stored Program?

Stored Procedures

Invocation A generic program unit that is executed on request

Parameters Accepts multiple input and output parameters

What is Stored Program?

Stored Procedures

Invocation A generic program unit that is executed on request

Parameters Accepts multiple input and output parameters

Stored Functions

Similar to stored procedures

Constraint Execution results in the return of single value

Invocation Can be used within standard SQL statements

Extend SQL Use of functions in SQL statements amount to extending SQL functionality

What is Stored Program?

Stored Procedures

Invocation	A generic program unit that is executed on request
Parameters	Accepts multiple input and output parameters

Stored Functions

Similar	to stored procedures
Constraint	Execution results in the return of single value
Invocation	Can be used within standard SQL statements
Extend SQL	Use of functions in SQL statements amount to extending SQL functionality

Triggers

Invocation	Activated in response to an activity within the database
DML	In particular when INSERT, UPDATE or DELETE statements are used

Why use Stored Programs?

Why another language?

- Developers have multitude of programming languages from which to choose
- Many of these are not database languages
- The code written in these languages does not reside in or managed by database server
- Stored programs offer many advantages. These are

Why use Stored Programs?

Advantages of Stored Programs

- Can lead to more secure database
- Offer mechanism to abstract data access routines in turn improve the maintainability of code as data structures evolve
- Reduces network traffic; Work on the data from within the server rather than transferring data across network
- Can be used to implement Common routines accessible from multiple applications
- They can be executed either within the database server
- Database-centric logic can be isolated in stored programs

Language Fundamentals

Variables

Declaration DECLARE `variable_name` datatype;

Example `DECLARE first_var INT;`

Value `first_var` is initialized with $\perp$ (NULL)

Example `DECLARE first_var INT DEFAULT 0;`

Value `first_var` is initialized with value 0

Language Fundamentals

Variables

Declaration DECLARE `variable_name` datatype;

Example `DECLARE first_var INT;`

Value `first_var` is initialized with $\perp$ (NULL)

Example `DECLARE first_var INT DEFAULT 0;`

Value `first_var` is initialized with value 0

More examples

- `DECLARE var1 INT DEFAULT -20000;`
- `DECLARE var2 FLOAT DEFAULT 1.8e-8;`
- `DECLARE var3 DOUBLE DEFAULT 2e45;`
- `DECLARE var4 DATE DEFAULT '1999-12-31';`

Assigning Values to Variables

Example - 1

```
SET variable_name = expression;
SET var1 = 10;
```

Example - 2

```
SET variable_name = expression;
SET var2 = 10.0001;
```

Example - 3

```
SET variable_name = expression;
SET var4 = '2018-11-12';
```

Parameters

Procedures and Functions

Are variables that can be passed into or out of the stored program

Three types exists

IN Value must be specified by calling program. Modifications within stored program cannot be accessed from calling program

OUT Modifications within stored program can be accessed from calling program.

INOUT AN INOUT parameter acts both as IN and as an OUT parameter

Parameter - IN

Example

```
DELIMITER //
CREATE PROCEDURE demoIN(IN var1 INT)
BEGIN
    -- See the value of IN parameter
    SELECT var1;

    -- Modify
    SET var1 = 2;

    -- See the value of IN parameter
    SELECT var1;
END;//
DELIMITER ;
```

Execution

```
mysql> SET @myvar = 1;
mysql> CALL demoIN(@myvar);
mysql> SELECT @myvar;
```

- First line initializes @myvar variable
- Second line calls the stored procedure `demoIN`
- Withing `demoIN` var1 is read containing value 1
- Withing `demoIN` var1 is modified to value 2
- Third line read the variable @myvar which is 1

Parameter - OUT

Example

```
DELIMITER //
CREATE PROCEDURE demoOUT(OUT var1 INT)
BEGIN
    -- See the value of OUT parameter
    SELECT var1;

    -- Modify
    SET var1 = 2;

    -- See the value of OUT parameter
    SELECT var1;
END;//
DELIMITER ;
```

Execution

```
mysql> SET @myvar = 1;
mysql> CALL demoOUT(@myvar);
mysql> SELECT @myvar;
```

- First line initializes @myvar variable
- Second line calls the stored procedure `demoOUT`
- Withing `demoOUT` var1 is read containing value NULL (irrespective of its initialization outside procedure)
- Withing `demoOUT` var1 is modified to value 2
- Third line read the variable @myvar which is 2

Parameter - INOUT

Example

```
DELIMITER //
CREATE PROCEDURE demoINOUT(INOUT var1 INT)
BEGIN
    -- See the value of INOUT parameter
    SELECT var1;

    -- Modify
    SET var1 = 2;

    -- See the value of INOUT parameter
    SELECT var1;
END;//
DELIMITER ;
```

Execution

```
mysql> SET @myvar = 1;
mysql> CALL demoINOUT(@myvar);
mysql> SELECT @myvar;
```

- First line initializes @myvar variable
- Second line calls the stored procedure `demoINOUT`
- Withing `demoINOUT` var1 is read containing value 1
- Withing `demoINOUT` var1 is modified to value 2
- Third line read the variable @myvar which is 2

Built-in Functions

Categories

String functions Perform string manipulation; concatenation of two strings, obtaining substring etc

Mathematical functions Example: trigonometric functions, random number functions, logarithms etc

Date and time functions add or subtract time intervals from dates; find difference between two dates etc

Miscellaneous functions every thing not easily categorized in the above three groupings; encryption functions etc

Built-in Functions

String functions

```
SELECT roll_number, CONCAT(sur_name, " ", first_name, " ", last_name) as full_name
FROM   Student
WHERE  Dept = 'EEE';
```

Built-in Functions

Mathematical functions

```
SELECT roll_number, ABS(quiz1_marks)
FROM   Student
WHERE  Dept = 'BSBE';
```

Built-in Functions

Mathematical functions

```
SELECT roll_number, ROUND(SPI, 2)
FROM   Student
WHERE  Dept = 'EEE';
```

Built-in Functions

Date and time functions

```
SELECT roll_number, DAYNAME(held_on)
FROM   Attendance
WHERE  cid = 'CS441M';
```

Built-in Functions

Date and time functions

```
SELECT DATE_ADD('2018-05-01',INTERVAL 1 DAY);
-- '2018-05-02'

SELECT DATE_SUB('2018-05-01',INTERVAL 1 YEAR);
-- '2017-05-01'

SELECT DATE_ADD('2020-12-31 23:59:59', INTERVAL 1 SECOND);
-- '2021-01-01 00:00:00'

SELECT DATE_ADD('2018-12-31 23:59:59', INTERVAL 1 DAY);
-- '2019-01-01 23:59:59'
```

Blocks, Conditional statements

Block structure of stored programs

- Stored program consists of one or more blocks
- Each block commences with a BEGIN statement and terminate by an END
- Blocks are useful for defining variables within a block
- Variable within a block are not visible outside the block

Blocks

Block structure

- Various types of declarations can appear in a block
- Order in which these can occur is as follows
- Variable and condition declarations (errors)
- Cursor declarations
- Handler declarations
- Program code
- Violation of this order results in error

Blocks

Block structure - order

```
[label:] BEGIN
    variable declarations
    condition declarations
    cursor declarations
    handler declarations

    program code
END [label];
```

Blocks

Block structure - Example

```
DELIMITER //
CREATE PROCEDURE f1()
BEGIN
    DECLARE var1 INT DEFAULT 10;
END;//
DELIMITER ;
```

Nested Blocks

Nested block structures

- Some instances needed nested block structures
- Blocks that are defined within an enclosing block
- Variables defined within a block are not available outside the block
- However the variables are visible to blocks that are declared within the block

Nested Blocks

Nested block structure - Example

```
DELIMITER //
CREATE PROCEDURE f1()
BEGIN
    DECLARE outer_variable INT DEFAULT 10;
    BEGIN
        DECLARE inner_variable INT DEFAULT 20;
        SET inner_variable = 22;
    END;

    SET outer_variable = 12;

END;//
DELIMITER ;
```

Nested Blocks

Nested block structure - Example

```
DELIMITER //
CREATE PROCEDURE f2()
BEGIN
    DECLARE outer_variable INT DEFAULT 10;
    BEGIN
        DECLARE inner_variable INT DEFAULT 20;
        SET inner_variable = 22;
    END;
    SET outer_variable = 12;
    SELECT inner_variable, 'This statement causes an error';

END; //
DELIMITER ;
```

Nested Blocks - Overriding variables

Nested block structure - Example

```
DELIMITER //
CREATE PROCEDURE f3()
BEGIN
    DECLARE outer_variable INT DEFAULT 10;
    SET outer_variable = 27;
    BEGIN
        SET outer_variable = 57;
    END;
    SELECT outer_variable, 'This statement causes overwriting
     on 27 with 57';
END;//
DELIMITER ;
```

Nested Blocks

Nested block structure - Example

Changes made to an overloaded variable in an inner block are not visible outside the block

```
DELIMITER //
CREATE PROCEDURE f4()
BEGIN
    DECLARE my_variable varchar(20);
    SET my_variable='This value was set in the OUTER block';

    BEGIN
        DECLARE  my_variable varchar(20);
        SET my_variable='This value was set in the INNER block';
    END;

    SELECT my_variable, 'Can''t see changes made in the INNER block';
    SELECT 'As the scope of INNER BLOCK is ended';
END;//
DELIMITER ;
```

LEAVE statement

Exiting nested blocks

```
DELIMITER //
CREATE PROCEDURE f5()
outer_block: BEGIN

    DECLARE l_status INT;
    SET l_status=1;

    inner_block: BEGIN
        IF (l_status = 1)
        THEN
            LEAVE inner_block;
        END IF
        SELECT 'This statement will never be executed';
    END inner_block;
    SELECT 'End of program';
END outer_block;//
                DELIMITER ;
```

Conditional Control

Conditional Statement - IF

```
DELIMITER //
CREATE FUNCTION s_AND_d(IN sale_id INT, IN sale_value FLOAT)
BEGIN
    IF( sale_value > 200 )
    THEN
        CALL apply_free_shipping(sale_id);

        IF ( sale_vale > 500 )
        THEN
            CALL apply_discount(sale_id, 20);
        END IF;

    END IF;

END; //
DELIMITER ;
```

```
mysql> SELECT customer_name, s_AND_d(sale_id, sale_value) FROM  Customer;
```

Conditional Control

Conditional Statement - IF

```
DELIMITER //
CREATE FUNCTION f6(IN cpi FLOAT)
BEGIN
    IF( cpi > 7.0 )
    THEN

        SELECT roll_number, full_name
        FROM   Student
        WHERE  Dept = 'EEE';

    ELSE IF ( cpi BETWEEN 5.0 AND 7.0 )
    THEN

        SELECT roll_number, full_name
        FROM   Student
        WHERE  Dept = 'BSBE';

    ELSE

        SELECT roll_number, full_name
        FROM   Student
        WHERE  Dept <> 'BSBE' AND Dept <> 'EEE';

    END IF;
END; //
DELIMITER ;
```

Conditional Control

Conditional Statement - CASE

Functionally equivalent to IF - ELSE IF - ELSE - END block

```
CASE
    WHEN condition THEN
        statements
    [WHEN condition THEN
        statements]
    [ELSE
        statements]
END CASE;
```

Conditional Control

Conditional Statement - CASE

```
DELIMITER //
CREATE FUNCTION f7(IN sale_value FLOAT, IN customer_status
        ENUM('PLATINUM', 'GOLD', 'SILVER', 'BRONZE'), IN sale_id INT)
BEGIN
    DECLARE dummy INT DEFAULT -1;
    CASE
        WHEN (sale_value > 200 AND customer_status = 'PLATINUM' ) THEN
            CALL free_shipping(sale_id);
            CALL apply_discount(sale_id, 20);
        WHEN (sale_value > 200 AND customer_status = 'GOLD' ) THEN
            CALL free_shipping(sale_id);
            CALL apply_discount(sale_id, 15);
        WHEN (sale_value > 200 AND customer_status = 'SILVER' ) THEN
            CALL free_shipping(sale_id);
            CALL apply_discount(sale_id, 10);
        WHEN (sale_value > 200 AND customer_status = 'BRONZE' ) THEN
            CALL free_shipping(sale_id);
            CALL apply_discount(sale_id, 5);
        WHEN (sale_value > 200 ) THEN
            CALL free_shipping(sale_id);
        ELSE
            SET dummy = 0;
    END CASE; //
DELIMITER ;
```

Iterative Processing with Loops

- LOOP statement
- REPEAT ... UNTIL
- WHILE

LOOP

Syntax

```
[label:] LOOP
      statements
END LOOP [label];
```

LOOP

Example

```
DELIMITER //
CREATE PROCEDURE f7()
BEGIN
    DECLARE i INT DEFAULT 1;
    SET i = 1;
    myloop: LOOP
        SET i = i + 1;
        IF i = 10
        THEN
            LEAVE myloop;
        END IF;
    END LOOP myloop;
    SELECT 'I can count 10';
END; //
DELIMITER ;
```

REPEAT ... UNTIL

Syntax

```
[label:] REPEAT
    statements
UNTIL expression
END REPEAT [label]
```

REPEAT ... UNTIL

Example

```
DELIMITER //
CREATE PROCEDURE f8()
BEGIN
    DECLARE i INT DEFAULT 1;
    SET i = 0;
    loop1: REPEAT

        SET i = i + 1;

        IF MOD(i, 2) <> 0
        THEN
            SELECT CONCAT(i, " is an ODD number");
        END IF;

    UNTIL i >= 10;

    END REPEAT loop1;
END; //
DELIMITER;
```

WHILE Loop

Syntax

```
[label:] WHILE expression
DO
    statements
END WHILE [label]
```

WHILE Statement

Example

```
DELIMITER //
CREATE PROCEDURE f9()
BEGIN
    DECLARE i INT DEFAULT 1;
    SET i = 1;
    loop1: WHILE i <= 10 DO

        IF MOD(i, 2) <> 0
        THEN
            SELECT CONCAT(i, " is ODD number");
        END IF;
        SET i = i + 1;

        END WHILE loop1;
END; //
DELIMITER;
```

Nested loops

Example

```
DELIMITER //
CREATE PROCEDURE f10()
BEGIN
    DECLARE i INT DEFAULT 1;
    DECLARE j INT DEFAULT 1;
    outer_loop: LOOP
        SET j = 1;
        inner_loop: LOOP
            SELECT CONCAT(i, " times ", j, " is ", i * j);
            SET j = j + 1;
            IF j > 12
            THEN
                LEAVE inner_loop;
            END IF;
        END LOOP inner_loop;

        SET i = i + 1;
        IF i > 12
        THEN
            LEAVE outer_loop;
        END IF;
    END outer_loop;
END; //
DELIMITER ;
```

Stored Procedure

Example

```
DELIMITER //
CREATE PROCEDURE simple_sqls()
BEGIN
    DECLARE i INT DEFAULT 1;

    DROP TABLE IF EXISTS test_table;
    CREATE TABLE test_table(id INT, some_data CHAR(30), PRIMARY KEY (id));

    WHILE ( i <= 10 )
    DO
        INSERT INTO test_table(i, CONCAT("record", i));
        SET i = i + 1;
    END WHILE;

END; //
DELIMITER ;
```

Cursor - I

Impedance Model Mis-match

- SQL always returns relations
- Other programming languages has data types that are not relations
- These languages cannot hold relations returned by SQL
- C language has pointers; where as SQL do not have any such construct
- As a result, passing data between SQL and other languages is not straightforward
- Mechanisms must be devised to allow the development of programs that use both SQL and other languages

Cursor - I

Impedance Model Mis-match

- Versatile way to connect SQL queries to a host language is with a cursor
- Cursor runs through the tuples of a relation
- This relation can be stored table, or it can be something that is generated by a query

Cursor - I

Details

- SELECT will return a relation
- Returned relation will not be stored
- Often the need to process one row at a time of returned relation arise
- Cursor helps examining one row at a time

Cursor - II

Details

- Assume the returned relation to be a file in itself
- Operations required for reading a file are
  - Declare file pointer
  - Open the file
  - Read one line at a time repeatedly
  - close the file
- Similar tasks are associated with cursor

Cursor - III

Declare cursor

```
DECLARE cursor_name CURSOR FOR SELECT statement;
```

```
OPEN cursor_name;
```

```
FETCH cursor_name INTO variable_list;
```

```
CLOSE cursor_name
```

Cursor - Example

Example

```
DELIMITER //
CREATE PROCEDURE f11()
BEGIN
-- Declare variables
    DECLARE i INT DEFAULT 1;

-- Declare cursors
    DECLARE my_first_cursor CURSOR FOR
    SELECT   *
    FROM     Sailors
    WHERE    age > 20 AND rating BETWEEN 5 AND 7;

-- Declare cursor handler
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET NO_records = 1;

    OPEN my_first_cursor;
-- loop through all the rows
    loop_1: REPEAT
-- Get one roll number from list of registered students into variable rn
        FETCH my_first_cursor INTO (sno, sname, rating, age);

-- Check number of records in the cursor
        IF NO_records = 1 THEN
            LEAVE loop_1;
        END IF;
```

Cursor - Example

Example

```
        UNTIL NO_records
    END REPEAT loop_1;
    CLOSE my_first_cursor;
END; //
DELIMITER ;
```

Cursor - IV

Scrolling

- Cursor gives us flexibility as how to move through the tuples of the relation
- The default choice is to start at the beginning of the relation and fetch the tuples in order
- Fetch all tuples until end of the relation
- Other orders in which tuples may be fetched
- These options are not available in `MySQL` yet we will discuss these

Cursor - V

Scrolling

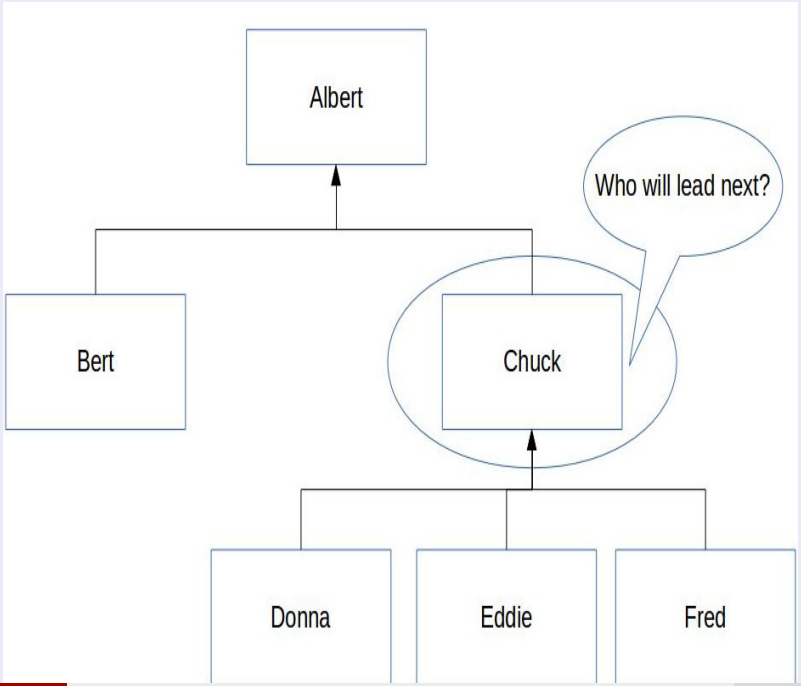
- Instruct the cursor to open in `SCROLL` model before the keyword `CURSOR`
- `EXEC SQL DECLARE name SCROLL CURSOR FOR MovieExec;`
- This will tell SQL that cursor may be used in a manner other than moving in forward direction alone
- The FETCH is responsible for specifying the direction from which the next tuple be obtained
  - `FETCH NEXT` retrieve next tuple
  - `FETCH PRIOR` retrieve previous tuple
  - `FETCH FIRST` retrieve first tuple
  - `FETCH LAST` retrieve last tuple
  - `FETCH ABSOLUTE i` specifies the position of the tuple to be fetched from the top of the relation

Supervisor-supervisee

Manages Relation

Employee	Boss	Salary
Albert	$\perp$	1000.00
Bert	Albert	900.00
Chuck	Albert	900.00
Donna	Chuck	800.00
Eddie	Chuck	700.00
Fred	Chuck	600.00

Manages Relation



Supervisor-supervisee

Anomalies

INSERT Can include cycles in the graph

UPDATE

DELETE

Structural

Insertion Anomaly Example

Employee	Boss	Salary
Albert	$\perp$	1000.00
Albert	Fred	100.00
Bert	Albert	900.00
Chuck	Albert	900.00
Donna	Chuck	800.00
Eddie	Chuck	700.00
Fred	Chuck	600.00

Supervisor-supervisee

Anomalies

INSERT	Can include cycles in the graph
UPDATE	`UPDATE manager set Employee='Charles' where Employee = 'Chuck';`
DELETE	
Structural	

UPDATE Anomaly Example

Employee	Boss	Salary
Albert	$\perp$	1000.00
Bert	Albert	900.00
Charles	Albert	900.00
Donna	Chuck	800.00
Eddie	Chuck	700.00
Fred	Chuck	600.00

In atomic fashion

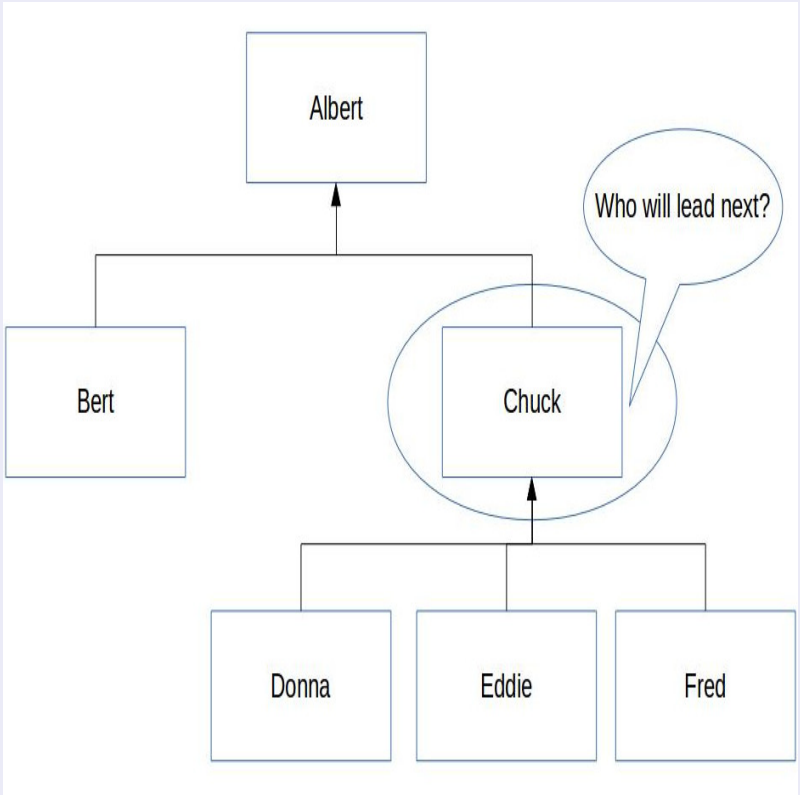
```
UPDATE manager set Employee='Charles' where Employee
= 'Chuck';
UPDATE manager set Boss='Charles' where Boss =
'Chuck';
```

Supervisor-supervisee

Anomalies

INSERT	Can include cycles in the graph
UPDATE	`UPDATE manager set Employee='Charles' where Employee = 'Chuck';`
DELETE	Chuck left the organization. What should be the right way?
Structural	

DELETE Anomaly Example



Supervisor-supervisee

Structural Anomalies

- `INSERT INTO Manager (Employee, Boss) VALUES ('a', 'a');`
- Create simple cycles
- `INSERT INTO Manager (Employee, Boss) VALUES ('b', 'c');`
- `INSERT INTO Manager (Employee, Boss) VALUES ('c', 'b');`

Supervisor-supervisee: Solution - Part I

Modify relation

- Employee details and organization hierarchy must be separated
- Create table for `Employee(eid, ename, address)`
- Create table for hierarchy `Manages(role, eid, boss_eid)`
- role should be primary key
- (eid, boss_eid) should be unique
- eid should be foreign key referring Employee
- eid default value should be 0 to indicate vacant position
- eid should not be NULL

Supervisor-supervisee: Solution - Part II

Constraints

- Self boss is not allowed. `CHECK(eid <> boss_eid);`
- boss_eid and eid should not be 0; `CHECK( boss_eid != 0 AND eid != 0 )`
- Number of nodes in tree: `SELECT COUNT(*) FROM Manages`
- Number of edges in tree: `SELECT COUNT(boss_eid) FROM Manages`
- Number of edges = number of nodes - 1; `CHECK( (SELECT COUNT(*) FROM Manages) - 1 = (SELECT COUNT(boss_eid) FROM Manages) )`
- Only one root:
  `CHECK(SELECT COUNT(*) FROM Manages where ISNULL(boss_eid) = 1)`

Supervisor-supervisee: Solution - Part III

Constraints - Check for Cycles

```
1 CREATE FUNCTION TreeTest() RETURNS CHAR(6)
2 BEGIN ATOMIC
3 -- put a copy in a temporary table
4   INSERT INTO Tree SELECT eid, boss_id FROM Manages
5
6 -- prune the leaves
7   WHILE ((SELECT COUNT(*) FROM Tree) - 1) = (SELECT COUNT(boss_id) FROM Tree)
8   DO
9       DELETE FROM Tree
10      -- Check employee is not the boss
11      WHERE Tree.eid
12      NOT IN (
13          -- Select all the bosses
14          SELECT T2.boss_id
15          FROM Tree AS T2
16          WHERE NOT ISNULL(T2.boss_id)
17      );
18
19  IF NOT EXISTS (SELECT * FROM Tree)
20  THEN
21      RETURN ('Tree');
22  ELSE
23      RETURN ('Cycles');
24  END IF;
25END WHILE;
END;
```

Supervisor-supervisee: Steps

Detailed Steps

```
Iteration #1
-----------------------------------------------------------------
Albert Not in {Albert, Albert, Chuck, Chuck, Chuck}? No;
Bert  Not in {Albert, Albert, Chuck, Chuck, Chuck}? Yes; Delet
Chuck Not in {Albert, Albert, Chuck, Chuck, Chuck}? No;
Donna Not in {Albert, Albert, Chuck, Chuck, Chuck}? Yes; Delet
Eddie Not in {Albert, Albert, Chuck, Chuck, Chuck}? Yes; Delet
Fred  Not in {Albert, Albert, Chuck, Chuck, Chuck}? Yes; Delet
-----------------------------------------------------------------
```

Supervisor-supervisee: Steps

Detailed Steps

```
Iteration #2
-----------------------------------------------------------
Albert NULL
Chuck  Albert

Albert  Not in {Albert} No;
Chuck   Not in {Albert} Yes; Delete
-----------------------------------------------------------
```

Supervisor-supervisee: Steps

Detailed Steps

```
Iteration #3
-----------------------------------------------------------------
Albert NULL
Albert  Not in {} Yes; Delete
-----------------------------------------------------------------
```

Exceptions

SQL exception - 01

- An SQL system indicates error conditions by setting non-zero sequence of digits in SQLSTATE
- Example 02000 `no tuple found`
- Example 21000 `single row select has returned more than one row`
- We can declare user defined exceptions called exception handler
- Invoked whenever one of a list of these error codes appear in SQLSTATE during execution of a statement
- Each exception handler is associated with a block of code
- delineated by `BEGIN ... END`

Exceptions

SQL exception - 02

- The form of a handler declaration is
- DECLARE [where to go] HANDLER FOR [condition list] [statement]
- where to go:

CONTINUE	means that after executing the statement in the handler declaration, we execute the statement after the one raised the exception
EXIT	after executing the handler's statement, control leaves `BEGIN ... END` block in which the handler is declared
UNDO	Same as EXIT which a difference that any changes to the database or local variables that were made by the block executed so far are undone