**3/10/2025**

Self annotation/notes

**Introduction to SQL and MySQL**

by K. Yue

**1. Introduction**

* SQL (Structured Query Language): defacto standard for relational databases.
* SQL-like languages are also used in non-relational DBMS.
* Contains core specifications and extensions. Latest SQL standard: 2016.
* Not using a *pure* relational model: e.g.
  1. Use the terms row, column, and table instead of tuple, attribute, and relation.
  2. The results may not be a set.
* Mostly based on Tuple Relational Calculus (TRC) and a little on Relational Algebra (RA).
* SQL is mostly declarative.
* DBMS vendor-specific extensions are common.
* SQL Contains:
  1. Data Definition Language (DDL): define the relation schema (structure)
  2. Data Manipulation Language (DML): manipulate data; CRUD:
     1. Create: Insert
     2. Read
     3. Update
     4. Delete
  3. Data Administration Language: for DB administration such as user and security management.

**2. MySQL**

Use [toyu](https://dcm.uhcl.edu/yue/courses/joinDB/Spring2025/notes/toyu/toyu.html), a drastically simplified university, as examples.

**2.1 DDL:**

* Make sure that you are familiar with the core SQL Data Definition Language (DDL) commands. Refer to, for example: <http://www.w3schools.com/sql/default.asp>.
* MariaDB DDL: <https://mariadb.com/kb/en/sql-statements/>
* MySQL DDL manual: <https://dev.mysql.com/doc/refman/8.1/en/sql-statements.html>.
* Basic DDL: some examples
  1. CREATE TABLE
  2. CREATE DATABASE: a database contains a collection of related tables for an application.
  3. CREATE VIEW: a view is a virtual table for users to access a subset of a database.
  4. CREATE INDEX: an index is a data structure to enhance access performance of specific queries.
  5. CREATE PROCEDURE
  6. CREATE FUNCTION
  7. CREATE TRIGGER: a trigger is event-driven procedural code activated by events.
  8. ALTER (No ALTER TRIGGER and ALTER INDEX)
  9. DROP

BNF:

* Production Rules: e.g. <postal-address> ::= <name-part> <street-address> <zip-part>
* Terminals: e.g.. "Sr.", "Jr."; in MYSQL: CREATE, TABLE, IF, NOT,…
* Non-terminals: <postal-address>, <name-part>, <street-address>, <zip-part>; In MYSQL: *create\_definition*, *table\_options*, *partition\_options*.
* Special symbol: |: or; []: optional; X,… means one or more X separated by a comma.
* Root non-terminals -> expand by using rules
* If these expansion cover the entire statement, it is syntactically correct.
* Note that in MySQL, DATABASE and SCHEMA having the same meaning. Hierarchy:
  1. MySQL: Database = Schema: contains a collection of tables.
  2. Postgres:
     1. A database contains a collection of schema.
     2. A schema contains a collection of tables.
* Constraints: to implement certain constraints in your data model.
  1. NOT NULL: attributes cannot have an null value.
  2. UNIQUE: KEY; the set of attributes must be unique for each row:
  3. PRIMARY KEY: unique, not null, and used for the physical structure of the relation.
  4. FOREIGN KEY
  5. CHECK: for a Boolean condition on the columns.
  6. DEFAULT: define a default value.
* Some other options:
  1. AUTO INCREMENT: automatic increment an integer if a value is not specified. Used for id.

***Example:***

Experimenting with the CREATE TABLE command. Execute the following code and ensure that you understand the result. For example,

* A temporary table is not persistent. It is created for a SQL client session. Its scope is the client session.

DROP SCHEMA IF EXISTS tinker;  
CREATE SCHEMA tinker;  
USE tinker;  
  
CREATE TABLE s2  
SELECT \* FROM toyu.student;  
  
SELECT \*  
FROM s2;  
  
CREATE TEMPORARY TABLE s3  
SELECT \* FROM toyu.student;  
  
SELECT \*  
FROM s3;  
  
CREATE TABLE s4 LIKE toyu.student;  
  
SELECT \*  
FROM s4;  
  
INSERT INTO s4  
SELECT \* FROM toyu.student;  
  
SELECT \*  
FROM s4;  
  
SHOW TABLES;  
  
-- Note that keys and constraints of student are missing in s2 and S3.  
DESC student;  
DESC s2;  
DESC s3;  
DESC s4;  
  
DROP TABLE s2;  
DROP TABLE s3;  
DROP TABLE s4;  
  
SHOW TABLES;  
  
DROP SCHEMA IF EXISTS tinker;  
  
Column names may include special characters. For example, you cannot use the name 'first name' directly as column name, as spaces are interpreted as separator. You will need special syntax. For example:

1. In MySQL, use back-quote: `first name`
2. In MS SQL Server, use []: [first name]

* For each column, there is a data type and optional specifiers (such as NULL, NOT NULL, default values, etc.)
* Additional constraints and indexes can be defined.
* In general, some important considerations in creating tables:
  1. What are the columns?
  2. What are the data types of the columns?
     + The right domain: be restrictive.
     + Performance consideration.
  3. Nullability of columns
  4. Primary key
  5. Candidate keys
  6. Foreign keys: can they be enforced by the selected storage engine?
  7. Indexes: performance tuning.
  8. Additional constraints: check whether they are enforced by the storage engine.

***Example:***

* MySQL only supports foreign key constraint in the InnoDB database engine.
* Older versions of MySQL ignore the 'check' clause.

**2.2 Data types**

* Be mindful of data types.
* Data types in MySQL are rich: <https://dev.mysql.com/doc/refman/8.1/en/data-types.html>
* Beside simple data types, other noticeable data types:
  1. JSON: JavaScript Object Notation
  2. Spatial: support OpenGIS Geometry Model
  3. BLOB: Binary large object
  4. TEXT: long character strings (VARCHAR is limited to 255, extensible to 64K).

**3. DML**

* Basically declarative.

**3.1 Writing to the DB**

* Basic *update* commands (write):
  1. INSERT
  2. UPDATE
  3. DELETE

INSERT INTO <<table>> [<<columns>>]  
VALUES <<expression>>

* If column names are missing, the proper column order during table creation will be used.
* Column names using default values or auto-increment values should not be included in the INSERT statement if they are used.
* NULL and DEFAULT can be used as values in INSERT.
* One may also insert values from a select statement. E.g.

INSERT INTO <<table>> [<<columns>>]  
<<select statement>>

* The DELETE statement includes a condition for selecting the rows for deletion.

DELETE FROM <<table>>  
WHERE <<condition>>

* The update statement is used to update rows and may have an update condition to identify the rows to be updated.

UPDATE <<table>>  
SET <<update assignments>>  
[WHERE <<update condition>>]

***Example:***

UPDATE Student  
SET major = 'ITEC'  
WHERE StuId = 100000;

* The update and delete statements can be used to affect multiple rows so be very careful.

***Example:***

-- All students will be majoring in CSCI  
UPDATE Student  
SET major = 'CSCI';

* Once changed, the effect is permanent. There is no 'undo' command.

***Example:***

Note the order of the insertions in createtoyu.sql below.

INSERT INTO Grade(grade, gradePoint) VALUES  
    ('A',4),('A-',3.6667),('B+',3.3333),('B',3),('B-',2.6667),  
    ('C+',2.3333),('C',2),('C-',1.6667),  
    ('D+',1.3333),('D',1),('D-',0.6667),('F',0),  
   ('P', NULL), ('IP', NULL), ('WX', NULL);  
     
INSERT INTO School(schoolCode, schoolName) VALUES  
    ('BUS','Business'),  
   ('EDU','Education'),  
    ('HSH','Human Sciences and Humanities'),  
    ('CSE','Science and Engineering');  
  
INSERT INTO Department(deptCode, deptName, schoolCode, numStaff) VALUES  
    ('ACCT','Accounting','BUS',10),  
    ('ARTS','Arts','HSH',5),  
    ('CINF','Computer Information Systems','CSE',5),  
    ('CSCI','Computer Science','CSE',12),  
    ('ENGL','English','HSH',12),  
    ('ITEC','Information Technology','CSE',4),  
    ('MATH','Mathematics','CSE',7);  
  
INSERT INTO Faculty(facId, fname, lname, deptCode, `rank`) VALUES  
    (1011,'Paul','Smith','CSCI','Professor'),  
    (1012,'Mary','Tran','CSCI','Associate Professor'),  
    (1013,'David','Love','CSCI',NULL),  
    (1014,'Sharon','Mannes','CSCI','Assistant Professor'),  
    (1015,'Daniel','Kim','CINF','Professor'),  
    (1016,'Andrew','Byre','CINF','Associate Professor'),  
    (1017,'Deborah','Gump','ITEC','Professor'),  
    (1018,'Art','Allister','ARTS','Assistant Professor'),  
    (1019,'Benjamin','Yu','ITEC','Lecturer'),  
    (1020,'Katrina','Bajaj','ENGL','Lecturer'),  
    (1021,'Jorginlo','Neymar','ACCT','Assistant Professor');  
  
INSERT INTO Course(courseId, rubric, number, title, credits) VALUES  
    (2000,'CSCI',3333,'Data Structures',3),  
    (2001,'CSCI',4333,'Design of Database Systems',3),  
    (2002,'CSCI',5333,'DBMS',3),  
    (2020,'CINF',3321,'Introduction to Information Systems',3),  
    (2021,'CINF',4320,'Web Application Development',3),  
    (2040,'ITEC',3335,'Database Development',3),  
    (2041,'ITEC',3312,'Introduction to Scripting',3),  
    (2060,'ENGL',1410,'English I',4),  
    (2061,'ENGL',1311,'English II',3),  
    (2080,'ARTS',3311,'Hindu Arts',3),  
    (2090,'ACCT',3333,'Managerial Accounting',3);  
  
INSERT INTO Class(classId, courseId, semester, year, facId, room) VALUES  
    (10000,2000,'Fall',2019,1011,'D241'),  
    (10001,2001,'Fall',2019,1011,'D242'),  
    (10002,2002,'Fall',2019,1012,'D136'),  
    (10003,2020,'Fall',2019,1014,'D241'),  
    (10004,2021,'Fall',2019,1014,'D241'),  
    (10005,2040,'Fall',2019,1015,'D237'),  
    (10006,2041,'Fall',2019,1019,'D217'),  
    (10007,2060,'Fall',2019,1020,'B101'),  
    (10008,2080,'Fall',2019,1018,'D241'),  
    (11000,2000,'Spring',2020,1011,'D241'),  
    (11001,2001,'Spring',2020,1012,'D242'),  
    (11002,2002,'Spring',2020,1013,'D136'),  
    (11003,2020,'Spring',2020,1016,'D217'),  
    (11004,2061,'Spring',2020,1018,'B101');  
  
INSERT INTO Student(stuId, fname, lname, major, minor, ach, advisor) VALUES  
    (100000,'Tony','Hawk','CSCI','CINF',40,1011),  
    (100001,'Mary','Hawk','CSCI','CINF',35,1011),  
    (100002,'David','Hawk','CSCI','ITEC',66,1012),  
    (100003,'Catherine','Lim','ITEC','CINF',20,NULL),  
    (100004,'Larry','Johnson','ITEC',NULL,66,1017),  
    (100005,'Linda','Johnson','CINF','ENGL',13,1015),  
    (100006,'Lillian','Johnson','CINF','ITEC',18,1016),  
    (100007,'Ben','Zico',NULL,NULL,16,NULL),  
    (100008,'Bill','Ching','ARTS',NULL,90,NULL),  
    (100009,'Linda','King','ARTS','CSCI',125,1018),  
   (100111,'Cathy','Johanson',NULL,NULL,0,1018);  
     
INSERT INTO Enroll(stuId, classId, grade, n\_alerts) VALUES  
    (100000,10000,'A',0),  
    (100001,10000,NULL,NULL),  
    (100002,10000,'B-',3),  
    (100000,10001,'A',2),  
    (100001,10001,'A-',0),  
    (100000,10002,'B+',1),  
    (100002,10002,'B+',2),  
    (100000,10003,'C',0),  
    (100002,10003,'D',4),  
    (100004,10003,'A',0),  
    (100005,10003,NULL,NULL),  
    (100000,10004,'A-',1),  
    (100004,10004,'B+',NULL),  
    (100005,10004,'A-',0),  
    (100006,10004,'C+',NULL),  
    (100005,10005,'A-',0),  
    (100006,10005,'A',NULL),  
    (100005,10006,'B+',NULL),  
    (100007,10007,'F',4),  
    (100008,10007,'C-',0),  
    (100007,10008,'A-',0),  
    (100000,11001,'D',4);

Note the explicit use of NULL, which is a keyword in SQL.

***Example:***

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

INSERT INTO student VALUES  
   (100010,'Bun','Yue',null,null,50,null),  
   (100011,'Paul','Harris','CSCI','ITEC',23,1015);  
    
SELECT \* FROM student;  
    
INSERT INTO student VALUES  
   (100010,'Bun','Yue',null,null,50,null),  
   (100011,'Paul','Harris','CSCI','ITEC',23,1015);  
    
INSERT INTO student VALUES    
   (100020,'Bunno','Yue','GEOG',null,50,null);  
INSERT INTO student VALUES    
   (100021,'Bunna','Yue',null,'GEOG',50,null);  
INSERT INTO student VALUES    
   (100022,'Bunno','Yue',null,null,50,8888);  
  
    
-- Remove the two new rows.  
DELETE FROM Student  
WHERE stuId = 100010 OR stuId = 100011;  
  
SELECT \* FROM student;

**3.2 Querying with the SELECT Statement**

* SELECT is the basic data retrieval statement in SQL
* Not to be confused with the select statement in Relational Algebra (RA).
* Basic format, with*conceptual* steps.

SELECT DISTINCT <<result\_columns>> -- [3] construct result columns  
FROM <<source\_tables>> -- [1] conceptually join sources to form a large table  
WHERE <<conditions\_for\_inclusion>> -- [2] Filter rows from [1]

1. <<source\_tables>>: the source tables to gather the result data
2. <<conditions\_for\_inclusion>>: the conditions to be satisfied for results to be included and possibly the conditions how the tables should be joined together.
3. <<result\_columns>>: the result columns or expressions desired to be displayed.

* Built-in functions and operators: <https://dev.mysql.com/doc/refman/8.1/en/built-in-function-reference.html>
* Some examples of common functions:
  1. BETWEEN lower\_range AND upper\_range
  2. IN: membership test for a set/table (binary operation)
  3. EXISTS: not an empty set (unary operation)
  4. IF: a ternary operation
  5. LIKE: inexact string matching.
     + wild cards:
       - % match any and all following characters.
       - \_: match any one character.

***Example:***

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

-- operators:  
-- student with credits in a range.  
SELECT DISTINCT \*  
FROM Student  
WHERE credits BETWEEN 30 AND 70;  
  
-- student in selected majors  
SELECT DISTINCT \*  
FROM Student  
WHERE major IN ('CSCI', 'CINF', 'ITEC');  
  
-- student enrolled in some classes.  
SELECT DISTINCT \*  
FROM Student AS s  
WHERE EXISTS  
(SELECT \*  -- a subquery  
FROM Enroll AS e  
WHERE e.stuId = s.stuId);  
-- or  
SELECT DISTINCT s.\*  
FROM Student AS s INNER JOIN Enroll AS e USING (stuId);  
  
-- students not enrolled in any class.  
SELECT DISTINCT \*  
FROM student AS s  
WHERE s.stuId NOT IN (SELECT DISTINCT e.stuID FROM enroll AS e);  
  
-- students wiht a 'k' in their last name.  
SELECT DISTINCT s.\*  
FROM student AS s  
WHERE s.lname LIKE '%k%';  
  
-- case sensitive version.  
SELECT DISTINCT s.\*  
FROM student AS s  
WHERE s.lname LIKE BINARY '%k%';

-- case sensitive version: a more complicated take.  
-- The mysql client sends the query using cp850.  
-- The default character set of MySQL server is utf8mb4.  
-- It is thus necessary to set the @@character\_set\_connection  
-- in order to use collate if MySQL client is used.  
-- If HeidiSQL is used, it is not necessary.  
SET @@character\_set\_connection=utf8mb4;

SELECT DISTINCT s.\*  
FROM student AS s  
WHERE s.lname LIKE '%k%' COLLATE utf8mb4\_bin;  
  
-- LIKE compares the whole string.  
SELECT DISTINCT s.\*  
FROM student AS s  
WHERE s.lname LIKE 'ng';  
  
-- student with last name of four characters, with ng the last two.  
SELECT DISTINCT s.\*  
FROM student AS s  
WHERE s.lname LIKE '\_\_ng';

**3.3 Joins**

* When multiple tables are needed for a query, it is common that foreign keys are used to connect the tables.
* It is thus necessary to ensure that the equality of the foreign key with the referenced key of the parent table.
* A popular style is shown in the example below.

***Example:*** one popular SQL style

SELECT DISTINCT s.fname, s.lname, c.classId, e.grade  
FROM student AS s, enroll AS e, class AS c  
WHERE s.stuId = e.stuId -- Join condition  
AND e.classId = c.classId -- Join condition  
AND c.semester = 'Fall' -- problem condition  
AND c.year = 2019; -- problem condition

Result:

mysql> SELECT DISTINCT s.fname, s.lname, c.classId, e.grade  
    -> FROM student s, enroll e, class c  
    -> WHERE s.stuId = e.stuId   -- Join condition  
    -> AND e.classId = c.classId -- Join condition  
    -> AND c.semester = 'Fall'   -- problem condition  
    -> AND c.year = 2019;        -- problem condition  
+---------+---------+---------+-------+  
| fname   | lname   | classId | grade |  
+---------+---------+---------+-------+  
| Tony    | Hawk    |   10000 | A     |  
| Mary    | Hawk    |   10000 | NULL  |  
| David   | Hawk    |   10000 | B-    |  
| Tony    | Hawk    |   10001 | A     |  
| Mary    | Hawk    |   10001 | A-    |  
| Tony    | Hawk    |   10002 | B+    |  
| David   | Hawk    |   10002 | B+    |  
| Tony    | Hawk    |   10003 | C     |  
| David   | Hawk    |   10003 | D     |  
| Larry   | Johnson |   10003 | A     |  
| Linda   | Johnson |   10003 | NULL  |  
| Tony    | Hawk    |   10004 | A-    |  
| Larry   | Johnson |   10004 | B+    |  
| Linda   | Johnson |   10004 | A-    |  
| Lillian | Johnson |   10004 | C+    |  
| Linda   | Johnson |   10005 | A-    |  
| Lillian | Johnson |   10005 | A     |  
| Linda   | Johnson |   10006 | B+    |  
| Ben     | Zico    |   10007 | F     |  
| Bill    | Ching   |   10007 | C-    |  
| Ben     | Zico    |   10008 | A-    |  
+---------+---------+---------+-------+  
21 rows in set (0.00 sec)

**3.3.1 Inner Join**

* In the SELECT statement, the FROM clause allows the results of JOIN statements in the table references.
* Using the JOIN operations in the FROM clause is the preferred technique:
  1. Potentially faster performance: better optimization by DB engines, especially when using indexes.
  2. Better style: separation of join conditions and query semantic conditions.
  3. Easier changes between different joins.
* There are many kind of joins, as discussed below.
* You may use the Explain statement in MySQL to find out the execution plan.

***Example:***

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

SELECT DISTINCT s.fname, s.lname, c.classId, e.grade  
FROM student AS s, enroll AS e, class AS c  
WHERE s.stuId = e.stuId -- Join condition  
AND e.classId = c.classId -- Join condition  
AND c.semester = 'Fall' -- problem condition  
AND c.year = 2019; -- problem condition  
  
SELECT DISTINCT s.fname, s.lname, c.classId, e.grade  
FROM student AS s INNER JOIN enroll e ON (s.stuId = e.stuId) -- Join condition  
    INNER JOIN class AS c ON (e.classId = c.classId) -- Join condition  
WHERE c.semester = 'Fall' -- Problem condition  
AND c.year = 2019; -- Problem condition  
  
-- alternative: using the USING clause.  
SELECT DISTINCT s.fname, s.lname, c.classId, e.grade  
FROM student AS s INNER JOIN enroll e USING (stuId) -- Join condition  
    INNER JOIN class AS c USING (classId) -- Join condition  
WHERE c.semester = 'Fall' -- Problem condition  
AND c.year = 2019; -- Problem condition  
  
-- the ON clause is more general and can be more effective.  
SELECT DISTINCT s.fname, s.lname, c.classId, e.grade  
FROM student AS s INNER JOIN enroll e ON (s.stuId = e.stuId) -- Join condition  
    INNER JOIN class AS c  
   ON (e.classId = c.classId -- Join condition  
      AND c.semester = 'Fall' -- Problem condition  
      AND c.year = 2019); -- Problem condition

[FROM *table\_references*

[PARTITION *partition\_list*]]

* ***table\_references*** indicates the table or tables from which to retrieve rows. Its syntax is described in [Section 15.2.13.2, “JOIN Clause”](https://dev.mysql.com/doc/refman/8.4/en/join.html).

|  |
| --- |
| *table\_references:*  *escaped\_table\_reference* [, *escaped\_table\_reference*] ...  *escaped\_table\_reference*: {  *table\_reference*  | { OJ *table\_reference* }  }  *table\_reference*: {  *table\_factor*  | *joined\_table*  }  *table\_factor*: {  *tbl\_name* [PARTITION (*partition\_names*)]  [[AS] *alias*] [*index\_hint\_list*]  | [LATERAL] *table\_subquery* [AS] *alias* [(*col\_list*)]: a subquery can be used as a source table in the from clause.  | ( *table\_references* )  }  *joined\_table*: {  *table\_reference* {[INNER | CROSS] JOIN | STRAIGHT\_JOIN} *table\_factor* [*join\_specification*]  | *table\_reference* {LEFT|RIGHT} [OUTER] JOIN *table\_reference* *join\_specification*  | *table\_reference* NATURAL [INNER | {LEFT|RIGHT} [OUTER]] JOIN *table\_factor*  }  *join\_specification*: {  ON *search\_condition*  | USING (*join\_column\_list*)  }  *join\_column\_list*:  *column\_name*[, *column\_name*] ... |

**3.3.2 Left and Right Join**

* Left joins are the most popular joins besides (inner) joins.
* R1 LEFT JOIN R2: same as INNER JOIN, except that for a tuple t1 in R1 without a matching tuple in R2, t1 will be kept in the result with attributes from R2 being null.
  + All rows in the left table will be in the result at least once.
* A right join is the mirror image of a left join.

***Example***

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

-- List the names of the students with their minors (in full name).  
-- Student with no department not listed.  
SELECT DISTINCT CONCAT(s.fname, ' ', s.lname) AS student,  
    d.deptName AS `minor department`  
FROM student AS s INNER JOIN department AS d ON (s.minor = d.deptCode);  
  
-- List the names of the students with their minors (in full name).  
SELECT DISTINCT CONCAT(s.fname, ' ', s.lname) AS student,  
    d.deptName AS `minor department`  
FROM student AS s LEFT JOIN department AS d ON (s.minor = d.deptCode);  
  
-- List the names of the students with their minors (in full name).  
-- more readable form.  
SELECT DISTINCT CONCAT(s.fname, ' ', s.lname) AS student,  
    IFNULL (d.deptName, 'N/A') AS `minor department`  
FROM student s LEFT JOIN department d ON (s.minor = d.deptCode);

* Joins are procedural. Join orders can be important. Use parenthesis to enforce the desired order.

***Example:***

(R1 LEFT JOIN R2) RIGHT JOIN R3  
-- may give different result than  
R1 LEFT JOIN (R2 RIGHT JOIN R3)

***Example:***

Problem: List student information and the CSCI class information. Include all students, leaving blanks when appropriate  
(i.e., no CSCI courses enrolled by the student).

+--------+-----------------+---------+-------------+-------+  
| stuId  | student         | classId | CSCI course | grade |  
+--------+-----------------+---------+-------------+-------+  
| 100000 | Tony Hawk       | 10000   | CSCI 3333   | A     |  
| 100000 | Tony Hawk       | 10001   | CSCI 4333   | A     |  
| 100000 | Tony Hawk       | 10002   | CSCI 5333   | B+    |  
| 100000 | Tony Hawk       | 11001   | CSCI 4333   | D     |  
| 100001 | Mary Hawk       | 10000   | CSCI 3333   |       |  
| 100001 | Mary Hawk       | 10001   | CSCI 4333   | A-    |  
| 100002 | David Hawk      | 10000   | CSCI 3333   | B-    |  
| 100002 | David Hawk      | 10002   | CSCI 5333   | B+    |  
| 100003 | Catherine Lim   |         |             |       |  
| 100004 | Larry Johnson   |         |             |       |  
| 100005 | Linda Johnson   |         |             |       |  
| 100006 | Lillian Johnson |         |             |       |  
| 100007 | Ben Zico        |         |             |       |  
| 100008 | Bill Ching      |         |             |       |  
| 100009 | Linda King      |         |             |       |  
| 100111 | Cathy Johanson  |         |             |       |  
+--------+-----------------+---------+-------------+-------+  
16 rows in set (0.001 sec)

***Example:*** (advanced)

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

-- List student information and the CSCI class information.  
SELECT DISTINCT s.stuId,  
   CONCAT(s.fname, ' ', s.lname) AS student,  
   e.classId,  
   CONCAT(co.rubric, ' ', co.number) AS `CSCI course`,  
   e.grade  
FROM student AS s INNER JOIN enroll AS e USING (stuId)  
   INNER JOIN class AS c USING (classId)  
   INNER JOIN course AS co USING (courseId)  
WHERE co.rubric = 'CSCI';  
  
-- List student information and the CSCI class information.  
-- Include all students, leaving blanks when appropriate  
-- (i.e. no CSCI courses enrolled by the student).  
  
-- These do not do the job. Why?  
SELECT DISTINCT s.stuId,  
   CONCAT(s.fname, ' ', s.lname) AS student,  
   IFNULL(e.classId, '') AS classId,  
   IFNULL(CONCAT(co.rubric, ' ', co.number), '') AS `CSCI course`,  
   IFNULL(e.grade, '') AS grade  
FROM student AS s LEFT JOIN enroll AS e USING (stuId)  
   LEFT JOIN class AS c USING (classId)  
   LEFT JOIN course AS co USING (courseId)  
WHERE co.rubric = 'CSCI';  
  
SELECT DISTINCT s.stuId,  
   CONCAT(s.fname, ' ', s.lname) AS student,  
   IFNULL(e.classId, '') AS classId,  
   IFNULL(CONCAT(co.rubric, ' ', co.number), '') AS `CSCI course`,  
   IFNULL(e.grade, '') AS grade  
FROM student AS s LEFT JOIN enroll AS e USING (stuId)  
   LEFT JOIN class AS c USING (classId)  
   LEFT JOIN course AS co ON (c.courseId = co.courseId AND co.rubric = 'CSCI' );  
  
-- This works. Note the LEFT JOIN and RIGHT JOIN.  
SELECT DISTINCT s.stuId,  
   CONCAT(s.fname, ' ', s.lname) AS student,  
   IFNULL(e.classId, '') AS classId,  
   IFNULL(CONCAT(co.rubric, ' ', co.number), '') AS `CSCI course`,  
   IFNULL(e.grade, '') AS grade  
FROM enroll AS e INNER JOIN class AS c USING (classId)  
   INNER JOIN course AS co ON (c.courseId = co.courseId AND co.rubric = 'CSCI' )  
   RIGHT JOIN student AS s USING (stuId);

* Note:
  + The inclusion of the condition co.rubric = 'CSCI' in the INNER JOIN condition.
  + The student table should be joined the last using RIGHT JOIN.

**3.4 Subqueries**

* A SQL subquery is a nested/inner subquery within a SQL statement or another query (for SELECT, INSERT, UPDATE or, DELETE).
* Subqueries usually appear in the FROM clause (as derived tables) and the WHERE clause.

***Example***

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

-- subqueries in the WHERE course  
-- students not enrolled in any class.  
SELECT DISTINCT \*  
FROM student AS s  
WHERE s.stuId NOT IN (SELECT DISTINCT e.stuID FROM enroll AS e);  
  
-- student with the maximum number of ach.  
SELECT DISTINCT MAX(ach)  
FROM student;  
  
-- student within 60 credits of the maximum number of ach any student may have.  
SELECT DISTINCT s.stuId,  
   CONCAT(s.fname, ' ', s.lname) AS student,  
   s.ach AS credits  
FROM student AS s  
WHERE s.ach + 60 >=  
   (SELECT DISTINCT MAX(ach) FROM student);  
  
-- subqueries as derived tables.  
SELECT DISTINCT s.stuId,  
   CONCAT(s.fname, ' ', s.lname) AS student,  
   s.ach AS credits  
FROM student AS s INNER JOIN  
   (SELECT DISTINCT MAX(ach) AS max FROM student) AS m -- an alias is required.  
WHERE s.ach + 60 >= m.max;  
  
**3.5 Common Table Expressions (CTE)**

* Supported by MySQL 8.0 and forward.
* Allow the definition of temporary common tables in a sequence before the body of a SELECT statement.
  + WITH t1 AS (definition of t1, a query...), t2 AS (...), ..., tn AS () SELECT ...
* A table defined in CTE can be used immediately until the end of the SELECT statement.
* Support a more natural way to implement *algorithmic solutions*, an (n+1) step solutions.
  + step 1 to n: constructions of the common tables t1, t2, ..., tn
  + step (n+1): the body of the SELECT statement.
* Allow recursion.
* May degrade performance.
* It is generally better than subqueries in the FROM clauses.
  + Tables in CTE can be used immediately after their definitions.
  + More natural order.
  + Can use recursion.

***Example:***

-- CTE  
WITH  t1 AS  
   (SELECT MAX(ach) AS max FROM student)  
SELECT s.stuId,  
   s.ach AS `ach credits`,  
   t1.max - s.ach AS `diff from max credits of all`  
FROM student AS s, t1  
ORDER BY `ach credits` DESC;  
  
-- multiple common tables (not efficient; used as demonstration.)  
WITH t1 AS  
   (SELECT MAX(ach) AS max FROM student),  
t2 AS  
   (SELECT s.stuId,  
      s.ach AS `ach credits`,  
      t1.max - s.ach AS diff,  
      s.major  
    FROM student AS s, t1)  
SELECT t2.stuId, t2.`ach credits`,  
   t2.diff AS `diff from max credits of all`,  
   d.deptName AS department  
FROM t2 LEFT JOIN department d ON (t2.major = d.deptCode)  
ORDER BY t2.`ach credits` DESC;

For those interesting in recursive CTE, here is an example. Recursive CTE will not be in the examinations.

Create and populate a simple relation that stores EmpId of an employee and the EmpId of the immediate supervisor.

CREATE SCHEMA CTETinker;  
USE SCHEMA CTEtinker;  
CREATE OR REPLACE TABLE Employee (  
    EmpId CHAR(7) NOT NULL,  
    SupervisorEmpId CHAR(7) NULL,  
    CONSTRAINT Emp\_EmpId\_pk PRIMARY KEY (EmpId),  
    CONSTRAINT Emp\_SupervisorEmpId\_fk FOREIGN KEY (SupervisorEmpId)  
        REFERENCES Employee(EmpId)  
);  
  
INSERT INTO Employee(EmpId, SupervisorEmpId) VALUES  
   ('E3', null);  
INSERT INTO Employee(EmpId, SupervisorEmpId) VALUES  
   ('E15', 'E3');  
INSERT INTO Employee(EmpId, SupervisorEmpId) VALUES  
   ('E50', 'E15');  
INSERT INTO Employee(EmpId, SupervisorEmpId) VALUES  
   ('E75', 'E50');  
INSERT INTO Employee(EmpId, SupervisorEmpId) VALUES  
   ('E100', 'E75');  
INSERT INTO Employee(EmpId, SupervisorEmpId) VALUES  
   ('E102', 'E75');  
INSERT INTO Employee(EmpId, SupervisorEmpId) VALUES  
   ('E70', 'E50');  
INSERT INTO Employee(EmpId, SupervisorEmpId) VALUES  
   ('E103', 'E70');  
    
SELECT \* FROM Employee;  
  
Result:

MariaDB [temp]> SELECT \* FROM Employee;  
+-------+-----------------+  
| EmpId | SupervisorEmpId |  
+-------+-----------------+  
| E3    | NULL            |  
| E50   | E15             |  
| E15   | E3              |  
| E70   | E50             |  
| E75   | E50             |  
| E103  | E70             |  
| E100  | E75             |  
| E102  | E75             |  
+-------+-----------------+  
8 rows in set (0.002 sec)  
   
A recursive CTE SQL to get all supervisors of employee 'E100':

WITH RECURSIVE Super(SEId) AS  
(  SELECT SupervisorEmpId AS SEId FROM Employee AS e WHERE e.EmpId = 'E100' -- initial condition/action  
   UNION ALL -- union all: add rows created by the recursive action to the result, table Super.  
   SELECT e.SupervisorEmpId AS SEId -- recursive action  
      FROM Employee AS e INNER JOIN Super  
      WHERE e.EmpId = Super.SEId  
      AND e.SupervisorEmpId IS NOT NULL  
      -- exit condition: when the recursive action returns an empty table.  
)  
SELECT \*  
FROM Super;

Result:

+------+  
| SEId |  
+------+  
| E75  |  
| E50  |  
| E15  |  
| E3   |  
+------+

DROP SCHEMA IF EXISTS CTEtinker;

**3.6 GROUP BY and HAVING**

* Useful for group reports: one result row per group, not per row as in the regular SELECT statement without GROUP BY.
* Allow aggregate functions (also known as *group functions* and column functions) to be performed by the groups defined.
* Output one row per *group*.
* A group is defined by an unique value of the columns in the group by clause.
* Example aggregate functions: MAX, MIN, AVG, COUNT, SUM, GROUP\_CONCAT, etc. See: <https://dev.mysql.com/doc/refman/8.0/en/aggregate-functions.html>
* The HAVING clause allows using group functions in the condition. The WHERE clause does not allow using group functions.
* Using GROUP BY, the columns of the SELECT clause can only have:
  1. Columns named in GROUP BY.
  2. Aggregate/group functions on other columns in the tables.
  3. Constant expressions.
* A number can be used in the GROUP BY and the ORDER BY clauses to refer to the positions of the result columns in the select clauses.

Thus, the conceptual steps and framework for the SELECT statement become

SELECT DISTINCT <<result\_columns>> -- [5] construct result columns  
FROM <<source\_tables>> -- [1] conceptually join tables to form a large table to produce initial rows  
WHERE <<conditions\_for\_inclusion>> -- [2] Filter initial rows  
GROUP BY <<group\_by\_columns>>  
         --[3] group initial rows into groups by values of the group\_by\_column. A group becomes a new row.  
HAVING <<conditions for filtering group>> -- [4] filter groups  
ORDER BY <<columns>>; -- [6] Order the result of [5].

***Example:***

-- Student names and number of classes enrolled.  
-- More than 2 classes to be included in the result.  
SELECT CONCAT(s.fname, ' ', s.lname) AS student,  
   COUNT(e.classId) AS `Enrolled classes`  
FROM student AS s INNER JOIN enroll e ON (s.stuId = e.stuId)  
GROUP BY student  
HAVING `Enrolled classes` > 2  
ORDER BY `Enrolled classes` DESC;

***Exercises***:

[1] Write a query to generate the student names and number of courses enrolled, including those not enrolled?

+-----------------+------------------+  
| name            | Enrolled classes |  
+-----------------+------------------+  
| Tony Hawk       |                6 |  
| Linda Johnson   |                4 |  
| David Hawk      |                3 |  
| Ben Zico        |                2 |  
| Larry Johnson   |                2 |  
| Mary Hawk       |                2 |  
| Lillian Johnson |                2 |  
| Bill Ching      |                1 |  
| Catherine Lim   |                0 |  
| Linda King      |                0 |  
+-----------------+------------------+  
10 rows in set (0.00 sec)  
  
Solution:

SELECT CONCAT(s.fname, ' ', s.lname) AS student,  
    COUNT(e.classId) AS `Enrolled classes`  
FROM student AS s LEFT JOIN enroll e ON (s.stuId = e.stuId)  
GROUP BY student   
ORDER BY `Enrolled classes` DESC; 

[2] Can you write a query to generate the following output?

+----------+------------------------------+------------+----------+----------+  
| deptCode | deptName                     | numFaculty | numMajor | numMinor |  
+----------+------------------------------+------------+----------+----------+  
| ACCT     | Accounting                   |          1 |        0 |        0 |  
| ARTS     | Arts                         |          1 |        2 |        0 |  
| CINF     | Computer Information Systems |          2 |        2 |        3 |  
| CSCI     | Computer Science             |          4 |        3 |        1 |  
| ENGL     | English                      |          1 |        0 |        2 |  
| ITEC     | Information Technology       |          2 |        2 |        2 |  
| MATH     | Mathematics                  |          0 |        0 |        0 |  
+----------+------------------------------+------------+----------+----------+  
7 rows in set (0.00 sec)

Solution:

WITH ma AS  
   (SELECT s.major AS deptCode, COUNT(s.stuId) AS numMajor  
    FROM student AS s  
    GROUP BY s.major),  
mi AS  
   (SELECT s.minor AS deptCode, COUNT(s.stuId) AS numMinor  
    FROM student AS s  
    GROUP BY s.minor),  
f AS  
   (SELECT f.deptCode, COUNT(f.facId) AS numFaculty  
    FROM faculty AS f  
    GROUP BY f.deptCode)  
SELECT d.deptCode,  
   d.deptName,  
   IFNULL(f.numFaculty, 0) AS numFaculty,  
   IFNULL(ma.numMajor, 0) AS numMajor,  
   IFNULL(mi.numMinor, 0) AS numMinor  
FROM department AS d LEFT JOIN ma USING (deptCode)  
   LEFT JOIN mi USING (deptCode)  
   LEFT JOIN f USING (deptCode);

**3.7 Window Functions**

* MySQL 8.x supports Window functions.
* A window function performs a computation on a set of rows (a window frame) in which the current row is in the window frame.
* It is not a clause.
* Unlike the GROUP BY clause, it does*not* form groups.
* The OVER clause is used to define the window frame.
* OVER(): all rows are in the window frame.
* OVER(PARTITION BY X): each X value defines a window frame.
* Many aggregate functions can be used by Window functions.
* Modern DBMS support a rich set of Window functions.

***Example:***

WITH temp AS  
(SELECT DISTINCT sc.schoolName AS college, d.deptName AS department,  
   COUNT(s.stuId) As deptMajor  
FROM school AS sc INNER JOIN department AS d ON (sc.schoolCode = d.schoolCode)  
   LEFT JOIN student AS s ON (s.major = d.deptCode)  
GROUP BY college, department)  
SELECT temp.college, temp.department,  
   temp.deptMajor AS `major in department`,  
   SUM(deptMajor) OVER(PARTITION BY college) AS `major in college`,  
   SUM(deptMajor) OVER() AS `major in university`  
FROM temp;

Please execute to see the output.

Adding row number and rank:

WITH ma AS  
(SELECT s.major AS deptCode, COUNT(s.stuId) AS numMajor  
FROM student AS s  
GROUP BY s.major),  
mi AS  
(SELECT s.minor AS deptCode, COUNT(s.stuId) AS numMinor  
FROM student AS s  
GROUP BY s.minor),  
f AS  
(SELECT f.deptCode, COUNT(f.facId) AS numFaculty  
FROM faculty AS f  
GROUP BY f.deptCode)  
SELECT ROW\_NUMBER() OVER () AS `#`,  
   RANK() OVER (ORDER BY f.numFaculty DESC) AS `# in descending number of faculty`,  
   d.deptCode,  
   d.deptName,  
   IFNULL(f.numFaculty, 0) AS numFaculty,  
   IFNULL(ma.numMajor, 0) AS numMajor,  
   IFNULL(mi.numMinor, 0) AS numMinor  
FROM department AS d LEFT JOIN ma USING (deptCode)  
   LEFT JOIN mi USING (deptCode)  
   LEFT JOIN f USING (deptCode);

**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.
  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/Spring2025/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
  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 //  
  
CREATE OR REPLACE PROCEDURE deptInfo(IN dCode VARCHAR(4), OUT numFaculty INT)  
BEGIN  
   --  Display some information.  
   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**

* 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 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.