**CSCI 4333.2**

10/21/2024

**More SQL**

by K. Yue

**1. More SQL Features**

**1.1 Prepared Statements**

* MySQL supports server-side prepared statements.
* A prepared statement is used in the following sequence.
	1. The statement is prepared. (compilation and creating an execution plan)
	2. The prepared statements are executed one to many times, including using different values on placeholders.
	3. The statement is deallocated.
* Prepared Statements have the following benefits:
	1. They are compiled once and can be executed many times.
	2. With the use of placeholders, they protect against SQL injections.
	3. They can be used to execute dynamic SQL statements.
		1. However, dynamic SQL statements can be serious security holes and should be used carefully.
* In MySQL, prepared statements can be used in
	1. Applications: such as SQL clients (e.g. MySQL prompt) or Python programs.
	2. SQL scripts: such as invoked by the source command.
* There are vendor-dependent restrictions in prepared statements.
* For examples, in MySQL,
	1. Constants or user variables should be used as the parameter values to the placeholders of the prepared statements.
	2. user variables should be used in the INTO clause of the SQL query in the prepared SELECT statement.

***Example:***

Try the following code in MySQL prompt.

-- Prepared statements.
SET @sql = "SELECT \* FROM toyu.student";
PREPARE stmt FROM @sql;
EXECUTE stmt;
DEALLOCATE PREPARE stmt;

-- with placeholders.
SET @sql = "SELECT \* FROM toyu.student WHERE major = ? AND ach >= ?";
PREPARE stmt FROM @sql;
SET @major = 'CSCI';
SET @ach = 38;
EXECUTE stmt USING @major, @ach;

EXECUTE stmt USING 'CSCI', 38;

SET @major = 'CINF';
SET @ach = 15;
EXECUTE stmt USING @major, @ach;

SET @major = 'ITEC';
SET @ach = 25;
EXECUTE stmt USING @major, @ach;

DEALLOCATE PREPARE stmt;

**2. Views**

* Views are *virtual* tables *derived* from other tables.
* In MySQL, "views are stored queries that when invoked produce a *result set*. A view acts as a virtual table." See: <https://dev.mysql.com/doc/refman/8.2/en/views.html>
* Some advantages of using views:
	1. Better data abstraction: hiding unnecessary information.
	2. Logical data independence
	3. Better consistency
	4. More security control
	5. Possibly more efficient
* Some disadvantages:
	1. More work
	2. Complicated especially when views are updated.
	3. Performance: view processing may not be optimized, e.g.,
	4. A view can be treated as a subquery by the db engine.
	5. Cannot create index for view in MySQL.
* See the [three layered DB architecture](https://dcm.uhcl.edu/yue/courses/joinDB/Fall2024/notes/intro/DBBasics.html).
* MySQL Create View Manual: search for "mysql view manual"

CREATE
    [OR REPLACE]
    [ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}]
    [DEFINER = { user | CURRENT\_USER }]
    [SQL SECURITY { DEFINER | INVOKER }]
    VIEW view\_name [(column\_list)]
    AS select\_statement
    [WITH [CASCADED | LOCAL] CHECK OPTION]

* There are some limitations in views. For example:
	+ In earlier versions of MySQL, subqueries are not allowed in the SELECT clause in the CREATE VIEW statement.
* You may need to define intermediate views accordingly.

***Example:***

Execute the following code and ensure that you understand the result.

CREATE OR REPLACE VIEW school\_summary(
   schoolCode, schoolName, n\_departments) AS
SELECT s.schoolCode, s.schoolName,
COUNT(d.deptCode) AS n\_departments
FROM school AS s LEFT JOIN department AS d ON (s.schoolCode = d.schoolCode)
GROUP BY s.schoolCode, s.schoolName;

SHOW CREATE VIEW school\_summary;
-- Note something like "ALGORITHM=UNDEFINED DEFINER=`yue`@`localhost` SQL SECURITY DEFINER"
-- (default values) may be added.

DESC school\_summary;

SELECT \*
FROM school\_summary
WHERE n\_departments > 0;

DROP VIEW school\_summary;

Notes:

* "CREATE OR REPLACE" can replace existing view definition.
* There are no data types explicitly specified for the columns in the example.
* The column list is optional.
* MySQL supports specification of algorithm for processing views:
	+ MERGE: view definition merged into the containing query (inlining)
	+ TEMPTABLE: view results stored in temporary table to be used in the containing query.
* MySQL supports updatable views. Use them carefully. E.g., the view 'user' in 'mysql' is updatable.
	+ In general, "for a view to be updatable, there must be a one-to-one relationship between the rows in the view and the rows in the underlying table."

**3. Stored Subroutines**

* Subroutines stored by the DBMS can be called.
* Subroutines can be
	1. functions: return a value, or
		1. black box
		2. should not have any side effects.
	2. procedures:
		1. do not return a value, and
		2. work by side effects.
* Known as SQL/PSM (SQL/Persistent Stored Modules) in MySQL.
	1. Can execute SQL statements.
	2. Include general programming constructs.
* Some advantages of stored subroutines:
	1. Enforcing application constraints and requirements.
	2. Providing consistency and security control.
	3. Possible performance optimization by both the developers and the DBMS.
	4. Sharing among DB applications, especially when they use different languages.
* Some disadvantages of stored procedures:
	1. Use up DB server's resources.
	2. Potentially inefficient because of limitations in language constructs.
	3. Potentially harder to develop because of relative lack of libraries and difficulty in debugging.
	4. Can be vendor specific.
* In general, stored subroutines should be used more often.
* Some general points:
	1. A stored subroutine is associated with a particular database.
	2. Stored functions cannot be recursive.

**3.1 Stored Procedures**

***Example:***

Execute the following code and ensure that you understand the result.

-- A very simple stored procedure.
-- Redefine the delimiter to end the procedure.
DELIMITER //

-- IN: pass by value; OUT: copy out hen exiting
CREATE OR REPLACE PROCEDURE deptInfo(IN dCode VARCHAR(4), OUT numFaculty INT)
BEGIN
   --  Display some information.

 -- e.g. dCode: ‘CSCI’; numFaculty: null
   SELECT d.deptName, d.SchoolCode, t1.n\_majors, t2.n\_minors
   FROM department AS d INNER JOIN
      (SELECT COUNT(stuId) AS n\_majors
      FROM student
      WHERE major = dCode) AS t1 INNER JOIN
      (SELECT COUNT(stuId) AS n\_minors
      FROM student
      WHERE minor = dCode) AS t2
   WHERE d.deptCode = dCode;

   -- MySQL does not direct sending output to console.
   -- It is necessary to use a SQL statement.
   SELECT 'Debuggin comment can be put here.';
   SELECT CONCAT('Faculty in the department: ', dCode) AS faculty;

   SELECT \*
   FROM faculty AS f
   WHERE f.deptCode = dCode;

   SELECT COUNT(f.facId) INTO numFaculty
   FROM faculty AS f
   WHERE f.deptCode = dCode;
END //

DELIMITER ;

SHOW CREATE PROCEDURE deptInfo;

SET @numFaculty = 0;
SET @dCode = 'CSCI';
CALL deptInfo(@dCode, @numFaculty);
SELECT @dCode, @numFaculty;

SET @dCode = 'ITEC';
CALL deptInfo(@dCode, @numFaculty);
SELECT @dCode, @numFaculty;

DROP PROCEDURE deptInfo;

Note:

1. Use of the DELIMITER command to redefine '//' as the delimiter indicating the end of the stored procedure. Otherwise, the default ';' is the delimiter indicating the end of the stored procedure.
2. Two parameters for this procedure: one using IN and one using OUT as the parameter passing mechanism.
3. A procedure does not return any value and accomplish its goal through *side effects*.
4. Side effects include:
	1. Return SELECT results in the procedure body.
	2. Copy numFaculty out upon completion.

***Example: using toyu***

DELIMITER //
CREATE OR REPLACE PROCEDURE AddNewCourse(
    IN course\_id INT,
    IN rubric CHAR(4),
    IN course\_number CHAR(4),
    IN course\_title VARCHAR(80),
    IN credits TINYINT
)
BEGIN
    INSERT INTO Course (courseId, rubric, number, title, credits)
    VALUES (course\_id, rubric, course\_number, course\_title, credits);
END //
DELIMITER ;

CALL AddNewCourse(3009, 'CSCI', '4436', 'Systems Administration', 3);

**3.2 Stored Functions**

***Example:***

Execute the following code and ensure that you understand the result.

-- A simple function
DELIMITER //

CREATE OR REPLACE FUNCTION n\_major(dCode varchar(4)) RETURNS INT
READS SQL DATA
BEGIN
   DECLARE count INT DEFAULT 0;

   SELECT COUNT(\*) INTO count
   FROM student
   WHERE major = dCode;

   RETURN count;
END //

DELIMITER ;

SHOW CREATE FUNCTION n\_major;

SELECT n\_major('CSCI');
SELECT n\_major('ITEC');

Note:

* A function returns a value and does not accomplish its goal through side effects.
* Note the argument passed into the function.
* Formal arguments of a function use the IN parameter passing mode.
* Note the variable declaration and assignment.
* The scope of the variable is within the function.
* 'READS SQL DATA' describes the characteristic of the subroutine: it reads but not writes SQL data.

***Example:***

The following example functions are included in the script of creating toyu. Function calls are added.

-- get the full name of a student.
DELIMITER //
CREATE OR REPLACE FUNCTION GetStudentFullName(
    student\_id INT
)
RETURNS VARCHAR(61)
DETERMINISTIC
BEGIN
    DECLARE full\_name VARCHAR(61);
    SELECT CONCAT(fname, ' ', lname) INTO full\_name
    FROM Student
    WHERE stuId = student\_id;
    RETURN full\_name;
END //
DELIMITER ;

SELECT GetStudentFullName(100000);
SELECT GetStudentFullName(100001);

-- get the full name of a department code
DELIMITER //
CREATE OR REPLACE FUNCTION GetDepartmentName(
    dept\_code CHAR(4)
)
RETURNS VARCHAR(30)
DETERMINISTIC
BEGIN
    DECLARE dept\_name VARCHAR(30);
    SELECT d.deptName INTO dept\_name
    FROM Department d
    WHERE d.deptCode = dept\_code;
    RETURN dept\_name;
END //
DELIMITER ;

SELECT GetDepartmentName('CSCI');
SELECT GetDepartmentName('ITEC');

-- Compute and return the GPA of a student
DELIMITER //
CREATE OR REPLACE FUNCTION GetStudentGPA(
    student\_id INT
)
RETURNS DECIMAL(3,2)
DETERMINISTIC
BEGIN
    DECLARE gpa DECIMAL(3,2);
    SELECT ROUND(SUM(g.gradePoint \* co.credits) / SUM(co.credits), 2) INTO gpa
    FROM Enroll e
    JOIN Class c ON e.classId = c.classId
    JOIN Course co ON c.courseId = co.courseId
    JOIN Grade g ON e.grade = g.grade
    WHERE e.stuId = student\_id;
    RETURN gpa;
END //
DELIMITER ;

SELECT GetStudentGPA(100000);
SELECT GetStudentGPA(100001);
SELECT GetStudentGPA(100002);

-- Return the top n students in a major department with GPA.
DELIMITER //
CREATE OR REPLACE FUNCTION GetTopStudentsInDepartment(
    dept\_code CHAR(4),
    top\_count INT
)
RETURNS VARCHAR(1000)
DETERMINISTIC
BEGIN
    DECLARE student\_list VARCHAR(1000);
    SET student\_list = '';

    WITH temp AS(
        SELECT s.stuId, CONCAT(s.fname, ' ', s.lname) AS student,
            ROUND(SUM(g.gradePoint \* co.credits) / SUM(co.credits), 2) AS gpa
        FROM Student s
        JOIN Department d ON s.major = d.deptCode
        JOIN Enroll e ON s.stuId = e.stuId
        JOIN Class c ON e.classId = c.classId
        JOIN Course co ON c.courseId = co.courseId
        JOIN Grade g ON e.grade = g.grade
        WHERE d.deptCode = dept\_code
        GROUP BY s.stuId, student
        ORDER BY gpa DESC
        LIMIT top\_count)
    SELECT GROUP\_CONCAT(CONCAT(temp.student, '(', temp.stuId, '):', temp.gpa) SEPARATOR ', ')
        INTO student\_list
    FROM temp
    LIMIT 1;

    RETURN student\_list;
END //
DELIMITER ;

SELECT GetTopStudentsInDepartment('CSCI', 2);
SELECT GetTopStudentsInDepartment('CSCI', 3);

**3.3 Cursors: iterator through resultset**

* In stored subroutines in MySQL, various high level language constructs are supported, such as variable declarations, conditional statements, control statements, etc.
* In particular, cursors are supported for allowing iteration through the result sets.

***Example:***

Execute the following code and ensure that you understand the result. The example is artifically constructed as there are better ways.

-- using cursor.
DELIMITER //

CREATE FUNCTION major\_students(dept VARCHAR(4))
RETURNS VARCHAR(1000)
READS SQL DATA
BEGIN
DECLARE result VARCHAR(1000) DEFAULT '';
DECLARE name VARCHAR(41) DEFAULT '';
DECLARE done INT DEFAULT FALSE;

DECLARE cursor\_1 CURSOR FOR
SELECT DISTINCT CONCAT(fName, ' ', lName) AS name
FROM student
WHERE major = dept;

DECLARE continue handler FOR NOT FOUND SET done = TRUE;

OPEN cursor\_1;
SET result = '';

compute\_loop: LOOP
-- This is needed as there are other SQL statements that may set done to true.
SET done = false;
FETCH cursor\_1 INTO name;
IF done THEN
LEAVE compute\_loop;
END IF;

IF (result <> '') THEN
SET result = CONCAT(result, ', ');
END IF;

SET result = CONCAT(result, name);
END LOOP;

CLOSE cursor\_1;

RETURN result;
END //

DELIMITER ;

SELECT major\_students('CSCI');
SELECT major\_students('CINF');

DROP FUNCTION major\_students;

* Note that the example is only used to demonstrate cursors in a stored function. A much better solution is to simply use GROUP\_CONCAT, such as:

SELECT GROUP\_CONCAT(CONCAT(fName, ' ', lName) SEPARATOR ', ') AS majors
FROM student
WHERE major = 'CSCI';

**4. Triggers**

* Triggers allow *event-driven* programming.
* A trigger is activated when certain events occur. Unlike stored routines, triggers are not explicitly called.
* Four questions to ask for an event model:
	1. What are the events?
	2. What information can be obtained from the events?
	3. What actions can be performed to handle events?
	4. How do the events propagate?

**4.1 The event model for triggers**

* Events:
	1. Inserting a row
	2. Updating a row
	3. Deleting a row
* Actions can be executed:
	1. Before the event
	2. After the event
* Information can be obtained through two keywords:
	1. old:
		+ the old value of the row before the event.
		+ for update and delete row events
	2. new:
		+ the new value of the row after the event.
		+ for update and insert events
* Triggers do not propagate.

Advantages of triggers

1. The event driven model suits certain kinds of tasks better.
2. Can ease the implementation of consistency check.
3. Can ease the implementation of business logic and integrity check.

Disadvantages:

1. Relatively invisible, and possibly overlooked by developers.
2. Relatively difficult to debug.
3. Potential performance issues.
4. Potential complicated interactions.

***Example:***

An example of using trigger is for auditing, e.g. <https://vladmihalcea.com/mysql-audit-logging-triggers/>

Execute the following code for auditing and ensure that you understand the result.

-- trigger

CREATE TABLE courseUpdate(
   cuId        INT NOT NULL AUTO\_INCREMENT,
   `type`      CHAR(1),
    courseId    INT NOT NULL,
    rubric      CHAR(4) NOT NULL,
    number      CHAR(4) NOT NULL,
    title       VARCHAR(80) NOT NULL,
    credits     TINYINT NULL,
    ts          TIMESTAMP DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP,
    CONSTRAINT CourseUpdate\_courseId\_pk PRIMARY KEY (cuId),
    CONSTRAINT CourseUpdate\_deptCode\_fk FOREIGN KEY (rubric)
        REFERENCES Department(deptCode));

-- trigger example.
DELIMITER $$

CREATE TRIGGER update\_Course AFTER UPDATE ON Course FOR EACH ROW
BEGIN
   -- code should be more sophisticated.
   INSERT INTO courseUpdate(`type`, courseId, rubric, number, title, credits)
   VALUES('U', new.courseId, new.rubric, new.number, new.title, new.credits);
END $$

CREATE TRIGGER insert\_Course AFTER INSERT ON Course FOR EACH ROW
BEGIN
   -- code should be more sophisticated.
   INSERT INTO courseUpdate(`type`, courseId, rubric, number, title, credits)
   VALUES('I', new.courseId, new.rubric, new.number, new.title, new.credits);
END $$

DELIMITER ;

SELECT \* FROM Course;

UPDATE Course
SET number = 2315
WHERE courseId = 2000;

INSERT INTO Course(courseId, rubric, number, title, credits) VALUES
    (4000,'CSCI',3532,'Advanced Data Structures',3);

UPDATE Course
SET number = 3341
WHERE courseId = 4000;

SELECT \* FROM course;
SELECT \* FROM courseUpdate;

-- Clean up.
DROP TRIGGER update\_Course;
DROP TRIGGER insert\_Course;

DROP TABLE courseUpdate;

DELETE FROM Course
WHERE CourseId = 4000;

UPDATE Course
SET number = 3333
WHERE courseId = 2000;

SELECT \* FROM Course;

**5. System Catalog**

* Named collections of *meta-data* in SQL DBMS.
* Important for DB administrators and developers to know.
* Usually stored in relations.
* Common terms with similar meaning: data dictionary, catalog, etc.
* Three major databases that are included with MySQL: mysql, information\_schema and performance\_schema.

***Example:***

mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information\_schema |
| mysql              |
| performance\_schema |
| swim              |
| toytu              |
| world              |
| yue\_exp            |
+--------------------+
7 rows in set (0.00 sec)
>

**5.1 mysql database**

* MySQL database stores information about the system. It is the system schema.
* See: <https://dev.mysql.com/doc/refman/8.1/en/system-schema.html>.
* However, many of these system tables cannot be accessed directly. They should be access through other means (such as information\_schema, see below)

***Example:***

SELECT DISTINCT u.user, u.password
FROM mysql.user AS u
WHERE u.host = 'localhost';

**5.2 information\_schema database**

* Store information about all databases in MySQL.
* Contents are system*views* using the MEMORY storage engine and they cannot be updated.
* Triggers cannot be defined on tables in information\_schema.

***Example:***

mysql> SELECT table\_name, table\_type, row\_format, table\_rows, avg\_row\_length
    -> FROM information\_schema.tables
    -> WHERE table\_schema = 'information\_schema'
    -> ORDER BY table\_name DESC;
+---------------------------------------+-------------+------------+------------+----------------+
| table\_name                            | table\_type  | row\_format | table\_rows | avg\_row\_length |
+---------------------------------------+-------------+------------+------------+----------------+
| VIEWS                                 | SYSTEM VIEW | Dynamic    |       NULL |              0 |
| USER\_PRIVILEGES                       | SYSTEM VIEW | Fixed      |       NULL |           1986 |
| TRIGGERS                              | SYSTEM VIEW | Dynamic    |       NULL |              0 |
| TABLE\_PRIVILEGES                      | SYSTEM VIEW | Fixed      |       NULL |           2372 |
| TABLE\_CONSTRAINTS                     | SYSTEM VIEW | Fixed      |       NULL |           2504 |
| TABLESPACES                           | SYSTEM VIEW | Fixed      |       NULL |           6951 |
| TABLES                                | SYSTEM VIEW | Fixed      |       NULL |           9450 |
| STATISTICS                            | SYSTEM VIEW | Fixed      |       NULL |           5753 |
| SESSION\_VARIABLES                     | SYSTEM VIEW | Fixed      |       NULL |           3268 |
| SESSION\_STATUS                        | SYSTEM VIEW | Fixed      |       NULL |           3268 |
| SCHEMA\_PRIVILEGES                     | SYSTEM VIEW | Fixed      |       NULL |           2179 |
| SCHEMATA                              | SYSTEM VIEW | Fixed      |       NULL |           3464 |
| ROUTINES                              | SYSTEM VIEW | Dynamic    |       NULL |              0 |
| REFERENTIAL\_CONSTRAINTS               | SYSTEM VIEW | Fixed      |       NULL |           4814 |
| PROFILING                             | SYSTEM VIEW | Fixed      |       NULL |            308 |
| PROCESSLIST                           | SYSTEM VIEW | Dynamic    |       NULL |              0 |
| PLUGINS                               | SYSTEM VIEW | Dynamic    |       NULL |              0 |
| PARTITIONS                            | SYSTEM VIEW | Dynamic    |       NULL |              0 |
| PARAMETERS                            | SYSTEM VIEW | Dynamic    |       NULL |              0 |
| KEY\_COLUMN\_USAGE                      | SYSTEM VIEW | Fixed      |       NULL |           4637 |
| INNODB\_TRX                            | SYSTEM VIEW | Fixed      |       NULL |           4534 |
| INNODB\_LOCK\_WAITS                     | SYSTEM VIEW | Fixed      |       NULL |            599 |
| INNODB\_LOCKS                          | SYSTEM VIEW | Fixed      |       NULL |          31244 |
| INNODB\_CMP\_RESET                      | SYSTEM VIEW | Fixed      |       NULL |             25 |
| INNODB\_CMPMEM\_RESET                   | SYSTEM VIEW | Fixed      |       NULL |             29 |
| INNODB\_CMPMEM                         | SYSTEM VIEW | Fixed      |       NULL |             29 |
| INNODB\_CMP                            | SYSTEM VIEW | Fixed      |       NULL |             25 |
| GLOBAL\_VARIABLES                      | SYSTEM VIEW | Fixed      |       NULL |           3268 |
| GLOBAL\_STATUS                         | SYSTEM VIEW | Fixed      |       NULL |           3268 |
| FILES                                 | SYSTEM VIEW | Fixed      |       NULL |           2677 |
| EVENTS                                | SYSTEM VIEW | Dynamic    |       NULL |              0 |
| ENGINES                               | SYSTEM VIEW | Fixed      |       NULL |            490 |
| COLUMN\_PRIVILEGES                     | SYSTEM VIEW | Fixed      |       NULL |           2565 |
| COLUMNS                               | SYSTEM VIEW | Dynamic    |       NULL |              0 |
| COLLATION\_CHARACTER\_SET\_APPLICABILITY | SYSTEM VIEW | Fixed      |       NULL |            195 |
| COLLATIONS                            | SYSTEM VIEW | Fixed      |       NULL |            231 |
| CHARACTER\_SETS                        | SYSTEM VIEW | Fixed      |       NULL |            384 |
+---------------------------------------+-------------+------------+------------+----------------+
37 rows in set (0.08 sec)

* It is more flexible than the SHOW command.

***Example:***

Execute the following code and ensure that you understand the results.

-- System Catalog
-- Getting selected columns from information\_schema
SELECT table\_name, table\_type, row\_format, table\_rows, avg\_row\_length
FROM information\_schema.tables
WHERE table\_schema = 'information\_schema'
ORDER BY table\_name DESC;

-- databases and tables
SELECT t.TABLE\_SCHEMA AS `schema`, COUNT(t.TABLE\_NAME) AS num\_tables
FROM information\_schema.tables AS t
GROUP BY `schema`
ORDER BY num\_tables DESC;

SELECT t.TABLE\_SCHEMA AS `schema`, t.ENGINE, COUNT(t.TABLE\_NAME) AS num\_tables
FROM information\_schema.tables t
GROUP BY `schema`, t.ENGINE
ORDER BY `schema`, num\_tables DESC;

***Example:***

DROP SCHEMA IF EXISTS dbtool;
CREATE SCHEMA dbtool;
USE dbtool;

-- Return the number of columns of a table in a schema (database) in the output parameter column\_count
DELIMITER //
CREATE OR REPLACE PROCEDURE count\_columns(
    IN schema\_name VARCHAR(64),
    IN table\_name VARCHAR(64),
    OUT column\_count INT
)
BEGIN
    SET @\_\_cc\_query = CONCAT('SELECT COUNT(\*) INTO @\_\_cc\_column\_count FROM INFORMATION\_SCHEMA.COLUMNS WHERE TABLE\_SCHEMA = ? AND TABLE\_NAME = ?');
    PREPARE stmt FROM @\_\_cc\_query;
    SET @\_\_cc\_schema\_name = schema\_name;
    SET @\_\_cc\_table\_name = table\_name;

    EXECUTE stmt USING @\_\_cc\_schema\_name, @\_\_cc\_table\_name;
    SET column\_count = @\_\_cc\_column\_count;
    DEALLOCATE PREPARE stmt;
END //
DELIMITER ;

CALL count\_columns('toyu', 'student', @column\_count);
SELECT @column\_count;
CALL count\_columns('swim', 'swimmer', @column\_count);
SELECT @column\_count;

**5.3 performance\_schema database**

* contain performance related server data.