University College Cork, Ireland
Coláiste na hOllscoile Corcaigh

Information Storage and Management I

Dr. Alejandro Arbelaez

Views

Today

- Stored Procedures
- Views

Advantages of Views

- **Simplify complex query**:
  - If you have any frequently used complex query, you can create a view based on it so that you can reference to the view by using a simple SELECT statement instead of typing the query all over again.
- **Make the business logic consistent:**
  - suppose you have to repeatedly write the same formula in every query.
  - or you have a query that has complex business logic. To make this logic consistent across queries, you can use a view to store the calculation and hide the complexity.
- **Add extra security layers:**
  - A table may expose a lot of data including sensitive data such as personal and banking information.
- **Enable backward compatibility**:
  - In legacy systems, views can enable backward compatibility.

Example

Student

sid	name	login	age	gpa
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@eecs	18	3.2
53650	Smith	smith@math	19	3.8

Enrolled

sid	cid	grade
53831	Carnatic101	C
53831	Reggae203	B
53650	Topology112	A
53666	History105	B

Views

- A ***view*** is just a relation, but we store a ***definition***, rather than a set of tuples.

```
CREATE VIEW YoungActiveStudents (name, grade)
        AS SELECT S.name, E.grade
        FROM Students S, Enrolled E
        WHERE S.sid = E.sid and S.age<21
```

Views make life easy

SELECT * **FROM** YoungActiveStudents **WHERE** grade > 'A’;

SELECT S.sid, S.name, E.grade, S.age **FROM** Students S, Enrolled E
WHERE S.sid = E.sid **AND** S.age < 21 **AND** E.grade > 'A';

Deleting Views

- **DROP VIEW** <view name>
- Dropping a view does not affect any tuples in the underlying relation
- Dropping a table would remove all of the tuples and views that use that table

```
DROP TABLE Students;
SELECT * FROM YoungActiveStudents;
```

- What would happen if we delete Students:

Deleting Views

- **DROP VIEW** <view name>
- Dropping a view does not affect any tuples in the underlying relation
- Dropping a table would remove all of the tuples and views that use that table

```
DROP TABLE Students;
SELECT * FROM YoungActiveStudents;
```

- What would happen if we delete Students:
  - Some DBMS would not allow it because it has dependencies
  - Some would delete it and then we will be unable to answer additional queries

Uses for Views

- Views can be used to present necessary information (or a summary), while hiding details in underlying relation(s) (***security***).
- Views are also useful for maintaining ***logical data independence*** when the conceptual schema changes.
- Can be used to precompute (***materialize***) results or partial results of common queries.

Views vs. Relations

- Logical distinctions:
  - ***Updates not always possible*** to a view
- Physical distinctions:
  - Relations must be physically stored somewhere
  - Views are either:
    - Computed ***on-demand*** (not indexable)
    - Stored physically (***materialized***) to enhance performance, and the DBA (or system) must manage the replication.

Updates to Views

- Whether view is materialized or not, we **can't always update a view** because there may not be a unique update to base tables that reflects the update to the view.
- **Single-table** views are usually updateable.
- **Multi-table** views are more difficult. We will consider views defined using union, intersect, minus, and join.

Updates to Single-Table Views

- ***Selection-based views***: INSERT, DELETE are mapped directly to the base relation.
- ***Projection-based views***: view must include all fields of base relation that disallow null; base table insertion is padded with nulls.
- ***Aggregate views***: not updateable.

```
CREATE VIEW YearAvg AS
SELECT S.year, AVG (S.gpa)
FROM Students S
GROUP BY S.year
```

Updatable views

- There must be a one-to-one relationship between the rows in the view and the rows in the underlying table
- A view is not updatable if it contains any of the following:
  - Aggregate functions
  - DISTINCT
  - GROUP BY
  - HAVING
  - UNION
  - Subquery
  - https://dev.mysql.com/doc/refman/8.0/en/view-updatability.html

Example

```
mysql> CREATE TABLE t (qty INT, price INT);
mysql> INSERT INTO t VALUES(3, 50), (5, 60);
mysql> CREATE VIEW v AS SELECT qty, price, qty*price AS value FROM t;
mysql> SELECT * FROM v;
+------+-------+-------+
| qty  | price | value |
+------+-------+-------+
|    3 |    50 |   150 |
|    5 |    60 |   300 |
+------+-------+-------+
mysql> SELECT * FROM v WHERE qty = 5;
+------+-------+-------+
| qty  | price | value |
+------+-------+-------+
|    5 |    60 |   300 |
+------+-------+-------+
```

Example

```
mysql> UPDATE V SET price=50 WHERE qty=5;
```

Updatable View?

Example

```
mysql> UPDATE V SET price=50 WHERE qty=5;
Query OK, 0 rows affected (0.00 sec)
Rows matched: 1  Changed: 0  Warnings: 0
```

Updatable View? ✓

```
mysql> SELECT * FROM V;
+------+-------+-------+
| qty  | price | value |
+------+-------+-------+
|    3 |    50 |   150 |
|    5 |    50 |   250 |
+------+-------+-------+
2 rows in set (0.00 sec)
```

Example

```
mysql> UPDATE V SET VALUE=50 WHERE qty=5;
```

Updatable View?

Example

```
mysql> UPDATE V SET VALUE=50 WHERE qty=5;
ERROR 1348 (HY000): Column 'value' is not updatable
```

Updatable View? **X**

Not a one-to-one relationship between value and the original table

sid	name	login	age	gpa
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@eecs	18	3.2
53650	Smith	smith@math	19	3.8

sid	cid	grade
53831	Carnatic101	C
53831	Reggae203	B
53650	Topology112	A
53666	History105	B

```
CREATE VIEW YoungActiveStudents (name, grade)
        AS SELECT S.name, E.grade
        FROM Students S, Enrolled E WHERE S.sid = E.sid and S.age<21
```

sid	name	grade	age
53650	Smith	A	19
53666	Jones	B	18

sid	name	login	age	gpa
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@eecs	18	3.2
53650	Smith	smith@math	19	3.8

sid	cid	grade
53831	Carnatic101	C
53831	Reggae203	B
53650	Topology112	A
53666	History105	B

```
CREATE VIEW YoungActiveStudents (name, grade)
        AS SELECT S.name, E.grade
        FROM Students S, Enrolled E WHERE S.sid = E.sid and S.age<21

UPDATE YoungActiveStudents SET grade = 'F' WHERE sid = '53650';
```

sid	name	grade	age
53650	Smith	A	19
53666	Jones	B	18

sid	name	grade	age
53666	Jones	B	18
53650	Smith	F	19

sid	name	login	age	gpa
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@eecs	18	3.2
53650	Smith	smith@math	19	3.8

sid	cid	grade
53831	Carnatic101	C
53831	Reggae203	B
53650	Topology112	A
53666	History105	B

```
CREATE VIEW YoungActiveStudents (name, grade)
        AS SELECT S.name, E.grade
        FROM Students S, Enrolled E WHERE S.sid = E.sid and S.age<21

UPDATE YoungActiveStudents SET age = 28 WHERE sid = '53650';
```

sid	name	grade	age
53666	Jones	B	18

YoungActiveStudents

sid	name	login	age	gpa
53650	Smith	smith@math	28	4
53666	Jones	jones@cs	18	3
53688	Smith	smith@eecs	18	3

Students

Views – WITH CHECK OPTION

- It is applicable to a updatable view
- The WITH CHECK OPTION clause is used to prevent the insertion of rows in the view where the condition in the WHERE clause in CREATE VIEW statement is not satisfied.
- If we have used the WITH CHECK OPTION clause in the CREATE VIEW statement, and if the UPDATE or INSERT clause does not satisfy the conditions then they will return an error.

sid	name	login	age	gpa
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@eecs	18	3.2
53650	Smith	smith@math	19	3.8

sid	cid	grade
53831	Carnatic101	C
53831	Reggae203	B
53650	Topology112	A
53666	History105	B

```
CREATE  VIEW  YoungActiveStudents (name, grade)
        AS  SELECT   S.name, E.grade
        FROM  Students S, Enrolled E WHERE  S.sid = E.sid and S.age<21
        WITH CHECK OPTION
UPDATE YoungActiveStudents SET age = 28 WHERE sid = '53650';
```

Error Code: 1369. CHECK OPTION failed 'students_db.youngactivestudents'

The View is updatable but the query is not valid

Summary

- Views useful for security, logical data independence, performance
- Stored logically (query modification required) or physically (materialized)
- View updates must be unambiguously mappable to base relation updates in order to be allowed.
- Most systems don’t allow as many view updates as they could

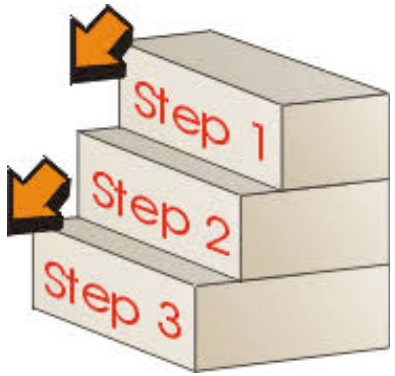
State of the Art (views)

- Views are becoming important for processing "**decision support**" queries
- Automated view creation and management (based on evolving workload)
- View and trigger interactions (semantics, optimization)
- Views for answering aggregation queries (query modification algorithms, etc.)
- Views to integrate multiple data sources
- Algorithms for deferred view maintenance

University College Cork, Ireland
Coláiste na hOllscoile Corcaigh

Information Storage and Management I

Dr. Alejandro Arbelaez



Stored Procedures

Stored Procedure

- A stored procedure is a program with SQL code which is stored in the database catalog and can be invoked later by a program, a trigger or even a stored procedure.
- MySQL supports stored procedure since version 5.0 to allow MySQL more flexible and powerful.

A stored produced is a function that performs more complex logic inside of the DBMS

- Can have many input/output parameters
- Can modify the database table/structures
- Not normally used within a SQL query

Stored Procedures

- Database program modules that are stored and executed by the DBMS at the server

You must redefine the delimiter temporarily to cause **mysql** to pass the entire stored program definition to the server.

```
DELIMITER //
```

The delimiter is changed to // to enable the entire definition to be passed to the server as a single statement. It can be restored to ";"
mysql>delimiter ;

```
DELIMITER ;
```

Stored Procedures

- Database program modules that are stored and executed by the DBMS at the server

You must redefine the delimiter temporarily to cause **mysql** to pass the entire stored program definition to the server.

```
DELIMITER //
CREATE PROCEDURE dorepeat(p1 INT, INOUT x)
```

The delimiter is changed to // to enable the entire definition to be passed to the server as a single statement. It can be restored to ";"
mysql>delimiter ;

```
DELIMITER ;
```

Stored Procedures

- Database program modules that are stored and executed by the DBMS at the server

You must redefine the delimiter temporarily to cause **mysql** to pass the entire stored program definition to the server.

```
DELIMITER //
CREATE PROCEDURE dorepeat(p1 INT, INOUT x)
BEGIN



END;
DELIMITER ;
```

The delimiter is changed to // to enable the entire definition to be passed to the server as a single statement. It can be restored to ";"
mysql>delimiter ;

Stored Procedures

- Database program modules that are stored and executed by the DBMS at the server

You must redefine the delimiter temporarily to cause **mysql** to pass the entire stored program definition to the server.

```
DELIMITER //
CREATE PROCEDURE dorepeat(p1 INT, INOUT x)
BEGIN
        SET x = 0;
        REPEAT
                SET x = x + 1;
        UNTIL x > p1
        END REPEAT;
END;
DELIMITER ;
```

The delimiter is changed to // to enable the entire definition to be passed to the server as a single statement. It can be restored to ";"
mysql>delimiter ;

Stored Procedures

- Database program modules that are stored and executed by the DBMS at the server

```
DELIMITER //
CREATE PROCEDURE GetAllProducts()
  BEGIN
  SELECT *  FROM products;
  END //
DELIMITER ;
```

The delimiter is changed to // to enable the entire definition to be passed to the server as a single statement. It can be restored to ";"
mysql>delimiter ;

Why Stored Procedures

- Reduces Duplication of effort and improves software modularity
  - Multiple applications can use the stored procedure vs. the SQL statements being stored in the application language (Python or PHP)
- Reduces communication and data transfer cost between client and server (in certain situations)
  - Instead of sending multiple lengthy SQL statements, the application only has to send the name and parameters of the stored procedure
- Can be more secure than SQL statements
  - Permission can be granted to certain stored procedures without granting access to databases tables

Disadvantages of Stored Procedures

- Difficult to debug
  - MySQL does not provide ways for debugging stored procedures
- Many stored procedures can increase memory use
  - The more stored procedures you use, the more memory is used
- Can be difficult to maintain and develop stored procedures
  - Another programming language to learn

Creating Stored Procedures

```
DELIMITER //
CREATE PROCEDURE NAME
  BEGIN
        SQL STATEMENT
  END //
DELIMITER ;
```

```
DELIMITER //
CREATE PROCEDURE GetAllProducts()
  BEGIN
  SELECT *  FROM products;
  END //
DELIMITER ;
```

Calling Stored Procedures

```
CALL STORED_PROCEDURE_NAME

CALL GetAllProducts();
```

Variables

- A variable is a name that refers to a value
- Python:

  name = “Alex”

  age = 35
- MySQL

  DECLARE name VARCHAR(225)

  DECLARE age INT

Define parameters within a stored procedure

- Parameter list is empty
  - CREATE PROCEDURE proc1 () :
- Define input parameter with key word IN:
  - CREATE PROCEDURE proc1 (IN varname DATA-TYPE)
  - The word IN is optional because parameters are IN (input) by default.
- Define output parameter with OUT:
  - CREATE PROCEDURE proc1 (OUT varname DATA-TYPE)
- A procedure may have input and output paramters:
  - CREATE PROCEDURE proc1 (INOUT varname DATA-TYPE)

Three Types of Parameters

- **IN**
  - **Default**
- OUT
- INOUT

In Parameter

- Calling program has to pass an argument to the stored procedure.

Arguments and Parameters

```
DELIMITER //
CREATE PROCEDURE GetOfficeByCountry(IN countryName VARCHAR(255))
 BEGIN
 SELECT *  FROM offices WHERE country = countryName;
 END //
DELIMITER ;
```

Defining

```
CALL GetOfficeByCountry('USA')
```

Calling

Three Types of Parameters

- IN
  - Default
- **OUT**
- INOUT

Out Parameter

- **OUT** – the value of an OUT parameter can be changed inside the stored procedure and its new value is passed back to the calling program
- **OUT** is a keyword

Out Parameter

```
DELIMITER //
CREATE PROCEDURE CountOrderByStatus(IN orderStatus VARCHAR(25), OUT total INT)
BEGIN
 SELECT count(orderNumber) INTO total FROM orders WHERE status = orderStatus;
END//
DELIMITER ;
```

Defining

```
CALL CountOrderByStatus('Shipped',@total);

SELECT @total;
```

The out parameter is used outside of the stored procedure.

User-Defined Temporary Variables

User variables are written as @var_name.

```
mysql> SET @t1=1, @t2=2, @t3:=4;
mysql> SELECT @t1, @t2, @t3, @t4 := @t1+@t2+@t3;
+------+------+------+--------------------+
| @t1  | @t2  | @t3  | @t4 := @t1+@t2+@t3 |
+------+------+------+--------------------+
|    1 |    2 |    4 |                  7 |
+------+------+------+--------------------+
```

Example of running the procedure from the command prompt

```
mysql> delimiter ;
mysql> set @tax=0;
Query OK, 0 rows affected (0.00 sec)

mysql> call caltax('S1',0.1,@tax);
Query OK, 1 row affected (0.00 sec)

mysql> select @tax;
+------+
| @tax |
+------+
|  650 |
+------+
1 row in set (0.00 sec)
```

Three Types of Parameters

- IN
  - Default
- OUT
- INOUT

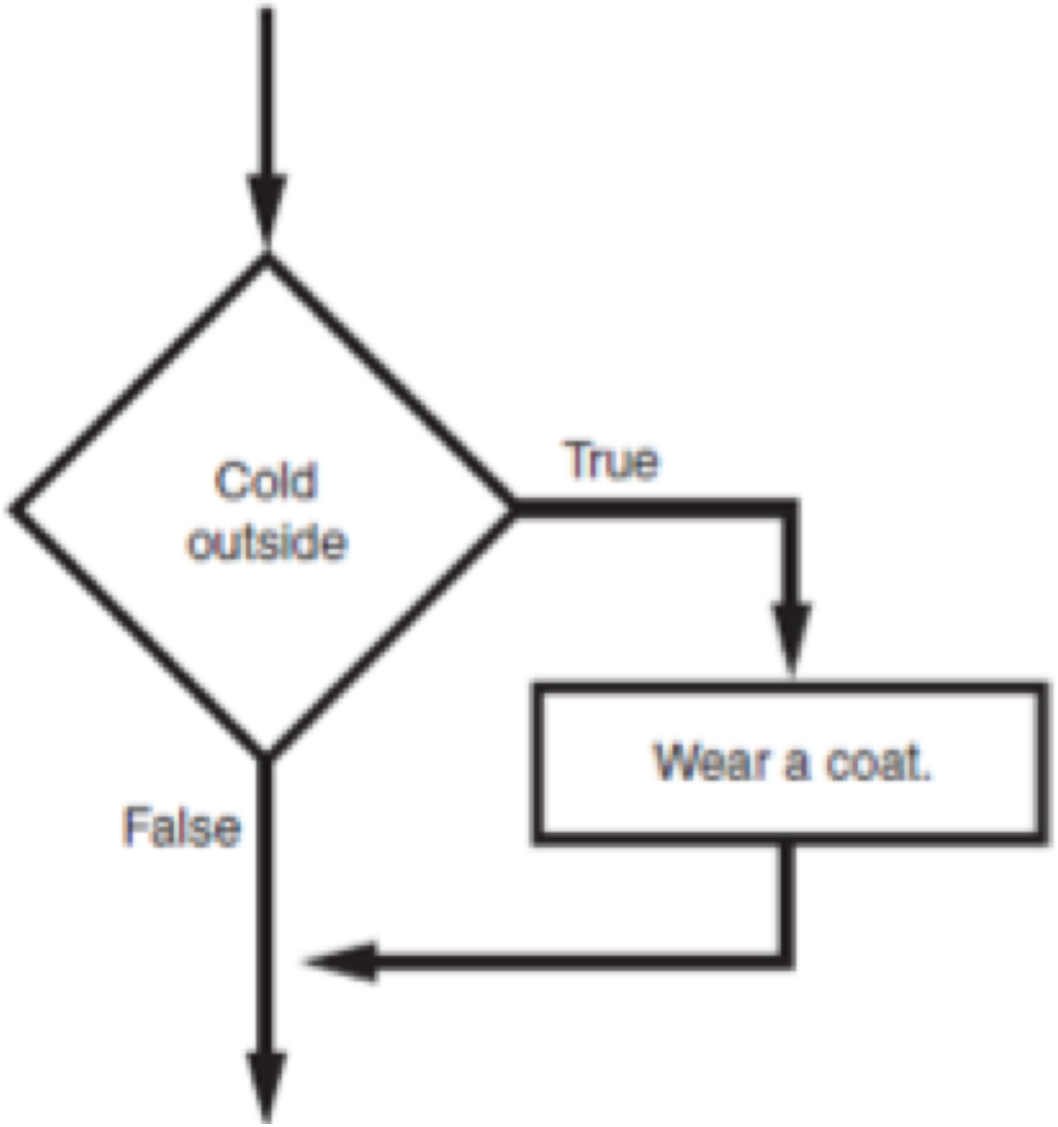
Examples of parameters

```
CREATE PROCEDURE proc_IN (IN var1 INT)
BEGIN
   SELECT var1 + 2 AS result;
END

CREATE PROCEDURE proc_OUT(OUT var1 VARCHAR(100))
BEGIN
 SET var1 = 'This is a test';
END

CREATE PROCEDURE proc_INOUT (IN var1 INT,OUT var2 INT)
BEGIN
   SET var2 = var1 * 2;
END
```

Conditionals



The “If” Statement (MySQL Syntax)

IF **if_expression** THEN commands

  [ELSEIF **elseif_expression** THEN commands]

  [ELSE commands]

END IF;

MySQL Comparison Operators

- EQUAL(=)
- LESS THAN(<)
- LESS THAN OR EQUAL(<=)
- GREATER THAN(>)
- GREATER THAN OR EQUAL(>=)
- NOT EQUAL(<>,!=)

“If Expression”: BOOLEAN Expressions and Operators

Name	Description
BETWEEN ... AND ...	Check whether a value is within a range of values
COALESCE()	Return the first non-NULL argument
<=>	NULL-safe equal to operator
=	Equal operator
>=	Greater than or equal operator
>	Greater than operator
GREATEST()	Return the largest argument
IN()	Check whether a value is within a set of values
INTERVAL()	Return the index of the argument that is less than the first argument
IS NOT NULL	NOT NULL value test
IS NOT	Test a value against a boolean
IS NULL	NULL value test
IS	Test a value against a boolean
ISNULL()	Test whether the argument is NULL
LEAST()	Return the smallest argument
<=	Less than or equal operator
<	Less than operator
LIKE	Simple pattern matching
NOT BETWEEN ... AND ...	Check whether a value is not within a range of values
!=, <>	Not equal operator
NOT IN()	Check whether a value is not within a set of values
NOT LIKE	Negation of simple pattern matching
STRCMP()	Compare two strings

Logical Operators

- Logical AND:
  - AND, &&
  - UnitsInStock < ReorderLevel AND CategoryID=1
  - UnitsInStock < ReorderLevel && CategoryID=1
- Negates value:
  - NOT, !
- Logical OR:
  - ||, OR
  - CategoryID=1 OR CategoryID=8
  - CategoryID=1 || CategoryID=8

IF Statement

```
DELIMITER //
CREATE PROCEDURE GetProductsInStockBasedOnQuantitityLevel(IN
p_operator VARCHAR(255), IN p_quantityInStock INT)
 BEGIN
  IF p_operator = "<" THEN
        select * from products WHERE quantityInStock < p_quantityInStock;
  ELSEIF p_operator = ">" THEN
        select * from products WHERE quantityInStock > p_quantityInStock;
  END IF;
 END //
DELIMITER ;
```

IF Statement

```
CREATE PROCEDURE GetProductsInStockBasedOnQuantitityLevel
(IN p_operator VARCHAR(255), IN p_quantityInStock INT)
```

The operator > or <

The number in stock

The IF Statement

```
IF p_operator = "<" THEN
        select * from products WHERE quantityInStock <p_quantityInStock;

ELSEIF p_operator = ">" THEN
        select * from products WHERE quantityInStock > p_quantityInStock;
END IF;
```

Loops

- While
- Repeat
- Loop

Repeats a set of commands until some conditions is met

Iteration: one execution of the body of a loop

If a condition is never met, we will have a infinite loop

While Loop

WHILE expression DO

Statements

END WHILE

The expression must evaluate to true or false

while loop is known as a *pretest* loop

Tests condition before performing an iteration

- Will never execute if condition is false to start with
- Requires performing some steps prior to the loop

Infinite Loops

- Loops must contain within themselves a way to terminate
  - Something inside a while loop must eventually make the condition false
- Infinite loop: loop that does not have a way of stopping
  - Repeats until program is interrupted
  - Occurs when programmer forgets to include stopping code in the loop

While Loop

```
DELIMITER //
CREATE PROCEDURE WhileLoopProc()
    BEGIN
        DECLARE x  INT;
        DECLARE str  VARCHAR(255);
        SET x = 1;
        SET str =  '';
        WHILE x  <= 5 DO
            SET  str = CONCAT(str,x,',');
            SET  x = x + 1;
        END WHILE;
        SELECT str;
    END//
  DELIMITER ;
```

While Loop

Creating Variables

```
DECLARE x  INT;
DECLARE str  VARCHAR(255);
 SET x = 1;
 SET str =  '';
```

While Loop

```
WHILE x  <= 5 DO
                SET  str = CONCAT(str,x,',');
                SET  x = x + 1;
END WHILE;
```

Cursors

- A cursor is a pointer to a set of records returned by a SQL statement. It enables you to take a set of records and deal with it on a row-by-row basis.
- To handle a result set inside a stored procedure, you use a cursor. A cursor allows you to iterate a set of rows returned by a query and process each row individually.

Cursor has three important properties

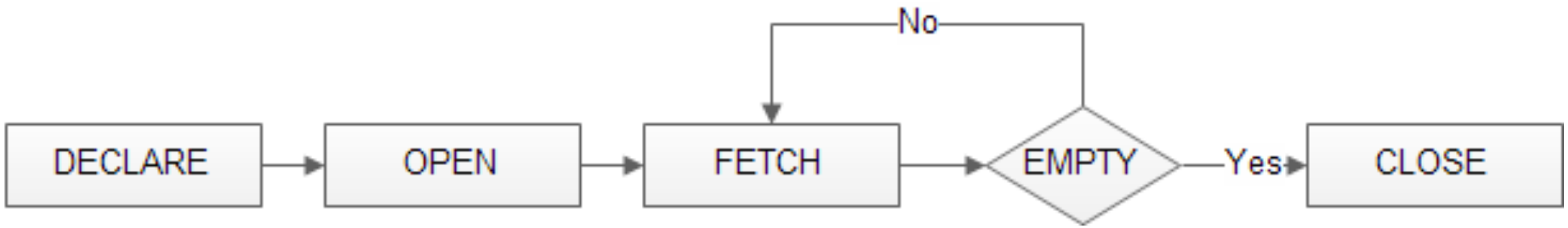
- The cursor will not reflect changes in its source tables.
- Read Only : Cursors are not updatable.
- Not Scrollable : Cursors can be traversed only in one direction, forward, and you can't skip records from fetching.

Defining and Using Cursors

- Declare cursor:
  - DECLARE cursor-name CURSOR FOR SELECT ...;
- DECLARE CONTINUE HANDLER FOR NOT FOUND: Specify what to do when no more records found
  - DECLARE b INT;
  - DECLARE CONTINUE HANDLER FOR NOT FOUND SET b = 1;
- Open cursor:
  - OPEN cursor-name;
- Fetch data into variables:
  - FETCH cursor-name INTO variable [, variable];
- CLOSE cursor:
  - CLOSE cursor-name;

Use the FETCH statement to retrieve the next row pointed by the cursor and move the cursor to the next row in the result set.

MySQL Cursor



A procedure to create email list using cursor

Concatenate all emails where each email is separated by a semicolon(;):

Employee

	123 empNo	ABC name	ABC email
1	1	E1	E1@gmail.com
2	2	E2	E2@gmail.com
3	3	E3	E3@gmail.com
4	4	E4	E4@gmail.com

E1@gmail.com; E2@gmail.com; E3@gmail.com; E3@gmail.com;

Source:
http://www.mysqltutorial.org/mysql-cursor/

A procedure to create email list using cursor

```
DELIMITER //
CREATE PROCEDURE createEmailList (INOUT emailList VARCHAR(5000))
BEGIN
    DECLARE finish INT;
    DECLARE emailAddress VARCHAR(20);

    #declare cursor for employee email
    DECLARE curEmail CURSOR FOR SELECT email from EMPLOYEES;

    #declare NOT FOUND handler
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET finish = 1;

    OPEN curEmail;

    getEmail: LOOP
        FETCH curEmail INTO emailAddress;
        IF finish = 1 THEN
            LEAVE getEmail;
        END IF;
        SET emailList = CONCAT(emailAddress, ";", emailList);
    END LOOP getEmail;

    CLOSE curEmail;
END //
DELIMITER ;

SET @emailList = "";
CALL createEmailList(@emailList);
SELECT @emailList;
```

The cursor declaration must be after any variable declaration

Source: http://www.mysqltutorial.org/mysql-cursor/

A procedure to create email list using cursor

```
DELIMITER //
CREATE PROCEDURE createEmailList (INOUT emailList VARCHAR(5000))
BEGIN
    DECLARE finish INT;
    DECLARE emailAddress VARCHAR(20);

    #declare cursor for employee email
    DECLARE curEmail CURSOR FOR SELECT email from EMPLOYEES;

    #declare NOT FOUND handler
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET finish = 1;

    OPEN curEmail;

    getEmail: LOOP
        FETCH curEmail INTO emailAddress;
        IF finish = 1 THEN
            LEAVE getEmail;
        END IF;
        SET emailList = CONCAT(emailAddress, ";", emailList);
    END LOOP getEmail;

    CLOSE curEmail;
END //
DELIMITER ;

SET @emailList = "";
CALL createEmailList(@emailList);
SELECT @emailList;
```

The cursor declaration must be after any variable declaration
Cursor for employee email

NOT FOUND handler

Iterate the email list, and concatenate all emails where each email is separated by a semicolon(;)

Source: http://www.mysqltutorial.org/mysql-cursor/

A procedure to create email list using cursor

```
DELIMITER //
CREATE PROCEDURE createEmailList (INOUT emailList VARCHAR(5000))
BEGIN
    DECLARE finish INT;
    DECLARE emailAddress VARCHAR(20);

    #declare cursor for employee email
    DECLARE curEmail CURSOR FOR SELECT email from EMPLOYEES;

    #declare NOT FOUND handler
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET finish = 1;

    OPEN curEmail;

    getEmail: LOOP
        FETCH curEmail INTO emailAddress;
        IF finish = 1 THEN
            LEAVE getEmail;
        END IF;
        SET emailList = CONCAT(emailAddress, ";", emailList);
    END LOOP getEmail;

    CLOSE curEmail;
END //
DELIMITER ;

SET @emailList = "";
CALL createEmailList(@emailList);
SELECT @emailList;
```

curEmail

	123 empNo	ABC name	ABC email
1	1	E1	E1@gmail.com
2	2	E2	E2@gmail.com
3	3	E3	E3@gmail.com
4	4	E4	E4@gmail.com

emailList = “”
finish = 0

A procedure to create email list using cursor

```
DELIMITER //
CREATE PROCEDURE createEmailList (INOUT emailList VARCHAR(5000))
BEGIN
    DECLARE finish INT;
    DECLARE emailAddress VARCHAR(20);

    #declare cursor for employee email
    DECLARE curEmail CURSOR FOR SELECT email from EMPLOYEES;

    #declare NOT FOUND handler
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET finish = 1;

    OPEN curEmail;

    getEmail: LOOP
        FETCH curEmail INTO emailAddress;
        IF finish = 1 THEN
            LEAVE getEmail;
        END IF;
        SET emailList = CONCAT(emailAddress, ";", emailList);
    END LOOP getEmail;

    CLOSE curEmail;
END //
DELIMITER ;

SET @emailList = "";
CALL createEmailList(@emailList);
SELECT @emailList;
```

curEmail

	123 empNo	ABC name	ABC email
1	1	E1	E1@gmail.com
2	2	E2	E2@gmail.com
3	3	E3	E3@gmail.com
4	4	E4	E4@gmail.com

emailList = “”
finish = 0
emailAddress = E1@gmail.com

A procedure to create email list using cursor

```
DELIMITER //
CREATE PROCEDURE createEmailList (INOUT emailList VARCHAR(5000))
BEGIN
    DECLARE finish INT;
    DECLARE emailAddress VARCHAR(20);

    #declare cursor for employee email
    DECLARE curEmail CURSOR FOR SELECT email from EMPLOYEES;

    #declare NOT FOUND handler
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET finish = 1;

    OPEN curEmail;

    getEmail: LOOP
        FETCH curEmail INTO emailAddress;
        IF finish = 1 THEN
            LEAVE getEmail;
        END IF;
        SET emailList = CONCAT(emailAddress, ";", emailList);
    END LOOP getEmail;

    CLOSE curEmail;
END //
DELIMITER ;

SET @emailList = "";
CALL createEmailList(@emailList);
SELECT @emailList;
```

curEmail

	123 empNo	ABC name	ABC email
1	1	E1	E1@gmail.com
2	2	E2	E2@gmail.com
3	3	E3	E3@gmail.com
4	4	E4	E4@gmail.com

finish = 0
emailAddress = E1@gmail.com
emailList = “E1@gmail.com;”

A procedure to create email list using cursor

```
DELIMITER //
CREATE PROCEDURE createEmailList (INOUT emailList VARCHAR(5000))
BEGIN
    DECLARE finish INT;
    DECLARE emailAddress VARCHAR(20);

    #declare cursor for employee email
    DECLARE curEmail CURSOR FOR SELECT email from EMPLOYEES;

    #declare NOT FOUND handler
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET finish = 1;

    OPEN curEmail;

    getEmail: LOOP
        FETCH curEmail INTO emailAddress;
        IF finish = 1 THEN
            LEAVE getEmail;
        END IF;
        SET emailList = CONCAT(emailAddress, ";", emailList);
    END LOOP getEmail;

    CLOSE curEmail;
END //
DELIMITER ;

SET @emailList = "";
CALL createEmailList(@emailList);
SELECT @emailList;
```

curEmail

	123 empNo	ABC name	ABC email
1	1	E1	E1@gmail.com
2	2	E2	E2@gmail.com
3	3	E3	E3@gmail.com
4	4	E4	E4@gmail.com

finish = 0
emailAddress = E2@gmail.com
emailList = “E1@gmail.com;”

A procedure to create email list using cursor

```
DELIMITER //
CREATE PROCEDURE createEmailList (INOUT emailList VARCHAR(5000))
BEGIN
    DECLARE finish INT;
    DECLARE emailAddress VARCHAR(20);

    #declare cursor for employee email
    DECLARE curEmail CURSOR FOR SELECT email from EMPLOYEES;

    #declare NOT FOUND handler
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET finish = 1;

    OPEN curEmail;

    getEmail: LOOP
        FETCH curEmail INTO emailAddress;
        IF finish = 1 THEN
            LEAVE getEmail;
        END IF;
        SET emailList = CONCAT(emailAddress, ";", emailList);
    END LOOP getEmail;

    CLOSE curEmail;
END //
DELIMITER ;

SET @emailList = "";
CALL createEmailList(@emailList);
SELECT @emailList;
```

curEmail

	123 empNo	ABC name	ABC email
1	1	E1	E1@gmail.com
2	2	E2	E2@gmail.com
3	3	E3	E3@gmail.com
4	4	E4	E4@gmail.com

finish = 0
emailAddress = E2@gmail.com
emailList = "E1@gmail.com; E2@gmail.com"

A procedure to create email list using cursor

```
DELIMITER //
CREATE PROCEDURE createEmailList (INOUT emailList VARCHAR(5000))
BEGIN
    DECLARE finish INT;
    DECLARE emailAddress VARCHAR(20);

    #declare cursor for employee email
    DECLARE curEmail CURSOR FOR SELECT email from EMPLOYEES;

    #declare NOT FOUND handler
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET finish = 1;

    OPEN curEmail;

    getEmail: LOOP
        FETCH curEmail INTO emailAddress;
        IF finish = 1 THEN
            LEAVE getEmail;
        END IF;
        SET emailList = CONCAT(emailAddress, ";", emailList);
    END LOOP getEmail;

    CLOSE curEmail;
END //
DELIMITER ;

SET @emailList = "";
CALL createEmailList(@emailList);
SELECT @emailList;
```

curEmail

	123 empNo	ABC name	ABC email
1	1	E1	E1@gmail.com
2	2	E2	E2@gmail.com
3	3	E3	E3@gmail.com
4	4	E4	E4@gmail.com

finish = 0
emailAddress = E3@gmail.com
emailList = "E1@gmail.com; E2@gmail.com"

A procedure to create email list using cursor

```
DELIMITER //
CREATE PROCEDURE createEmailList (INOUT emailList VARCHAR(5000))
BEGIN
    DECLARE finish INT;
    DECLARE emailAddress VARCHAR(20);

    #declare cursor for employee email
    DECLARE curEmail CURSOR FOR SELECT email from EMPLOYEES;

    #declare NOT FOUND handler
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET finish = 1;

    OPEN curEmail;

    getEmail: LOOP
        FETCH curEmail INTO emailAddress;
        IF finish = 1 THEN
            LEAVE getEmail;
        END IF;
        SET emailList = CONCAT(emailAddress, ";", emailList);
    END LOOP getEmail;

    CLOSE curEmail;
END //
DELIMITER ;

SET @emailList = "";
CALL createEmailList(@emailList);
SELECT @emailList;
```

curEmail

	123 empNo	ABC name	ABC email
1	1	E1	E1@gmail.com
2	2	E2	E2@gmail.com
3	3	E3	E3@gmail.com
4	4	E4	E4@gmail.com

finish = 0
emailAddress = E3@gmail.com
emailList = "E1@gmail.com; E2@gmail.com;
E3@gmail.com;
"

A procedure to create email list using cursor

```
DELIMITER //
CREATE PROCEDURE createEmailList (INOUT emailList VARCHAR(5000))
BEGIN
    DECLARE finish INT;
    DECLARE emailAddress VARCHAR(20);

    #declare cursor for employee email
    DECLARE curEmail CURSOR FOR SELECT email from EMPLOYEES;

    #declare NOT FOUND handler
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET finish = 1;

    OPEN curEmail;

    getEmail: LOOP
        FETCH curEmail INTO emailAddress;
        IF finish = 1 THEN
            LEAVE getEmail;
        END IF;
        SET emailList = CONCAT(emailAddress, ";", emailList);
    END LOOP getEmail;

    CLOSE curEmail;
END //
DELIMITER ;

SET @emailList = "";
CALL createEmailList(@emailList);
SELECT @emailList;
```

curEmail

	123 empNo	ABC name	ABC email
1	1	E1	E1@gmail.com
2	2	E2	E2@gmail.com
3	3	E3	E3@gmail.com
4	4	E4	E4@gmail.com

finish = 0
emailAddress = E4@gmail.com
emailList = "E1@gmail.com; E2@gmail.com;
E3@gmail.com;
"

A procedure to create email list using cursor

```
DELIMITER //
CREATE PROCEDURE createEmailList (INOUT emailList VARCHAR(5000))
BEGIN
    DECLARE finish INT;
    DECLARE emailAddress VARCHAR(20);

    #declare cursor for employee email
    DECLARE curEmail CURSOR FOR SELECT email from EMPLOYEES;

    #declare NOT FOUND handler
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET finish = 1;

    OPEN curEmail;

    getEmail: LOOP
        FETCH curEmail INTO emailAddress;
        IF finish = 1 THEN
            LEAVE getEmail;
        END IF;
        SET emailList = CONCAT(emailAddress, ";", emailList);
    END LOOP getEmail;

    CLOSE curEmail;
END //
DELIMITER ;

SET @emailList = "";
CALL createEmailList(@emailList);
SELECT @emailList;
```

curEmail

	123 empNo	ABC name	ABC email
1	1	E1	E1@gmail.com
2	2	E2	E2@gmail.com
3	3	E3	E3@gmail.com
4	4	E4	E4@gmail.com

finish = 0
emailAddress = E4@gmail.com
emailList = "E1@gmail.com; E2@gmail.com; E3@gmail.com; E4@gmail.com"

A procedure to create email list using cursor

```
DELIMITER //
CREATE PROCEDURE createEmailList (INOUT emailList VARCHAR(5000))
BEGIN
    DECLARE finish INT;
    DECLARE emailAddress VARCHAR(20);

    #declare cursor for employee email
    DECLARE curEmail CURSOR FOR SELECT email from EMPLOYEES;

    #declare NOT FOUND handler
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET finish = 1;

    OPEN curEmail;

    getEmail: LOOP
        FETCH curEmail INTO emailAddress;
        IF finish = 1 THEN
            LEAVE getEmail;
        END IF;
        SET emailList = CONCAT(emailAddress, ";", emailList);
    END LOOP getEmail;

    CLOSE curEmail;
END //
DELIMITER ;

SET @emailList = "";
CALL createEmailList(@emailList);
SELECT @emailList;
```

	123 empNo	ABC name	ABC email
1	1	E1	E1@gmail.com
2	2	E2	E2@gmail.com
3	3	E3	E3@gmail.com
4	4	E4	E4@gmail.com

curEmail

finish = 1
emailAddress = E4@gmail.com
emailList = "E1@gmail.com; E2@gmail.com; E3@gmail.com; E4@gmail.com"

A procedure to create email list using cursor

```
DELIMITER //
CREATE PROCEDURE createEmailList (INOUT emailList VARCHAR(5000))
BEGIN
    DECLARE finish INT;
    DECLARE emailAddress VARCHAR(20);

    #declare cursor for employee email
    DECLARE curEmail CURSOR FOR SELECT email from EMPLOYEES;

    #declare NOT FOUND handler
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET finish = 1;

    OPEN curEmail;

    getEmail: LOOP
        FETCH curEmail INTO emailAddress;
        IF finish = 1 THEN
            LEAVE getEmail;
        END IF;
        SET emailList = CONCAT(emailAddress, ";", emailList);
    END LOOP getEmail;

    CLOSE curEmail;
END //
DELIMITER ;

SET @emailList = "";
CALL createEmailList(@emailList);
SELECT @emailList;
```

Finish this loop

	123 empNo	ABC name	ABC email
1	1	E1	E1@gmail.com
2	2	E2	E2@gmail.com
3	3	E3	E3@gmail.com
4	4	E4	E4@gmail.com

curEmail

finish = 1
emailAddress = E4@gmail.com
emailList = "E1@gmail.com; E2@gmail.com; E3@gmail.com; E4@gmail.com"